Applied Choreographies

Saverio Giallorenzo, Fabrizio Montesi, Maurizio Gabbrielli

Applied Choreographies

Saverio Giallorenzo[1], Fabrizio Montesi[1], and Maurizio Gabbrielli[2]

[1] University of Southern Denmark
[2] Università di Bologna/INRIA

Abstract. Choreographic Programming is a paradigm for distributed programming, where high-level "Alice and Bob" descriptions of communications (choreographies) are used to synthesise correct-by-construction programs. However, implementations of choreographic models use message routing technologies distant from their related theoretical models (e.g., CCS/π channels). This drives implementers to mediate discrepancies with the theory through undocumented, unproven adaptations, weakening the reliability of their implementations.
As a solution, we propose the framework of Applied Choreographies (AC). In AC, programmers write choreographies in a language that follows the standard syntax and semantics of previous works. Then, choreographies are compiled to a real-world execution model for Service-Oriented Computing (SOC). To manage the complexity of this task, our compilation happens in three steps, respectively dealing with: implementing name-based communications using the concrete mechanism found in SOC, projecting a choreography to a set of processes, and translating processes to a distributed implementation in terms of services.

1 Introduction

Background. In Choreographic Programming, programs are choreographies of communications used to synthesise correct implementations through an End-Point Projection (EPP) procedure [1]. The generated code is guaranteed to follow the behaviour specified in the choreography and to be deadlock-free [2]. For these reasons, the communities of business processes and Service-Oriented Computing (SOC) widely adopted choreographies, using them in standards (e.g., WS-CDL [3] and BPMN [4]), languages [5,6,7,8,9,10,11,12], and type systems and logics [13,14,15,16].

Example 1. Below, we give a representative example of a choreography. In the example, we implement a simple business protocol among a client process c, a seller service located at $\mathsf{l_S}$ and a bank service located at $\mathsf{l_B}$ (locations are abstractions of network addresses, or URIs). At line 1, the client c asks the seller and the bank services to create two new processes, respectively s and b. The three processes c, s, and b can now communicate over a multiparty session k, intended as in Multiparty Session Types [13]: when a session is created, each process gets ownership of a statically-defined *role*, which identifies a message queue that the process uses to asynchronously receive messages from other processes. For

simplicity, at line 1, we assign role C to process c, S to s, and B to b. At line 2, over session k, the client c invokes operation buy of the seller s with the name of a product it wishes to buy, which the seller stores in its local variable x. As usual, processes have local state and run concurrently. At line 3, the seller uses its internal function mkOrder to prepare an order (e.g., compute the price of the product) and sends it to the bank on operation openTx, for opening a payment transaction. At line 4, the client sends its credit card information cc to the bank on operation pay. Then, at line 5, the bank makes an internal choice on whether the payment can be performed (with internal function closeTx, which takes the local variables cc and order as parameters). The bank then notifies the client and the seller of the final outcome, by invoking them both either on operation ok or ko.

```
1  start k : c[C] <=> l_S.s[S], l_B.b[B];
2  k:c[C].product --> s[S].buy(x);
3  k:s[S].mkOrder(x) --> b[B].openTx(order);
4  k:c[C].cc --> b[B].pay(cc);
5  if b.closeTx(cc, order)
6   { k:b[B] --> c[C].ok(); k:b[B] --> q[S].ok() }
7  else
8   { k:b[B] --> c[C].ko(); k:b[B] --> q[S].ko() }
```

Motivation. In previous definitions of EPP, both the choreography language and the target language abstract from how real-world frameworks support communications [17,5,14,2,15], by modelling communications as synchronisations on *names* (cf. [18,19]). Thus, the implementations of choreographic programming [11,12] significantly depart from their respective formalisations [2,8]. In particular, the implemented EPPs realise channel creation and message routing with additional data structures and message exchanges [1,20]. The specific communication mechanism used in these implementations is message correlation. Correlation is the reference message routing technology in Service-Oriented Computing (SOC)—the major field of application of choreographies—and it is supported by mainstream technologies (e.g., WS-BPEL [21], Java/JMS, C#/.NET). The gap between formalisations and implementations of choreographic programming can compromise its correctness-by-construction guarantee.

Contributions. We reduce the gap between choreographies and their implementations by developing Applied Choreographies, a choreographic framework consisting of three calculi: the Frontend Calculus (FC), which offers the well-known simplicity of abstract channel semantics to programmers; the Backend Calculus (BC), which formalises how abstract channels can be implemented on top of message correlation; and the Dynamic Correlation Calculus (DCC), an abstract model of Service-Oriented Computing where distributed services communicate through correlation. Differently from BC, DCC has no global view on the state of the system (which is instead distributed), and there are no multiparty synchronisation primitives.

Our main contribution is the definition of a behaviour-preserving compiler from choreographies in FC to distributed services in DCC, which uses BC as

intermediate representation. This is the first correctness result of an end-to-end translation from choreographies to an abstract model based on a real-world communication mechanism. Our compiler proceeds in three steps:

- it projects (EPP) a choreography, describing the behaviours of many participants, into a composition of *modules* called *endpoint choreographies*, each describing the behaviour of a single participant;
- it generates the data structures needed by BC to support the execution of the obtained endpoint choreographies using message correlation;
- it synthesises a correct distributed implementation in DCC, where the multiparty synchronisations in choreographies are translated to correct distributed handshakes (consisting of many communications) on correlation data.

Full definitions and proof sketches are in [22].

2 Frontend Calculus

We start by presenting the Frontend Calculus (FC), which follows the structure of Compositional Choreographies [23]. The main novelty is in the semantics, which is based on a notion of deployment and deployment effects that will ease the definition of our translation. Remarkably, it also yields a new concise formalisation of asynchrony in choreographies.

Syntax. In the syntax of FC (Fig. 1), C denotes an FC program. FC programs are choreographies, as in Example 1, and we often refer to them as Frontend choreographies. A choreography describes the behaviour of some processes. Processes, denoted $\mathsf{p}, \mathsf{q} \in \mathcal{P}$, are intended as usual: they are independent execution units running concurrently and equipped with local variables, denoted $\mathsf{x} \in \mathit{Var}$. Message exchanges happen through a session, denoted $\mathsf{k} \in \mathcal{K}$, which acts as a communication channel. Intuitively, a session is an instantiation of a protocol, where each process is responsible for implementing the actions of a role defined in the protocol. We denote roles with $\mathsf{A}, \mathsf{B} \in \mathcal{R}$. A process can create new processes and sessions at runtime by invoking service processes (services for short). Services are always available at fixed locations, denoted $\mathsf{l} \in \mathcal{L}$, meaning that they can be used multiple times (in process calculus terms, they act as replicated processes [24]). Locations are novel to choreographies: they are addresses of always-available services able to create processes at runtime (cf. public channels in [2,23]).

Processes communicate by exchanging messages. A message consists of two elements: *i*) a payload, representing the data exchanged between two processes; and *ii*) an operation, which is a label used by the receiver to determine what it should do with the message—in object-oriented programming, these labels are called method names [25]; in service-oriented computing, labels are typically called operations as in here. Operations are denoted $\mathsf{o} \in \mathcal{O}$.

In the rest of the section, we comment the syntax of FC and present its semantics. However, before doing that, we dedicate the next paragraph to illustrate with an example the purpose and relationship between complete and partial actions.

$$
\begin{array}{lllll}
C ::= & \eta; C & (seq) & | \quad C_1 \mid C_2 & (par) \\
| & \texttt{if}\ p.e\ \{C_1\}\ \texttt{else}\ \{C_2\} & (cond) & | \quad \mathbf{0} & (inact) \\
| & \texttt{def}\ X = C'\ \texttt{in}\ C & (rec) & | \quad X & (call) \\
| & k{:}A \longrightarrow q[B].\{o_i(x_i); C_i\}_{i\in I} & (recv) & & \\
\\
\eta ::= & k{:}p[A].e \longrightarrow q[B].o(x) & (com) & | \quad \texttt{start}\ k : p[A] \Longleftrightarrow \widetilde{l.q[B]} & (start) \\
| & k{:}p[A].e \longrightarrow B.o & (send) & | \quad \texttt{req}\ k : p[A] \Longleftrightarrow \widetilde{l.B} & (req) \\
& & & | \quad \texttt{acc}\ k : \widetilde{l.q[B]} & (acc)
\end{array}
$$

Fig. 1. Frontend Calculus, syntax.

Complete and Partial Actions. As in [23], FC supports modular development by allowing choreographies, say C and C', to be composed in parallel, written $C \mid C'$. A parallel composition of choreographies is also a choreography, which can thus be used in further parallel compositions. Composing two choreographies in parallel allows the processes in the two choreographies to interact over shared location and session names. In particular, we distinguish between two kinds of terms inside of a choreography: complete and partial actions. A complete action is internal to the system defined by the choreography, and thus does not have any external dependency. By contrast, a partial action defines the behaviour of some processes that need to interact with another choreography in order to be executed. Therefore, a choreography containing partial actions needs to be composed with other choreographies that provide compatible partial actions. To exemplify the distinction between complete and partial actions, let us consider the case of a single communication between two processes.

Complete interaction	*Composed partial actions*
$k{:}c[C].\texttt{product} \longrightarrow s[S].\texttt{buy}(\ x\ )$	$k{:}c[C].\texttt{product} \longrightarrow S.\texttt{buy}$ $\mid$ $k{:}C \longrightarrow s[S].\texttt{buy}(\ x\ )$

Above, on the left we have the communication statement as written at line 2 of Example 1. This is a complete action: it defines exactly all the processes that should interact (`c` and `s`). On the right, we implement the same action as the parallel composition of two *modules*, i.e., choreographies with partial actions. At the left of the parallel we write a send action, performed by process `c` to role `S` over session `k`, at the right of the parallel we write the complementary reception from a role `C` and performed by process `s` over the session `k`. More specifically, we read the send action as "process `c` sends a message as role `C` with payload `product` for operation `buy` to the process playing role `S` on session `k`". Dually, we read the receive action as "process `s` receives a message for role `S` and operation `buy` over session `k` and stores the payload in variable `x`". The compatible roles,

session, and operation used in the two partial actions make them compliant. Thus, the choreography on the left is operationally equivalent to the one on the right. Observe that partial actions do not mention the name of the process on the other end—for example, the send action performed by process c does not specify that it wishes to communicate with process s precisely. This supports some information hiding: a partial action in a choreography can interact with partial actions in other choreographies, independently from the process names used in the latter. Expressions and variables used by senders and receivers are also kept local to statements that define local actions.

By equipping FC with both partial and complete actions, we purposefully made FC non-minimal. However, while a minimal version of FC (i.e., equipped with either partial or complete actions) can capture well-known choreographic models like [5,6,7,2,8], we included both kinds of actions to describe a comprehensive language for programmers. On the one hand, complete actions offer a concise syntax to express closed systems, as found in other choreographic models. On the other hand, partial actions support compositionality in Frontend choreographies, allowing developers to write FC modules separately and possibly reuse the same module in multiple compositions.

Complete Actions. We start commenting the syntax of FC from complete actions, marked with a shade in Fig. 1. In term (*start*), process p starts a new session k together with some new processes $\tilde{\mathsf{q}}$ ($\tilde{\mathsf{q}}$ is assumed non-empty). Process p, called *active process*, is already running, whereas each process q in $\widetilde{\mathsf{l}.\mathsf{q}}$, called *service process*, is dynamically created at the respective location l. Each process is annotated with the role it plays in the session. Term (*com*) models a communication: on session k, process p sends to process q a message for its operation o; the message carries the evaluation of expression e (we assume expressions to consist of standard computations on local variables) on the local state of p whilst x is the variable where q will store the content of the message.

Partial Actions. Partial actions correspond to the terms obtained by respectively splitting the (*start*) and (*com*) complete terms into their partial counterparts. In term (*req*), process p requests some external services respectively located at $\tilde{\mathsf{l}}$, to create some external processes and start a new session k. By "external", we mean that the behaviour of such processes is defined in a separate choreography module, to be composed in parallel. Role annotations follow those of term (*start*). Term (*acc*) is the dual of (*req*) and defines a choreography module that provides the implementation of some service processes. We assume (*acc*) terms to always be at the top level (not guarded by other actions), capturing always-available choreography modules. In term (*send*), process p sends a message to an external process that plays $\mathtt{B}$ in session k. Dually, in term (*recv*), process q receives a message for one of the operations o_i from an external process playing role $\mathtt{A}$ in session k, and then proceeds with the corresponding continuation (cf. [26]).

Other Terms. In a conditional (*cond*), process p evaluates a condition e in its local state to choose between the continuations C_1 and C_2. Term (*par*) is standard parallel composition, which allows partial actions in two choreographies C_1 and C_2 to interact. Respectively, terms (*def*), (*call*), and (*inact*) model the

definition of recursive procedures, procedure calls, and inaction. Some terms bind identifiers in continuations. In terms (*start*) and (*acc*), the session identifier k and the process identifiers $\tilde{\mathsf{q}}$ are bound (as they are freshly created). In terms (*com*) and (*recv*), the variables used by the receiver to store the message are bound (x and all the x_i, respectively). In term (*req*), the session identifier k is bound. Finally, in term (*def*), the procedure identifier X is bound. In the remainder, we omit $\mathbf{0}$ or irrelevant variables (e.g., in communications with empty messages). Terms (*com*), (*send*), and (*recv*) include role annotations only for clarity reasons; roles in such terms can be inferred, as shown in [1].

Semantics. The semantics of FC follows the standard principles of asynchronous multiparty sessions (cf. [13]) and it is formalised in terms of reductions of the form $\mathsf{D}, \mathsf{C} \to \mathsf{D}', \mathsf{C}'$, where D is a deployment. Deployments store the states of processes and the message queues that support message exchange in sessions. We start by formalising the notion of deployment.

Deployment. As in multiparty session types [13], we equip each pair of roles in a session with a dedicated asynchronous queue to communicate in each direction. Formally, let $\mathcal{Q} = \mathcal{K} \times \mathcal{R} \times \mathcal{R}$ be the set of all *queue identifiers*; we write $\mathsf{k}[\mathsf{A}\rangle\mathsf{B}] \in \mathcal{Q}$ to identify the queue from A to B in session k. Now, we define D as an overloaded partial function defined by cases as the sum of two partial functions $\mathsf{f_s} : \mathcal{P} \rightharpoonup \mathit{Var} \rightharpoonup \mathit{Val}$ and $\mathsf{f_q} : \mathcal{Q} \rightharpoonup \mathsf{Seq}(\mathcal{O} \times \mathit{Val})$ whose domains and co-domains are disjoint: $\mathsf{D} = \mathsf{f_s}(z)$ if $z \in \mathcal{P}$, $\mathsf{f_q}(z)$ if $z \in \mathcal{Q}$.

Function $\mathsf{f_s}$ maps a process p to its state, which is a partial function from variables $\mathsf{x}, \mathsf{y} \in \mathit{Var}$ to values $\mathsf{v} \in \mathit{Val}$. Function $\mathsf{f_q}$ stores the queues used in sessions. Each queue is a sequence of messages $\tilde{\mathsf{m}} = \mathsf{m}_1 :: \ldots :: \mathsf{m}_n \mid \varepsilon$ (ε is the empty queue), where each message $\mathsf{m} = (\mathsf{o}, \mathsf{v}) \in \mathcal{O} \times \mathit{Val}$ contains the operation o it was received on and the payload v.

Programmers do not deal with deployments. We assume that choreographies without free session names start execution with a *default deployment* that contains empty process states. Let $\mathbf{fp}(\mathsf{C})$ return the set of free process names in C and, with abuse of notation, $\varnothing$ be the undefined function for any given signature.

Definition 1 (Default Deployment). *Let* C *be a choreography without free session names. The default deployment of* D *for* C *is defined as the function mapping all free names in* C *to empty states, i.e.,* $\mathsf{D}[\mathsf{p} \mapsto \varnothing \mid \mathsf{p} \in \mathbf{fp}(\mathsf{C})]$.

Reductions. In our semantics, choreographic actions have effects on the state of a system—deployments change during execution. At the same time, a deployment also determines which choreographic actions can be performed. For example, a communication from role $\mathtt{A}$ to role $\mathtt{B}$ over session k requires a queue $\mathsf{k}[\mathtt{A}\rangle\mathtt{B}]$ to exist in the deployment of the system.

We formalise the notion of which choreographic actions are allowed by a deployment and their effects using transitions of the form $\mathsf{D} \xrightarrow{\delta} \mathsf{D}'$, read "the deployment D allows for the execution of δ and becomes D' as the result". Actions δ are defined by the following grammar: $\mathtt{start}\ \mathsf{k} : \mathsf{p}[\mathtt{A}] \Longleftrightarrow \overline{\mathsf{l}.\mathsf{q}[\mathtt{B}]}$, session start, $\mathsf{k}:\mathsf{p}[\mathtt{A}].e \longrightarrow \mathtt{B}.\mathsf{o}$, send in session, $\mathsf{k}:\mathtt{A} \longrightarrow \mathsf{q}[\mathtt{B}].\mathsf{o}(\mathsf{x})$, receive in session.

We report the rules defining $\mathsf{D} \xrightarrow{\delta} \mathsf{D}'$ in the top-half of Fig. 2.

$$\lfloor \mathsf{D}|_{\mathsf{Start}} \rceil \quad D \xrightarrow{\text{start } k\colon p[A] \iff \widetilde{l.q[B]}} D\left[q \mapsto \varnothing \mid q \in \{\tilde{q}\}\right]\left[k[C\rangle E] \mapsto \varepsilon \mid \{C,E\} \subseteq \{A,\tilde{B}\}\right]$$

$$\lfloor \mathsf{D}|_{\mathsf{Send}} \rceil \quad e\downarrow_{D(p)} v \;\Rightarrow\; D\left[k[A\rangle B] \mapsto m\right] \xrightarrow{k\ p[A].e \longrightarrow B.o} D\left[k[A\rangle B] \mapsto \tilde{m} :: (o,v)\right]$$

$$\lfloor \mathsf{D}|_{\mathsf{Recv}} \rceil \quad D\left[k[A\rangle B] \mapsto (o,v) :: \tilde{m}\right] \xrightarrow{k\ A \longrightarrow q[B].o(x)} D\left[k[A\rangle B] \mapsto \tilde{m}, q \mapsto D(q)[x \mapsto v]\right]$$

$$\lfloor \mathsf{C}|_{\mathsf{Start}} \rceil \begin{cases} \delta = \mathtt{start}\ k : p[A] \iff \widetilde{l.q[B]} \\ \wedge\ D\#k', \tilde{r} \ \wedge\ D \xrightarrow{\delta[k'/k][\tilde{r}/\tilde{q}]} D' \end{cases} \Rightarrow\ D,\ \delta; C \to D', C[k'/k][\tilde{r}/\tilde{q}]$$

$$\lfloor \mathsf{C}|_{\mathsf{Send}} \rceil \quad \eta = k\colon p[A].e \longrightarrow B.o \ \wedge\ D \xrightarrow{\eta} D' \quad \Rightarrow\ D,\ \eta; C \to D',\ C$$

$$\lfloor \mathsf{C}|_{\mathsf{Recv}} \rceil \begin{cases} \delta = k\colon A \longrightarrow q[B].o_j(x_j) \\ \wedge\ j \in I \ \wedge\ D \xrightarrow{\delta} D' \end{cases} \Rightarrow\ D,\ k\colon A \longrightarrow q[B].\{o_i(x_i); C_i\}_{i\in I} \to D',\ C_j$$

$$\lfloor \mathsf{C}|_{\mathsf{Eq}} \rceil \begin{cases} \mathcal{R} \in \{\equiv_C, \simeq_C\} \ \wedge\ C_1 \,\mathcal{R}\, C_1' \\ \wedge\ D, C_1' \to D', C_2' \ \wedge\ C_2' \,\mathcal{R}\, C_2 \end{cases} \Rightarrow\ D, C_1 \to D', C_2$$

Fig. 2. FC: top-half deployment transitions, bottom-half semantics (selected).

Rule $\lfloor \mathsf{D}|_{\mathsf{Start}} \rceil$ states that the creation of a new session k between an existing process p and new processes $\tilde{q}$ results in updating the deployment with: a new (empty) state for each of the new processes q in $\tilde{q}$ ($\left[q \mapsto \varnothing \mid q \in \{\tilde{q}\}\right]$); and a new (empty) queue between each pair of distinct roles in the session ($\left[k[C\rangle E] \mapsto \varepsilon \mid \{C,E\} \subseteq \{A,\tilde{B}\}\right]$). Rule $\lfloor \mathsf{D}|_{\mathsf{Send}} \rceil$ models the effect of a send action. In the premise, we use the auxiliary function $\downarrow$ to evaluate the local expression e in the state of process p, obtaining the value v to use as message payload. Then, in the conclusion, we append a message with its payload—(o,v)—to the end of the queue from the sender's role to the receiver's role ($k[A\rangle B]$). We assume that function $\downarrow$ always terminates—in practice, this can be obtained by using timeouts. Rule $\lfloor \mathsf{D}|_{\mathsf{Recv}} \rceil$ models the effect of a reception. If the queue $k[A\rangle B]$ contains in its head a message on operation o, it is always possible to remove it from the queue and to store its value v under variable x in the state of the receiver.

Using deployment transitions, we can now define the rules for reductions $D, C \to D', C'$. We call a configuration D, C a *running choreography*. The reduction relation $\to$ for FC is the smallest relation closed under the (excerpt of) rules given in the bottom-half of Fig. 2. Rule $\lfloor \mathsf{C}|_{\mathsf{Start}} \rceil$ creates a new session, by ensuring that both the new session name k' and new processes $\tilde{r}$ are fresh wrt D ($D\#k', \tilde{r}$). We use the fresh names in the continuation C, by using a standard substitution $C[k'/k][\tilde{r}/\tilde{q}]$. Rule $\lfloor \mathsf{C}|_{\mathsf{Send}} \rceil$ reduces a send action, if it is allowed by the deployment. Rule $\lfloor \mathsf{C}|_{\mathsf{Recv}} \rceil$ reduces a message reception, if the deployment allows for receiving a message on one of the branches in the receive term ($j \in I$). Recalling the corresponding rule $\lfloor \mathsf{D}|_{\mathsf{Recv}} \rceil$, this can happen only if the deployment D has

a message for operation o_j in the queue $k[A\rangle B]$. Rule $[^{C}|_{Eq}]$ closes $\rightarrow$ under the congruences $\equiv_C$ and $\simeq_C$. Structural congruence $\equiv_C$ is the smallest congruence supporting α-conversion, recursion unfolding, and commutativity and associativity of parallel composition. The swap relation $\simeq_C$ is the smallest congruence able to exchange the order of non-interfering concurrent actions. Rule $[^{C}|_{Eq}]$ also enables the reduction of complete communications on (*com*) terms with the equivalence

$$k : p[A].e \longrightarrow q[B].o(x); C \equiv_C k : p[A].e \longrightarrow B.o; k : A \longrightarrow q[B].\{o(x); C\}$$

unfolding complete communications into the corresponding send and receive terms.

Typing. We equip FC with a typing discipline based on multiparty session types [13,26], which checks that partial actions composed in parallel are compatible. Our typing discipline is a straightforward adaptation of that presented for Compositional Choreographies [23], so we omit most details here (reported in [22]). However, the type system also gives us important information that will be critical in the compilation that we will develop later. Specifically, we mainly use types to keep track of the location that each process has been created at, the types of variables, the roles played by processes in open sessions, the role played by the processes spawned at each location in a choreography, and the status of message queues. Keeping track of this information is straightforward; hence, for brevity, we simply report the definition of the main elements of the typing environment Γ that we use to store it.

$$\Gamma ::= \Gamma,\ \tilde{l} : G\langle A|\tilde{B}|\tilde{C}\rangle \ \mid\ \Gamma, k[A] : T \ \mid\ \Gamma, p : k[A] \ \mid\ \Gamma, p.x : U \ \mid\ \Gamma, p@l \ \mid\ k[B\rangle A] : T \ \mid\ \emptyset$$

In Γ, a service typing $\tilde{l} : G\langle A|\tilde{B}|\tilde{C}\rangle$ types with the (standard, from [26,23]) global type G any session started by contacting the services at the locations $\tilde{l}$. Role A is the role of the active process, whereas roles $\tilde{B}$ are the respective roles of the service processes located at $\tilde{l}$. (The roles $\tilde{C}$ are used to keep track of whether the implementation of some roles is provided by external choreography modules.) A session typing $k[A] : T$ defines that role A in session k follows the local type T. An ownership typing $p : k[A]$ states that process p owns the role A in session k. A location typing $p@l$ states that process p runs at location l. The typing $p.x : U$ states that variable x has type U in the state of process p. Finally, the buffer typing $k[A\rangle B] : T$ states that the queue $k[A\rangle B]$ contains a sequence of messages typed by the local type T.

A typing judgement $\Gamma \vdash D, C$ establishes that D and C are typed according to Γ. If such a Γ exists, we say that D, C is well-typed.

Thanks to Γ, we can define a formal notion of coherence **co**, useful to check if a given set of Frontend modules can correctly execute a typed session:

Definition 2 (Coherence). $\mathbf{co}(\Gamma)$ *holds iff* $\forall\ k \in \Gamma\ ,\ \exists\ G$ *s.t.*

- $\tilde{l} : G\langle A|\tilde{B}|\tilde{C}\rangle \in \Gamma\ \wedge\ \tilde{C} = \tilde{B}$ *and*
- $\forall\ A \in \mathbf{roles}(G),\ k[A] : T \in \Gamma\ \wedge\ T = [\![G]\!]_A$
  $\wedge\ \forall\ B \in \mathbf{roles}(G) \setminus \{A\},\ \Gamma \vdash k[A\rangle B] : [\![G]\!]^A_B$

Coherence checks that *i*) all services needed to start new sessions are present and *ii*) all the roles in every open session are correctly implemented by some processes. To do this, given a global type G, **co** uses function $[\![\mathsf{G}]\!]_{\mathtt{A}}$ to extract the local type of a role $\mathtt{A}$ in G and function $[\![\mathsf{G}]\!]^{\mathtt{A}}_{\mathtt{B}}$ to extract the local type of the queue where role $\mathtt{B}$ receives from role $\mathtt{A}$ in G.

Well-typed Frontend choreographies that contain all necessary modules never deadlock.

Theorem 1 (Deadlock freedom). $\Gamma \vdash \mathsf{D}, \mathsf{C}$ *and* $\mathsf{co}(\Gamma)$ *imply that either (i)* $\mathsf{C} \equiv \mathbf{0}$ *or (ii) there exist* D' *and* C' *such that* $\mathsf{D}, \mathsf{C} \to \mathsf{D}', \mathsf{C}'$.

3 Backend Calculus

We now present the *Backend Calculus* (BC). Formally, the syntax of programs in BC is the same as that of FC. The only difference between BC and FC is in the semantics: we replace the notions of deployment and deployment effects with new versions that formalise message exchanges based on message correlation, as found in Service-Oriented Computing (SOC) [21].

Deployments in BC. Deployments in BC capture communications in SOC, where data trees are used to correlate messages to input queues. We first informally introduce correlation.

Message Correlation. Processes in SOC run within services and communicate asynchronously: each process can retrieve messages from an unbound number of FIFO queues, managed by its enclosing service. To identify queues, services use some data, called *correlation key*. When a service receives a message from the network, it inspects its content looking for a correlation key that points one of its queues. If a queue can be found, the message is enqueued in its tail. As noted in the example, messages in SOC contain correlation keys as either part their payload or in some separate header. As in [27], we abstract from such details.

Data and Process state. Data in SOC is structured following a tree-like format, e.g., XML or JSON. We use trees to represent both the payload of messages and the state of running processes (as in, e.g., BPEL [21] and Jolie [28]). Formally, we consider rooted trees $\mathtt{t} \in \mathcal{T}$, where edges are labelled by names, ranged over by $\underline{\mathtt{x}}$, and $\varnothing$ is the empty tree. We assume that all outgoing edges of a node have distinct labels and that only leaves contain values of type $\mathit{Val} \cup \mathcal{L}$, i.e., basic data (**int**, **str**, ...) or locations. Variables $\mathtt{x}$ are paths to traverse a tree: $\mathtt{x}, \mathtt{y} ::= \underline{\mathtt{x}}.\mathtt{x} \mid \varepsilon$, where ε is the empty path (often omitted). Given a path $\mathtt{x}$ and a tree $\mathtt{t}$, $\mathtt{x}(\mathtt{t})$ is the node reached by following $\mathtt{x}$ in $\mathtt{t}$; otherwise, $\mathtt{x}(\mathtt{t})$ is undefined. Abusing notation, when $\mathtt{x}(\mathtt{t})$ is a leaf, then $\mathtt{x}(\mathtt{t})$ denotes also the value of the node. To manipulate trees, we will use the (total) replacement operator $\mathtt{t} \triangleleft (\mathtt{x}, \mathtt{t}')$. If $\mathtt{x}(\mathtt{t})$ is defined, $\mathtt{t} \triangleleft (\mathtt{x}, \mathtt{t}')$ returns the tree obtained by replacing in $\mathtt{t}$ the subtree rooted in $\mathtt{x}(\mathtt{t})$ by $\mathtt{t}'$. If $\mathtt{x}(\mathtt{t})$ is undefined, $\mathtt{t} \triangleleft (\mathtt{x}, \mathtt{t}')$ adds the smallest chain of empty nodes to $\mathtt{t}$ such that $\mathtt{x}(\mathtt{t})$ is defined and it stores $\mathtt{t}'$ in $\mathtt{x}(\mathtt{t})$.

Backend Deployment. We can now define the notion of deployment for BC, denoted $\mathcal{D}$. Formally, $\mathcal{D}$ is an overloaded partial function defined by cases as

the sum of three partial functions $\mathsf{g}_\mathsf{l} : \mathcal{L} \rightharpoonup \mathsf{Set}(\mathcal{P})$, $\mathsf{g}_\mathsf{m} : \mathcal{L} \times \mathcal{T} \rightharpoonup \mathsf{Seq}(\mathcal{O} \times \mathcal{T})$, and $\mathsf{g}_\mathsf{s} : \mathcal{P} \rightharpoonup \mathcal{T}$ whose domains and co-domains are disjoint: $\mathcal{D}(z) = \mathsf{g}_\mathsf{l}(z)$ if $z \in \mathcal{L}$, $\mathsf{g}_\mathsf{m}(z)$ if $z \in \mathcal{L} \times \mathcal{T}$, and $\mathsf{g}_\mathsf{s}(z)$ if $z \in \mathcal{P}$.

Function g_l maps a location to the set of processes running in the service at that location. Given l, we read $\mathcal{D}(\mathsf{l}) = \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$ as "the processes $\mathsf{p}_1, \ldots, \mathsf{p}_n$ are running at the location l" (we assume processes to run at most at one location). Function g_m maps a couple location-tree to a message queue. This reflects message correlation as informally described above, where a queue resides in a service, i.e., at its location, and is pointed by a correlation key. Given a couple $\mathsf{l} : \mathsf{t}$, we read $\mathcal{D}(\mathsf{l} : \mathsf{t}) = \tilde{\mathsf{m}}$ as "the queue $\tilde{\mathsf{m}}$ resides in a service at location l and is pointed by correlation key t". The queue $\tilde{\mathsf{m}}$ is a sequence of messages $\tilde{\mathsf{m}} ::= \mathsf{m}_1 :: \cdots :: \mathsf{m}_n \mid \varepsilon$ and a message of the queue is $\mathsf{m} ::= (\mathsf{o}, \mathsf{t})$, where t is the payload of the message and o is the operation on which the message was received. Function g_s maps a process to its local state. Given a process p, the notation $\mathcal{D}(\mathsf{p}) = \mathsf{t}$ means that p has local state t.

Deployment Effects in BC. In BC, we replace FC deployment effects (i.e., the rules for $\mathcal{D} \xrightarrow{\delta} \mathcal{D}'$) with the ones reported in Fig. 3, commented below.

$$\frac{\mathsf{p} \in \mathcal{D}(\mathsf{l}) \qquad \mathcal{D} \xrightarrow{\mathbf{sup}(\ \mathsf{l}.\mathsf{p}[\mathsf{A}], \widehat{\mathsf{l}.\mathsf{q}.[\mathsf{B}]}\ )} \mathcal{D}'}{\mathcal{D} \xrightarrow{\text{start k: p[A]} \iff \widehat{\mathsf{l}.\mathsf{q}[\mathsf{B}]}} \mathcal{D}'} \ \lfloor^{\mathcal{D}}{}_{\mathsf{Start}}\rceil$$

$$\frac{\begin{array}{c}\mathsf{q}_1 \in \mathcal{D} \ (1) \qquad \mathsf{j} \in \mathrm{I} \setminus \{\mathsf{i}\} \qquad \underline{\mathsf{B}_\mathsf{i}.\mathsf{l}}(\mathsf{t}) = \mathsf{l}_\mathsf{i} \ (2) \qquad \underline{\mathsf{B}_\mathsf{i}.\mathsf{B}_\mathsf{j}}(\mathsf{t}) = \mathsf{t}_{\mathsf{ij}} \ (3) \qquad \mathsf{l}_\mathsf{j} : \mathsf{t}_{\mathsf{ij}} \notin \mathcal{D} \ (4) \\ \mathcal{D}_1 = \mathcal{D}\big[\ \mathsf{l}_\mathsf{i} \mapsto \mathcal{D}(\mathsf{l}_\mathsf{i}) \cup \{\mathsf{q}_\mathsf{i}\}\ \big] \ (5) \quad \mathcal{D}_2 = \mathcal{D}_1\big[\ \mathsf{l}_\mathsf{i} : \mathsf{t}_{\mathsf{ij}} \mapsto \varepsilon\ \big] \ (6)\end{array}}{\mathcal{D} \xrightarrow{\mathbf{sup}(\ \widehat{\mathsf{l}_\mathsf{i}.\mathsf{q}_\mathsf{i}[\mathsf{B}_\mathsf{i}]}\ \mathsf{i} \in \mathrm{I}\ )} \mathcal{D}_2\big[\mathsf{q}_1 \mapsto \mathcal{D}_2(\mathsf{q}_1) \triangleleft (\underline{\mathsf{k}}, \mathsf{t})\ \big] (7) \big[\ \mathsf{q}_\mathsf{h} \mapsto \{\underline{\mathsf{k}} : \mathsf{t}\} \mid \mathsf{h} \in \mathrm{I} \setminus \{\mathsf{q}_1\}\ \big] (8)} \ \lfloor^{\mathcal{D}}{}_{\mathsf{Sup}}\rceil$$

$$\frac{\mathsf{l} = \underline{\mathsf{k}.\mathsf{B}.\mathsf{l}}(\ \mathcal{D}(\mathsf{p})\ ) \qquad \mathsf{t}_\mathsf{c} = \underline{\mathsf{k}.\mathsf{A}.\mathsf{B}}(\ \mathcal{D}(\mathsf{p})\ ) \quad \mathsf{e}{\downarrow_{\mathcal{D}(\mathsf{p})}}\ \mathsf{t}_\mathsf{m}}{\mathcal{D} \xrightarrow{\mathsf{k}\ \mathsf{p}[\mathsf{A}].\mathsf{e} \longrightarrow \mathsf{B}.\mathsf{o}} \mathcal{D}\ [\ \mathsf{l} : \mathsf{t}_\mathsf{c} \mapsto \mathcal{D}(\mathsf{l} : \mathsf{t}_\mathsf{c}) :: (\mathsf{o}, \mathsf{t}_\mathsf{m})\ ]} \ \lfloor^{\mathcal{D}}{}_{\mathsf{Send}}\rceil$$

$$\frac{\mathsf{t}_\mathsf{c} = \underline{\mathsf{k}.\mathsf{A}.\mathsf{B}}(\ \mathcal{D}(\mathsf{q})\ ) \qquad \mathsf{q} \in \mathcal{D}(\mathsf{l}) \qquad \mathcal{D}(\mathsf{l} : \mathsf{t}_\mathsf{c}) = (\mathsf{o}, \mathsf{t}_\mathsf{m}) :: \tilde{\mathsf{m}} \qquad \mathcal{D}' = \mathcal{D}[\ \mathsf{l} : \mathsf{t}_\mathsf{c} \mapsto \tilde{\mathsf{m}}\ ]}{\mathcal{D} \xrightarrow{\mathsf{k}\ \mathsf{A} \longrightarrow \mathsf{q}[\mathsf{B}].\mathsf{o}(\mathsf{x})} \mathcal{D}'\ [\ \mathsf{q} \mapsto \mathcal{D}'(\mathsf{q}) \triangleleft (\underline{\mathsf{x}}, \mathsf{t}_\mathsf{m})\ )\ ]} \ \lfloor^{\mathcal{D}}{}_{\mathsf{Recv}}\rceil$$

Fig. 3. Deployment effects for Backend Choreographies.

Rule $\lfloor^{\mathcal{D}}{}_{\mathsf{Start}}\rceil$ simply retrieves the location of process p (the one that requested the creation of session k) and uses Rule $\lfloor^{\mathcal{D}}{}_{\mathsf{Sup}}\rceil$ to obtain the new deployment $\mathcal{D}'$ that supports interactions over session k. Namely, $\mathcal{D}'$ is an updated version of $\mathcal{D}$ with: *i*) the newly created processes for session k and *ii*) the queues used by the new processes and p to communicate over session k. In addition, in $\mathcal{D}'$, *iii*) the

new processes and $\mathtt{p}$ contain in their states a structure, rooted in $\underline{\mathtt{k}}$ and called *session descriptor*, that includes all the information (correlation keys and the locations of all involved processes) to support correlation-based communication in session $\mathtt{k}$. Formally, this is done by Rule $[\mathcal{D}|_{\mathsf{Sup}}]$ where we ① retrieve the starter process, here called $\mathtt{q}_1$, which is the only process already present in $\mathcal{D}$. Then, given a tree $\mathtt{t}$, we ensure it is a proper session descriptor for session $\mathtt{k}$, i.e., that:

② $\mathtt{t}$ contains the location $\mathtt{l}_i$ of each process. The location is stored under path $\underline{\mathtt{B}_i.\mathtt{l}}$, where $\mathtt{B}_i$ is the role played by the i-th process in the session;
③ $\mathtt{t}$ contains a correlation key $\mathtt{t}_{ij}$ for each ordered couple of roles $\mathtt{B}_i$, $\mathtt{B}_j$ under path $\underline{\mathtt{B}_i.\mathtt{B}_j}$, such that ④ there is no queue in $\mathcal{D}$ at location $\mathtt{l}_j$ pointed by correlation key $\mathtt{t}_{ij}$;

Finally, we assemble the update of $\mathcal{D}$ in four steps:

⑤ we obtain $\mathcal{D}_1$ by adding in $\mathcal{D}$ the processes $\mathtt{q}_2, \ldots, \mathtt{q}_n$ at their respective locations;
⑥ we obtain $\mathcal{D}_2$ by adding to $\mathcal{D}_1$ an empty queue ε for each couple $\mathtt{l}_j : \mathtt{t}_{ij}$;
⑦ we update $\mathcal{D}_2$ to store in the state of (the starter) process $\mathtt{q}_1$ the session support $\mathtt{t}$ under path $\underline{\mathtt{k}}$;
⑧ we further update $\mathcal{D}_2$ such that each new created process $(\mathtt{q}_2, \ldots, \mathtt{q}_n)$ has in its state just the session descriptor $\mathtt{t}$ rooted under path $\underline{\mathtt{k}}$.

We deliberately define in $[\mathcal{D}|_{\mathsf{Sup}}]$ the session descriptor $\mathtt{t}$ with a set of constrains on data, rather than with a procedure to obtain the data for correlation. In this way, our model is general enough to capture different methodologies for creating correlation keys (e.g., UUIDs or API keys).

Rule $[\mathcal{D}|_{\mathsf{Send}}]$ models the sending of a message. From left to right of the premises: we retrieve the location $\mathtt{l}$ of the receiver $\mathtt{B}$ from the state of the sender $\mathtt{p}$; we retrieve the correlation key $\mathtt{t}_c$ in the state of $\mathtt{p}$ (playing role $\mathtt{A}$) to send messages to role $\mathtt{B}$; we compute the payload of the message by evaluating the expression e against the local state of the sender $\mathtt{p}$. Then we obtain the updated deployment by adding message $(\mathtt{o}, \mathtt{t}_m)$ in the queue pointed by $\mathtt{l} : \mathtt{t}_c$ that we found via correlation.

Rule $[\mathcal{D}|_{\mathsf{Recv}}]$ models a reception. From left to right of the premises: we find the correlation key $\mathtt{t}_c$ for the queue that $\mathtt{q}$ (playing role $\mathtt{B}$) uses to receive from $\mathtt{A}$ in session $\mathtt{k}$; we retrieve the location $\mathtt{l}$ of $\mathtt{q}$; we access the queue pointed by $\mathtt{l} : \mathtt{t}_c$ and retrieve message $(\mathtt{o}, \mathtt{t}_m)$; we obtain a partial update of $\mathcal{D}$ in $\mathcal{D}'$ removing $(\mathtt{o}, \mathtt{t}_m)$ from the interested queue; we obtain the updated deployment by storing the payload $\mathtt{t}_m$ in the state of $\mathtt{q}$ under path $\underline{\mathtt{x}}$.

Encoding FC to BC. Runtime terms D, C in FC can be translated to BC simply by encoding D to an appropriate Backend deployment, since the syntax of choreographies is the same. For this translation, we need to know the roles played by processes and their locations, which is not recorded in D. We extract this information from the typing of C.

Definition 3 (Encoding FC in BC). *Let* D, C *be well-typed:* $\Gamma \vdash \mathsf{D}, \mathsf{C}$. *Then, its Backend encoding is defined as* $\langle\!\langle \mathsf{D} \rangle\!\rangle^{\Gamma}, \mathsf{C}$, *where* $\langle\!\langle \mathsf{D} \rangle\!\rangle^{\Gamma}$ *is given by the algorithm in Listing 1.1.*

```
1  ⟪D⟫^Γ = 𝒟 := ∅;
2   foreach p@l in Γ. 𝒟 := 𝒟[ l ↦ 𝒟(l) ∪ {p} ]; 𝒟 := 𝒟[ p ↦ ∅ ]
3   foreach p.x: U in Γ. 𝒟 := 𝒟[ p ↦ 𝒟(p) ◁ ( x, D(p)(x) ) ];
4   foreach { p: k[A] q: k[B], q@l} in Γ.  t := fresh(𝒟, l);
5    𝒟 := 𝒟[ l: t ↦ D(k[A⟩B]) ];
6    𝒟 := 𝒟[ p ↦ 𝒟(p) ◁ ( k.A.B, t ) ]; 𝒟 := 𝒟[ p ↦ 𝒟(p) ◁ ( k.B.l, l ) ];
7    𝒟 := 𝒟[ q ↦ 𝒟(q) ◁ ( k.A.B, t ) ]; 𝒟 := 𝒟[ q ↦ 𝒟(q) ◁ ( k.B.l, l ) ]
8  return 𝒟
```

Listing 1.1. Encoding Frontend to Backend Deployments.

Commenting the algorithm of $\langle\!\langle \mathsf{D} \rangle\!\rangle^{\Gamma}$, at line 2 it includes in $\mathcal{D}$ all (located) processes present in D (and typed in Γ) and instantiate their empty states. At line 3 it translates the state (i.e., the association *Var*iable-*Val*ue) of each process in D to its correspondent tree-shaped state in $\mathcal{D}$. At lines 4–7, for each ongoing session in D (namely, for each couple of processes, each playing a role in a given session), it sets the proper correlation keys and queues in $\mathcal{D}$ and, for each queue, it imports and translates the sequence of related messages.

The encoding from FC to BC guarantees a strong operational correspondence.

Theorem 2 (Operational Correspondence (FC $\leftrightarrows$ BC)). *Let* $\Gamma \vdash \mathsf{D}, \mathsf{C}$. *Then:*

1. *(Completeness)* $\mathsf{D}, \mathsf{C} \to \mathsf{D}', \mathsf{C}'$ *implies* $\langle\!\langle \mathsf{D} \rangle\!\rangle^{\Gamma}, \mathsf{C} \to \langle\!\langle \mathsf{D}' \rangle\!\rangle^{\Gamma'}, \mathsf{C}'$ *for some* Γ' *s.t.* $\Gamma' \vdash \mathsf{D}', \mathsf{C}'$*;*
2. *(Soundness)* $\langle\!\langle \mathsf{D} \rangle\!\rangle^{\Gamma}, \mathsf{C} \to \mathcal{D}, \mathsf{C}'$ *implies* $\mathsf{D}, \mathsf{C} \to \mathsf{D}', \mathsf{C}'$ *and* $\mathcal{D} = \langle\!\langle \mathsf{D}' \rangle\!\rangle^{\Gamma'}$ *for some* Γ' *s.t.* $\Gamma' \vdash \mathsf{D}', \mathsf{C}'$.

4 Dynamic Correlation Calculus

We now introduce the Dynamic Correlation Calculus (DCC), the target language of our compilation. To define DCC, we considered a previous formal model for Service-Oriented Computing, based on correlation [27]. However, we found the calculus in [27] too simple for our purposes: there, each process has only one message queue, while here we need to manage many queues per process (as in our Backend deployments). Hence, to define DCC, we basically extend the calculus in [27] to let processes create and receive from multiple queues. Beside the requirement of this work, many languages for SOC (e.g., BPEL [21]) let processes create and receive from multiple queues, which makes DCC a useful reference calculus in general.

Syntax. The syntax of DCC, reported in Fig. 4, comprises two layers: *Services*, ranged over by S, and *Processes*, ranged over by P.

In the syntax of services, term (*srv*) is a service, located at l, with a *Start Behaviour* $\mathsf{B_s}$ and running processes P (both described later on) and a queue map M. The queue map is a partial function $\mathsf{M} : \mathcal{T} \rightharpoonup \mathit{Seq}(\mathcal{O} \times \mathcal{T})$ that, similarly to function $\mathsf{g_m}$ in BC deployments, associates a correlation key t to a message queue. We model messages as in BC: a message is a couple (o, t) where o is

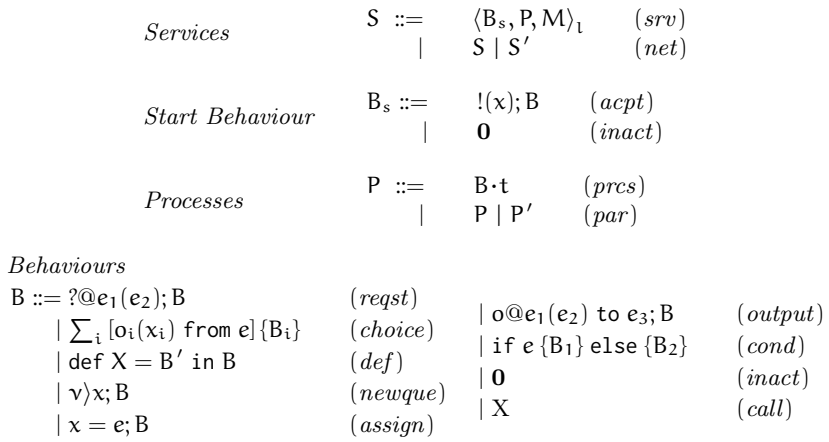

Fig. 4. Dynamic Correlation Calculus, syntax.

the operation on which the message has been received and $\mathtt{t}$ the payload of the message. Services are composed in parallel in term (*net*).

Concerning behaviours, in DCC we distinguish between start behaviours $\mathtt{B_s}$ and process behaviours B. Process behaviours define the general behaviour of processes in DCC, as described later on. Start behaviours use term $!(\mathtt{x})$ to indicate the availability of a service to generate new local processes on request. At runtime, the start behaviour $\mathtt{B_s}$ of a service is activated by the reception of a dedicated message that triggers the creation of a new process. The new process has (process) behaviour B, which is defined in $\mathtt{B_s}$ after the $!(\mathtt{x})$ term, and an empty state. The content of the request message is stored in the state of the newly created process, under the bound path $\mathtt{x}$. As in BC, also in DCC paths are used to access process states.

Operations ($\mathtt{o}$), procedures ($\mathtt{X}$), variables ($\mathtt{x}$, which are paths), and expressions ($\mathtt{e}$, evaluated at runtime on the state of the enclosing process) are as in BC. Terms (*choice*) and (*output*) model communications. In a (*choice*), when a message can be received from one of the operations $\mathtt{o_i}$ `from` the queue correlating with $\mathtt{e}$, the process stores under $\mathtt{x_i}$ the received message, it discards all other inputs, and executes the continuation $\mathtt{B_i}$. When only one input is available in a (*choice*), we use the contracted form $\mathtt{o(x)\ from\ e; B}$. Term (*output*) sends a message on operation $\mathtt{o}$, with content $\mathtt{e_2}$, while $\mathtt{e_1}$ defines the location of the service where the addressee (process) is running and $\mathtt{e_3}$ is the key that correlates with the receiving queue of the addressee. Term (*reqst*) is the dual of (*acpt*) and asks the service located at $\mathtt{e_1}$ to spawn a new process, passing to it the message in $\mathtt{e_2}$. Term (*cqueue*) models the creation of a new queue that correlates with the key contained in variable $\mathtt{x}$. Other terms are standard.

Semantics. In Fig. 5, we report a selection of the rules defining the semantics of DCC, a relation $\rightarrow$ closed under a (standard) structural congruence $\equiv$ that

$$\frac{e{\downarrow_t}\ t_c \quad j \in I \quad M(t_c) = (o_j, t_m) :: \tilde{m} \quad P = B_j \cdot t \triangleleft (x_j, t_m)}{\langle \_, \&_{i\in I}\,([o_i(x_i)\ \mathtt{from}\ e]\{B_i\}) \cdot t \mid \_, M\rangle_\_ \rightarrow \langle \_, P \mid \_, M[t_c \mapsto \tilde{m}]\rangle_\_}\ \lfloor^{DCC}{}_{|\mathsf{Recv}}\rceil$$

$$\frac{t_c \notin \mathbf{dom}(M)}{\langle \_, \nu\rangle x; B \cdot t \mid \_, M\rangle_\_ \rightarrow \langle \_, B \cdot t \triangleleft (x, t_c) \mid \_, M[t_c \mapsto \varepsilon]\rangle_\_}\ \lfloor^{DCC}{}_{|\mathsf{Newque}}\rceil$$

$$\frac{P = o@e_1(e_2)\ \mathtt{to}\ e_3; B \cdot t \quad e_1, e_2, e_3 \downarrow_t l, t_m, t_c \quad t_c \in \mathbf{dom}(M)}{\langle \_, P \mid \_, \_\rangle_\_ \mid \langle \_, \_, M\rangle_l \ \rightarrow\ \langle \_, B \cdot t \mid \_, M\rangle_\_ \mid \langle \_, \_, M[t_c \mapsto M(t_c) :: (o, t_m)]\rangle_l}\ \lfloor^{DCC}{}_{|\mathsf{Send}}\rceil$$

$$\frac{Q = B_2 \cdot \varnothing \triangleleft (x, t') \quad e_1 \downarrow_t l \quad e_2 \downarrow_t t'}{\langle \_,\ ?@e_1(e_2); B_1 \cdot t \mid \_, \_\rangle_\_ \mid \langle !(x); B_2, \_, \_\rangle_l \rightarrow \langle \_, B_1 \cdot t \mid \_, \_\rangle_\_ \mid \langle !(x); B_2, Q \mid \_,\ \_\rangle_l}\ \lfloor^{DCC}{}_{|\mathsf{Start}}\rceil$$

Fig. 5. Dynamic Correlation Calculus, semantics (selected).

supports commutativity and associativity of parallel composition. To enhance readability, in rules we omit irrelevant elements with the place-holder $\_$. We comment the rules. Rule $\lfloor^{DCC}{}_{|\mathsf{Recv}}\rceil$ models message reception: if the queue correlating with t_c (obtained from the evaluation of expression e against the state of the receiving process) has a message on operation o_j, we remove the message from the queue and assign the payload to the variable x_j in the state of the process. Rule $\lfloor^{DCC}{}_{|\mathsf{Newque}}\rceil$ adds to M an empty queue (ε) correlating with a key stored in x. As for BC in rule $\lfloor^{\mathcal{P}}{}_{|\mathsf{Sup}}\rceil$, we do not impose a structure for correlation keys, yet we require that they are distinct within their service. Rule $\lfloor^{DCC}{}_{|\mathsf{Send}}\rceil$ models message delivery between processes in different services: the rule adds the message from the sender at the end of the correlating queue of the receiver. Rule $\lfloor^{DCC}{}_{|\mathsf{Start}}\rceil$ accepts the creation of a new processes in a service upon request from an external process. The spawned process has B_2 as its behaviour and an empty state, except for x that stores the payload of the request.

5 Compiler from FC to DCC and Properties

We now present our main result: the correct compilation of FC programs into networks of DCC services. Given a term D, C in FC, our compilation consists of three steps: 1) it projects C into a parallel composition of *endpoint choreographies*, each describing the behaviour of a single process or service in C; 2) it encodes D to a Backend deployment; 3) it compiles the Backend choreography, result of the two previous steps, into DCC programs.

Step 1: Endpoint Projection (EPP). Given a choreography C, its EPP, denoted $[\![C]\!]$, returns an operationally-equivalent composition of endpoint choreographies. Intuitively, an endpoint choreography is a choreography that does not contain complete actions—terms (*start*) and (*com*) in FC—describing the behaviour of a process, which can be either a service process or an active one. Our definition of EPP is a straightforward adaptation of that presented in [23], so we omit it here (see [22] for the full definition). First, we define a *process projection*

to derive the endpoint choreography of a single process p from a choreography C, written $[\![\mathsf{C}]\!]_{\mathsf{p}}$. Then, we formalise the EPP of a choreography as the parallel composition of *i*) the projections of all active processes and *ii*) the merging of all service processes accepting requests at the same location. In the definition below, we use two standard auxiliary operators: the grouping operator $\lfloor \mathsf{C} \rfloor_{\mathsf{l}}$ returns the set of all service processes accepting requests at location l, and the merging operator $\mathsf{C} \sqcup \mathsf{C}'$ returns the service process whose behaviour is the merge of the behaviours of all the service processes accepting requests at the same location.

Definition 4 (Endpoint Projection). *Let* C *be a choreography. The endpoint projection of* C*, denoted* $[\![\mathsf{C}]\!]$*, is:*

$$[\![\mathsf{C}]\!] = \prod_{\mathsf{p} \,\in\, \mathbf{fp}(\mathsf{C})} [\![\mathsf{C}]\!]_{\mathsf{p}} \mid \prod_{\mathsf{l}} \left(\bigsqcup_{\mathsf{p} \,\in\, \lfloor \mathsf{C} \rfloor_{\mathsf{l}}} [\![\mathsf{C}]\!]_{\mathsf{p}} \right)$$

Example 2. As an example, let C be lines 5–8 of Example 1. Its EPP $[\![\mathsf{C}]\!]$ is the parallel composition of the endpoint choreographies of processes c, s, and b, let them be respectively $[\![\mathsf{C}]\!]_{\mathsf{c}}$, $[\![\mathsf{C}]\!]_{\mathsf{s}}$, and $[\![\mathsf{C}]\!]_{\mathsf{b}}$, then $[\![\mathsf{C}]\!] = [\![\mathsf{C}]\!]_{\mathsf{c}} \mid [\![\mathsf{C}]\!]_{\mathsf{s}} \mid [\![\mathsf{C}]\!]_{\mathsf{b}}$

$$[\![\mathsf{C}]\!]_{\mathsf{b}} = \mathtt{if}\ \mathsf{b}.\mathtt{closeTx}(\mathsf{cc},\ \mathsf{order})\ \{\ \mathsf{k}{:}\mathsf{b}[\mathsf{B}] \longrightarrow \mathsf{C}.\mathsf{ok}; \mathsf{k}{:}\mathsf{b}[\mathsf{B}] \longrightarrow \mathsf{S}.\mathsf{ok}\ \}\ \mathtt{else}\ \{\ \mathsf{k}{:}\mathsf{b}[\mathsf{B}] \longrightarrow \mathsf{C}.\mathsf{ko}; \mathsf{k}{:}\mathsf{b}[\mathsf{B}] \longrightarrow \mathsf{S}.\mathsf{ko}\ \}$$

$$[\![\mathsf{C}]\!]_{\mathsf{c}} = \mathsf{k}{:}\mathsf{B} \longrightarrow \mathsf{c}[\mathsf{C}].\{\ \mathsf{ok}(),\ \mathsf{ko}()\ \}$$

$$[\![\mathsf{C}]\!]_{\mathsf{s}} = \mathsf{k}{:}\mathsf{B} \longrightarrow \mathsf{s}[\mathsf{S}].\{\ \mathsf{ok}(),\ \mathsf{ko}()\ \}$$

As shown above, the projection of the conditional is homomorphic on the process (b) that evaluates it. The projection of (*com*) terms results into a partial (*send*) for the sender — as in the two branches of the conditional in $[\![\mathsf{C}]\!]_{\mathsf{b}}$ — and a partial (*recv*) for the receiver — as in $[\![\mathsf{C}]\!]_{\mathsf{c}}$ and $[\![\mathsf{C}]\!]_{\mathsf{s}}$. Note that the EPP merges branching behaviours: in $[\![\mathsf{C}]\!]_{\mathsf{c}}$ and $[\![\mathsf{C}]\!]_{\mathsf{s}}$ the two complete communications are merged into a partial reception on either operation ok or ko.

Below, $\mathsf{C} \prec \mathsf{C}'$ is the standard *pruning relation* [14], a strong typed bisimilarity such that C has some unused branches and always-available accepts. In FC, the EPP preserves well-typedness and behaviour:

Theorem 3 (EPP Theorem). *Let* D, C *be well-typed. Then,*

1. *(Well-typedness)* $\mathsf{D}, [\![\mathsf{C}]\!]$ *is well-typed.*
2. *(Completeness)* $\mathsf{D}, \mathsf{C} \to \mathsf{D}', \mathsf{C}'$ *implies* $\mathsf{D}, [\![\mathsf{C}]\!] \to \mathsf{D}', \mathsf{C}''$ *and* $[\![\mathsf{C}']\!] \prec \mathsf{C}''$.
3. *(Soundness)* $\mathsf{D}, [\![\mathsf{C}]\!] \to \mathsf{D}', \mathsf{C}'$ *implies* $\mathsf{D}, \mathsf{C} \to \mathsf{D}', \mathsf{C}''$ *and* $[\![\mathsf{C}'']\!] \prec \mathsf{C}'$.

Step 2: Encoding to BC. After the EPP, we use our deployment encoding to obtain an operationally-equivalent system in BC. From Theorems 2 and 3 we derive Corollary 1:

Corollary 1. *Let* $\Gamma \vdash \mathsf{D}, \mathsf{C}$*. Then:*

1. *(Completeness)* $\mathsf{D}, \mathsf{C} \to \mathsf{D}', \mathsf{C}'$ *implies* $\langle\!\langle \mathsf{D} \rangle\!\rangle^{\Gamma}, [\![\mathsf{C}]\!] \to \langle\!\langle \mathsf{D}' \rangle\!\rangle^{\Gamma'}, \mathsf{C}''$ *for some* Γ' *s.t.* $\Gamma' \vdash \mathsf{D}', \mathsf{C}''$ *and* $[\![\mathsf{C}']\!] \prec \mathsf{C}''$*;*
2. *(Soundness)* $\langle\!\langle \mathsf{D} \rangle\!\rangle^{\Gamma}, [\![\mathsf{C}]\!] \to \mathcal{D}, \mathsf{C}'$ *implies* $\mathsf{D}, \mathsf{C} \to \mathsf{D}', \mathsf{C}''$ *for some* Γ' *s.t.* $\Gamma' \vdash \mathsf{D}', \mathsf{C}''$*,* $\langle\!\langle \mathsf{D}' \rangle\!\rangle^{\Gamma'} = \mathcal{D}$ *and* $[\![\mathsf{C}'']\!] \prec \mathsf{C}'$*.*

Step 3: from BC to DCC. Given $\Gamma \vdash \mathsf{D}, \mathsf{C}$, where C is a composition of endpoint choreographies as returned by our EPP, we define a compilation $\boxed{\mathsf{D}, \mathsf{C}}^{\Gamma}$ into DCC by using the deployment encoding from FC to BC. To define $\boxed{\mathsf{D}, \mathsf{C}}^{\Gamma}$, we use:

- $\mathsf{C}|_{\mathsf{l}}$, to return the endpoint choreography for location l in C (e.g., $\mathsf{acc}\ \mathsf{k} : \mathsf{l}.\mathsf{p}[\mathsf{A}]; \mathsf{C}''$);
- $\mathsf{C}|_{\mathsf{p}}$, to return the endpoint choreography of process p in C;
- $\boxed{\mathsf{C}}^{\Gamma}$, to compile an endpoint choreography C to DCC, using the (selected) rules in Fig. 6;
- $\mathsf{l} \in \Gamma$, a predicate satisfied if, according to Γ, location l contains or can spawn processes;
- $\mathcal{D}|_{\mathsf{l}}$ returns the partial function of type $\mathcal{T} \rightharpoonup \mathit{Seq}(\mathcal{O} \times \mathcal{T})$ that corresponds to the projection of function g_{m} in $\mathcal{D}$ with location l fixed.

Definition 5 (Compilation). *Let* $\Gamma \vdash \mathsf{D}, \mathsf{C}$*, where* C *is a composition of endpoint choreographies, and* $\mathcal{D} = \langle\!\langle \mathsf{D} \rangle\!\rangle^{\Gamma}$*. The compilation* $\boxed{\mathsf{D}, \mathsf{C}}^{\Gamma}$ *is defined as*

$$\boxed{\mathsf{D}, \mathsf{C}}^{\Gamma} = \prod_{\mathsf{l} \in \Gamma} \left\langle \boxed{\mathsf{C}|_{\mathsf{l}}}^{\Gamma}, \prod_{\mathsf{p} \in \mathcal{D}(\mathsf{l})} \boxed{\mathsf{C}|_{\mathsf{p}}}^{\Gamma \cdot \mathcal{D}(\mathsf{p}),\, \mathcal{D}|_{\mathsf{l}}} \right\rangle_{\mathsf{l}}$$

Intuitively, for each service $\langle \mathsf{B}_s, \mathsf{P}, \mathsf{M} \rangle_{\mathsf{l}}$ in the compiled network: *i*) the start behaviour B_s is the compilation of the endpoint choreography in C accepting the creation of processes at location l; *ii*) P is the parallel composition of the compilation of all active processes located at l, equipped with their respective states according to $\mathcal{D} = \langle\!\langle \mathsf{D} \rangle\!\rangle^{\Gamma}$; *iii*) M is the set of queues in $\mathcal{D}$ corresponding to location l. We comment the rules in Fig. 6, where the notation $\odot$ is the sequence of behaviours $\odot_{i \in [1,n]}(\mathsf{B}_i) = \mathsf{B}_1; \ldots; \mathsf{B}_n$.

Requests. Function `start` defines the compilation of (*req*) terms: it compiles (*req*) terms to create the queues and a part of the session descriptor of a valid session support (mirroring rule $[^{\mathcal{P}}|_{\mathsf{sup}}]$) for the starter. Given a session identifier k, the located role of the starter ($\mathsf{l}_{\mathsf{A}}.\mathsf{A}$), and the other located roles in the session ($\overline{\mathsf{l}_{\mathsf{B}}.\mathsf{B}}$), function `start` returns DCC code that: (s_1) includes in the session descriptor all the locations of the processes involved in the session. In (s_2) it adds all the keys correlating with the queues of the starter for the session, it requests the creation of all the service processes for the session, and it waits for them to be ready using the reserved operation `sync`. Finally, (s_3) it sends to them the complete session descriptor obtained after the reception (in the `sync` step) of all correlation keys from all processes.

Accepts. Term (*acc*) defines the start behaviour of a spawned process at a location. Given a session identifier k, the role B of the service process, and the service typing $\mathsf{G}\langle \mathsf{A} | \tilde{\mathsf{C}} | \tilde{\mathsf{D}} \rangle$ of the location, function `accept` compiles the code that:

Let $\mathsf{p}@\mathsf{l}' \in \Gamma$, $\boxed{\mathsf{req\ k : p[A] \Longleftrightarrow \widetilde{l.B}; C}}^{\Gamma} = \mathbf{start}(\mathsf{k}, \mathsf{l}'.\mathsf{A}, \widetilde{\mathsf{l.B}}); \boxed{\mathsf{C}}^{\Gamma}$

$\mathbf{start}(\mathsf{k}, \mathsf{l}_{\mathsf{A}}.\mathsf{A}, \widetilde{\mathsf{l}_{\mathsf{B}}.\mathsf{B}}) =$

$$\underbrace{\bigodot_{\mathsf{I}\in\{\mathsf{A},\tilde{\mathsf{B}}\}} \underline{\mathsf{k.I.l}} = \mathsf{l}_{\mathsf{I}}\ ;}_{(s_1)\ \text{store locations}}\ \underbrace{\bigodot_{\mathsf{I}\in\{\tilde{\mathsf{B}}\}} \left(\begin{array}{l}\nu\rangle\underline{\mathsf{k.I.A}}\ ;?@\underline{\mathsf{k.I.l}}(\underline{\mathsf{k}})\ ;\\ \mathsf{sync}(\underline{\mathsf{k}})\ \mathsf{from}\ \underline{\mathsf{k.I.A}}\end{array}\right);}_{(s_2)\ \text{correlation keys and service processes}}\ \underbrace{\bigodot_{\mathsf{I}\in\{\tilde{\mathsf{B}}\}} \mathsf{start}@\underline{\mathsf{k.I.l}}(\underline{\mathsf{k}})\ \mathsf{to}\ \underline{\mathsf{k.A.I}}}_{(s_3)\ \text{handle session start}}$$

Let $\mathsf{l} \in \tilde{\mathsf{l}}, \tilde{\mathsf{l}}\colon \mathsf{G}\langle\mathsf{A}|\tilde{\mathsf{C}}|\tilde{\mathsf{D}}\rangle \in \Gamma$, $\boxed{\mathsf{acc\ k : l.q[B]; C}}^{\Gamma} = \mathbf{accept}(\ \mathsf{k},\ \mathsf{B}, \Gamma(\tilde{\mathsf{l}})\ ); \boxed{\mathsf{C}}^{\Gamma}$,

$\mathbf{accept}(\mathsf{k}, \mathsf{B}, \mathsf{G}\langle\mathsf{A}|\tilde{\mathsf{C}}|\tilde{\mathsf{D}}\rangle) =$

$$\underbrace{!(\underline{\mathsf{k}})\ ;}_{(a_1)}\ \underbrace{\bigodot_{\mathsf{I}\in\{\mathsf{A},\tilde{\mathsf{C}}\}\setminus\{\mathsf{B}\}} \left(\nu\rangle\underline{\mathsf{k.I.B}}\right);}_{(a_2)}\ \underbrace{\mathsf{sync}@\underline{\mathsf{k.A.l}}(\underline{\mathsf{k}})\ \mathsf{to}\ \underline{\mathsf{k.B.A}};}_{(a_3)}\ \underbrace{\mathsf{start}(\underline{\mathsf{k}})\ \mathsf{from}\ \underline{\mathsf{k.A.B}}}_{(a_4)}$$

Fig. 6. Compiler from Endpoint Choreographies to DCC.

(a_1) accepts the request to spawn a process, (a_2) creates its queues and keys, updates the session descriptor received from the starter, and sends it back to the latter (a_3). Finally with (a_4) the new process waits to start the session.

Example 3. We compile the first two lines of the choreography C in Example 1.

$$\boxed{\mathsf{D}, [\![\mathsf{C}]\!]}^{\Gamma} = \langle \mathbf{0}, \mathsf{P_c}\rangle_{\mathsf{l_c}}\ \mid\ \langle \mathsf{B_S}, \mathbf{0}\rangle_{\mathsf{l_s}}\ \mid\ \langle \mathsf{B_B}, \mathbf{0}\rangle_{\mathsf{l_B}}$$

where $\mathsf{P_c} = \begin{cases} \underline{\mathsf{k.S.l}} = \mathsf{l_s}; \underline{\mathsf{k.B.l}} = \mathsf{l_B}; \nu\rangle\underline{\mathsf{k.S.C}}; ?@\underline{\mathsf{k.S.l}}(\underline{\mathsf{k}}); \mathsf{sync}(\underline{\mathsf{k}})\ \mathsf{from}\ \underline{\mathsf{k.S.C}}; \nu\rangle\underline{\mathsf{k.B.C}}; ?@\underline{\mathsf{k.B.l}}(\underline{\mathsf{k}}); \\ \mathsf{sync}(\underline{\mathsf{k}})\ \mathsf{from}\ \underline{\mathsf{k.B.C}}; \mathsf{start}@\underline{\mathsf{k.S.l}}(\underline{\mathsf{k}})\ \mathsf{to}\ \underline{\mathsf{k.C.S}}; \mathsf{start}@\underline{\mathsf{k.B.l}}(\underline{\mathsf{k}})\ \mathsf{to}\ \underline{\mathsf{k.C.B}}; \\ \texttt{/* end of start-request */}\ \mathsf{buy}@\underline{\mathsf{k.S.l}}(\mathsf{product})\ \mathsf{to}\ \underline{\mathsf{k.C.S}}; \ldots \end{cases}$

and $\mathsf{B_S} = \begin{cases} !(\underline{\mathsf{k}}); \nu\rangle\underline{\mathsf{k.C.S}}; \nu\rangle\underline{\mathsf{k.B.S}}; \mathsf{sync}@\underline{\mathsf{k.C.l}}(\underline{\mathsf{k}})\ \mathsf{to}\ \underline{\mathsf{k.S.C}}; \mathsf{start}(\underline{\mathsf{k}})\ \mathsf{from}\ \underline{\mathsf{k.C.S}}; \\ \texttt{/* end of accept */}\ \mathsf{buy}(\mathsf{x})\ \mathsf{from}\ \underline{\mathsf{k.C.S}}; \ldots \end{cases}$

We omit to report $\mathsf{B_B}$, which is similar to $\mathsf{B_S}$.

Properties. We report the main properties of our compilation to DCC.

In our definition, we use the term *projectable* to indicate that, given a choreography C, we can obtain its projection $[\![\mathsf{C}]\!]$. Theorem 4 defines our result, for which, given a well-typed, projectable Frontend choreography, we can obtain its correct implementation as a DCC network.

Theorem 4 (Applied Choreographies).
Let D, C *be a Frontend choreography where* C *is projectable and* $\Gamma \vdash \mathsf{D}, \mathsf{C}$ *for some* Γ*. Then:*

1. *(Completeness)* $\mathsf{D}, \mathsf{C} \to \mathsf{D}', \mathsf{C}'$ *implies* $\boxed{\langle\!\langle \mathsf{D} \rangle\!\rangle^{\Gamma}, [\![\mathsf{C}]\!]}^{\Gamma} \to^{+} \boxed{\langle\!\langle \mathsf{D}' \rangle\!\rangle^{\Gamma'}, \mathsf{C}''}^{\Gamma'}$ *and* $[\![\mathsf{C}']\!] \prec \mathsf{C}''$ *and for some* Γ', $\Gamma' \vdash \mathsf{D}', \mathsf{C}'$.

2. *(Soundness)* $\boxed{\langle\!\langle \mathsf{D} \rangle\!\rangle^{\Gamma}, [\![\mathsf{C}]\!]}^{\Gamma} \rightarrow^* \mathsf{S}$ *implies* $\mathsf{D}, \mathsf{C} \rightarrow^* \mathsf{D}', \mathsf{C}'$ *and* $\mathsf{S} \rightarrow^* \boxed{\langle\!\langle \mathsf{D}' \rangle\!\rangle^{\Gamma'}, \mathsf{C}''}^{\Gamma'}$ *and* $[\![\mathsf{C}']\!] \prec \mathsf{C}''$ *and for some* $\Gamma',\ \Gamma' \vdash \mathsf{D}', \mathsf{C}'$.

By Theorem 1 and Theorem 4, deadlock-freedom is preserved from well-typed choreographies to their final translation in DCC. We say that a network S in DCC is deadlock-free if it is either a composition of services with terminated running processes or it can reduce.

Corollary 2. $\Gamma \vdash \mathsf{D}, \mathsf{C}$ *and* $\mathbf{co}(\Gamma)$ *imply that* $\boxed{\mathsf{D}, [\![\mathsf{C}]\!]}^{\Gamma}$ *is deadlock-free.*

6 Related Work and Discussion

Choreography Languages. This is the first correctness result of an end-to-end translation from choreographies to an abstract model based on a real-world communication mechanism. Previous formal choreography languages specify only an EPP procedure towards a calculus based on name synchronisation, leaving the design of its concrete support to implementors. Chor and AIOCJ [11,29] are the respective implementations of the models found in [2] and [8]. However, the implementation of their EPP significantly departs from their respective formalisation, since the former are based on message correlation. This gap breaks the correctness-by-construction guarantee of choreographies—there is no proof that the implementation correctly supports synchronisation on names. Implementations of other frameworks based on sessions share similar issues. For example, Scribble [7] is a protocol definition language based on multiparty asynchronous session types [13] used to statically [30] and dynamically [31,32] check compliance of interacting programs. Our work can be a useful reference to formalise the implementation of these session-based languages.

Delegation. Delegation supports the transferring of the responsibility to continue a session from a process to another [13] and it was introduced to choreographies in [2]. Introducing delegation in FC is straightforward, since we can just import the development from [2]. Implementing it in BC and DCC would be more involved, but not difficult: delegating a role in a session translates to moving the content of a queue from a process to another, and ensuring that future messages reach the new process. The mechanisms to achieve the latter part have been investigated in [30], which would be interesting to formalise in our framework.

Correlation keys. In the semantics of BC, we abstract from how correlation keys are generated. With this loose definition we capture several implementations, provided they satisfy the requirement of uniqueness of keys (wrt to locations). As future work, we plan to implement a language, based on our framework, able to support custom procedures for the generation of correlation keys (e.g., from database queries, cookies, etc.).

References

1. F. Montesi, *Choreographic Programming.* Ph.D. thesis, IT University of Copenhagen, 2013.

2. M. Carbone and F. Montesi, "Deadlock-freedom-by-design: multiparty asynchronous global programming," in *POPL*, pp. 263–274, ACM, 2013.
3. W3C WS-CDL Working Group, "WS-CDL version 1.0," 2004. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.
4. OMG, "Business Process Model and Notation." http://www.omg.org/spec/BPMN/2.0/.
5. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro, "Bridging the gap between interaction- and process-oriented choreographies," in *SEFM*, pp. 323–332, IEEE, 2008.
6. S. Basu, T. Bultan, and M. Ouederni, "Deciding choreography realizability," in *POPL*, pp. 191–202, ACM, 2012.
7. K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida, "Scribbling interactions with a formal foundation," in *ICDCIT*, pp. 55–75, Springer, 2011.
8. M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro, "Dynamic choreographies," in *COORDINATION*, pp. 67–82, Springer, 2015.
9. "Pi4soa," 2008. http://www.pi4soa.org.
10. JBoss Community, "Savara." http://www.jboss.org/savara/.
11. "Chor Programming Language." http://www.chor-lang.org/.
12. "AIOCJ framework." http://www.cs.unibo.it/projects/jolie/aiocj.html.
13. K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," *Journal of the ACM (JACM)*, vol. 63, no. 1, p. 9, 2016.
14. M. Carbone, K. Honda, and N. Yoshida, "Structured communication-centered programming for web services," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 2, pp. 8:1–8:78, 2012.
15. M. Carbone, F. Montesi, and C. Schürmann, "Choreographies, logically," *Distributed Computing*, vol. 31, no. 1, pp. 51–67, 2018.
16. M. Carbone, F. Montesi, C. Schürmann, and N. Yoshida, "Multiparty session types as coherence proofs," *Acta Inf.*, vol. 54, no. 3, pp. 243–269, 2017.
17. Z. Qiu, X. Zhao, C. Cai, and H. Yang, "Towards the theoretical foundation of choreography," in *WWW*, pp. 973–982, IEEE Computer Society Press, 2007.
18. R. Milner, *A Calculus of Communicating Systems*, vol. 92 of *LNCS*. Springer, 1980.
19. R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I and II," *Information and Computation*, vol. 100, pp. 1–40,41–77, Sept. 1992.
20. S. Giallorenzo, *Real-World Choreographies.* PhD thesis, University of Bologna, Italy, 2016.
21. OASIS, "WS-BPEL." http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, 2007.
22. M. Gabbrielli, S. Giallorenzo, and F. Montesi, "Applied choreographies," Technical Report, 2018. http://www.saveriogiallorenzo.com/publications/AC/AC_tr.pdf.
23. F. Montesi and N. Yoshida, "Compositional choreographies," in *CONCUR*, pp. 425–439, Springer, 2013.
24. D. Sangiorgi and D. Walker, *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.
25. B. C. Pierce, *Types and Programming Languages.* MA, USA: MIT Press, 2002.
26. M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani, "Global progress for dynamically interleaved multiparty sessions," *Mathematical Structures in Computer Science*, vol. 26, no. 2, pp. 238–302, 2016.
27. F. Montesi and M. Carbone, "Programming services with correlation sets," in *ICSOC*, pp. 125–141, Springer, 2011.
28. F. Montesi, C. Guidi, and G. Zavattaro, "Service-oriented programming with Jolie," in *Web Services Foundations*, pp. 81–107, 2014.

29. M. Dalla Preda, S. Giallorenzo, I. Lanese, J. Mauro, and M. Gabbrielli, "AIOCJ: A choreographic framework for safe adaptive distributed applications," in *SLE*, pp. 161–170, Springer, 2014.
30. R. Hu, N. Yoshida, and K. Honda, "Session-based distributed programming in java," in *ECOOP*, pp. 516–541, Springer, 2008.
31. R. Demangeon, K. Honda, R. Hu, R. Neykova, and N. Yoshida, "Practical interruptible conversations: distributed dynamic verification with multiparty session types and python," *Formal Methods in System Design*, vol. 46, no. 3, pp. 197–225, 2015.
32. R. Neykova and N. Yoshida, "Multiparty session actors," *Logical Methods in Computer Science*, vol. 13, no. 1, 2017.