From global choreographies to verifiable efficient distributed implementations

Mohamad Jaber, Yliès Falcone, Paul Attie, Al-Abbass Khalil, Rayan Hallal, Antoine El-Hokayem

From Global Choreographies to Verifiable Efficient Distributed Implementations

Mohamad Jaber[a], Yliès Falcone[b], Paul Attie[c], Al-Abbass Khalil[a], Rayan Hallal[a], Antoine El-Hokayem[b]

[a] *Computer Science Department, American University of Beirut, Beirut, Lebanon*
[b] *Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, Laboratoire d'Informatique de Grenoble, 38000 Grenoble, France*
[c] *School of Computer and Cyber Sciences, Augusta University, Augusta, Georgia, USA*

Abstract

We define a method to automatically synthesize efficient distributed implementations from high-level global choreographies. A global choreography describes the execution and communication logic between a set of provided processes which are described by their interfaces. At the choreography level, the operations include multiparty communications, choice, loop, and branching. A choreography is master triggered: it has one master to trigger its execution. This allows us to automatically generate conflict-free distributed implementations without controllers. The behavior of the synthesized implementations follows the behavior of choreographies. In addition, the absence of controllers ensures the efficiency of the implementation and reduces the communication needed at runtime. Moreover, we define a translation of the distributed implementations to equivalent `Promela` versions. The translation allows verifying the distributed system against behavioral properties. We implemented a Java prototype to validate the approach and applied it to automatically synthesize micro-service architectures. We also illustrate our method on the automatic synthesis of a verified distributed buying system.

1. Introduction

Developing correct distributed software is notoriously difficult. This is mainly due to their complex structure that consists of interactions between distributed processes. We mainly distinguish two possible directions to cope with the complexity of the interaction model: (1) high-level modeling frameworks [7]; and (2) session types [6, 22, 8, 37, 18, 11]. The former facilitates expressing the communication models but makes efficient code generation difficult. High-level

Email addresses: `mj54@aub.edu.lb` (Mohamad Jaber), `ylies.falcone@univ-grenoble-alpes.fr` (Yliès Falcone), `pattie@augusta.edu` (Paul Attie), `aak103@mail.aub.edu@aub.edu.lb` (Al-Abbass Khalil), `rah74@aub.edu.lb` (Rayan Hallal), `antoine.el-hokayem@univ-grenoble-alpes.fr` (Antoine El-Hokayem)

Preprint submitted to Journal of Logical and Algebraic Methods in Programming *June 25, 2020*

and expressive communication models require the generation of controllers to implement their communication logic. For instance, if we consider multiparty interactions with non-deterministic behavior that may introduce conflicts between processes, such conflicts would be resolved by creating new processes (controllers). Additionally, it is easier to develop distributed systems by reasoning about the global communication model and not local processes. For these reasons, session types were introduced. Session types feature the notions of (i) *global protocol* which describes the communication protocol between processes and (ii) *local types* which are the projections of the global protocol on processes. Session types are generally developed following the steps below:

1. design of the global protocol;
2. automatic synthesis of the local types;
3. development of the code of processes;
4. static type checking of the local code of the processes w.r.t. their local protocols.

As a result, the obtained distributed software follows the stipulated global protocol. However, the current approach to developing session types suffers from several limitations. First, there is redundancy in the code of local processes: even though the code skeleton of the local processes can be inferred from the local types, the programmer has to explicitly write the full code of the processes. Second, the communication logic is tangled as modifying the global protocol requires reimplementing some of the local code of the affected processes. Moreover, it suffers from the absence of facilities to handle and combine both communication and computation concerns.

Contributions. In this paper, we introduce a new framework which allows the automatic synthesis of the local code of the processes starting from a global choreography. First, inspired from the Behavior Interaction Priority framework (BIP) [5], we consider a set of components/processes with their interfaces and a configuration file that defines the variables of each component as well as the mapping between ports and their computation blocks. Then, given a global choreography, which is defined on the set of ports of the components and which models coordination and composition operators, we automatically synthesize the local code of the processes, which embeds all communication and control flow logic. The choreography allows us to define: (1) multiparty interaction; (2) branching; (3) loop; (4) sequential composition; and (5) parallel composition. Without loss of generality, as in most distributed system applications, we consider master-based protocols. In master-based protocols, each interaction has a master component deciding whether it can take place and what are the components involved in the interaction. This allows for the generation of fully distributed implementations, i.e., without the need of controllers, hence reducing the need for communication at runtime. Moreover, we discuss some correctness arguments about the behavior of the synthesized implementations following the semantics of choreographies. Furthermore, we define a translation of the distributed implementations to equivalent `Promela` versions. Such a translation allows us to verify user-defined properties on the implementations. We use the SPIN model-checker to verify properties. Our transformations are implemented in a Java tool that we applied to automatically synthesize micro-service architectures starting from global protocols.

Differences with HPC 4PAD paper. This paper revises and extends a paper that appeared in the proceedings of the International Symposium on Formal Approaches to Parallel and Distributed Systems (HPCS 4PAD 2018) [17]. The additional contributions can be summarized as follows. First, we defined a formal semantics for choreographies, using structured operational semantics rules.

Second, we defined a translation of the distributed implementations to equivalent `Promela` processes. This permits the verification of the implementations against (safety and liveness) behavioral properties and thus provides additional confidence in the behavior of the distributed implementation. Third, we added a synthesis example of a micro-service for a buying system, inspired from the examples tackled in collaboration with Murex Services S.A.L. industry [29]. Fourth, we revisited and extended the related work. Finally, we improved the presentation and readability by adding more details and examples.

Paper organization. The remainder of this paper is structured as follows. Section 2 fixes some notation used throughout the paper. Section 3 introduces some preliminary notions, common to choreography and distributed component-based systems. To illustrate our approach, we present a toy example of a variant of producer-consumer in Section 4. In Section 5, we define the syntax and the semantics of the choreography model. In Section 6, we present an illustrating example by modeling the two-phase commit protocol using our choreography model. In Section 7, we introduce a distributed component-based model that is used to define the semantics of our choreography model. In Section 8, we transform choreographies to distributed component-based systems and informally argue about its correctness. In Section 9, we provide an efficient code generation of the obtained distributed component-based model and present a real case study. In Section 10, we present one of the case studies on a micro-service architecture to automatically derive the skeleton of each micro-service, in collaboration with Murex Services S.A.L. industry [29]. In Section 11, we define a translation of the code generated from a choreography into `Promela` for the purpose of verifying the generated code. In Section 12, we present a case Study to synthesize an implementation of a buying system. We present related work in Section 13. We draw conclusions and outline future work in Section 14.

2. Notation

We denote by $\mathbb{N}$ the set of natural numbers with the usual total orders $\leq$ and $\geq$; $\mathbb{N}^+$ denotes the set $\mathbb{N} \setminus \{0\}$. Given two natural numbers a and b such that $a \leq b$, we denote by $[a, b]$, the interval between a and b, i.e., the set

$\{x \in \mathbb{N} \mid x \geq a \wedge x \leq b\}$. A sequence of elements over a set E of length $n \in \mathbb{N}$ is formally defined as a (total) function from $[1, n]$ to E. The empty sequence over E (function from $\emptyset$ to E) is denoted by ϵ_E (or ϵ when clear from the context). The length of a sequence s is denoted by $|s|$. The set of (finite) sequences over E is denoted by E^*. The (usual) concatenation of a sequence s to a sequence s' is the sequence denoted by $s \cdot s'$. Given two sets E and F, we denote by $[E \to F]$ the set of functions from E to F. Given some function $f \in [E \to F]$ and an element $e \in E$, we denote by $f(e)$ the element in F associated with e according to f.

3. Preliminary Notions

To later construct a system, we assume an architecture with n components $\{B_i\}_{i=1}^n$, with $n \in \mathbb{N}^+$. At this stage, components are just interfaces with ports for communication. To each port of a component is attached a (unique) variable. In this section, we define these notions common to choreographies and component-based systems, later defined in Section 5 and Section 7 respectively.

Types, variables, expressions, and functions. We use a set of data types, $\mathit{DataTypes}$, including the set of usual types found in programming languages $\{\texttt{int}, \texttt{str}, \texttt{bool}, \ldots\}$ and a set of (typed) variables Vars. Variables are partitioned over components, i.e., $\mathit{Vars} = \bigcup_{i=1}^n \mathit{Vars}_i$ and $\forall i, j \in [1, n] : i \neq j \implies \mathit{Vars}_i \cap \mathit{Vars}_j = \emptyset$. Variables take values in a general data domain Data containing all values associated with the types in $\mathit{DataTypes}$ plus a neutral communication element denoted by $\bot_{\mathrm{d}}$. We call any function with codomain Data a valuation. Moreover, for two valuations v and v', v'/v denotes the valuation where values in v' have priority over those in v. For a set of variables $X \subseteq \mathit{Vars}$, we denote by $\mathcal{G}(X)$ (resp. $\mathit{Expr}(X)$) the set of boolean (resp. all, i.e., boolean and arithmetic) expressions over X, constructed in the usual manner. Expressions can be used as function descriptions, and, for an expression $e \in \mathit{Expr}(X)$ and a valuation $v \in [X \to \mathit{Data}]$, we note $e(v)$ the value in Data of expression e according to v.

Types and ports. We define the notion of port type, and then of port.

Definition 1 (Port type). *The set of port types, denoted by PortTypes, is* $\{\texttt{ss}, \texttt{as}, \texttt{r}, \texttt{in}\}$*, where* $\texttt{ss}$ *(resp.* $\texttt{as}, \texttt{r}, \texttt{in}$*) denotes a synchronous send (resp. asynchronous send, receive, internal) communication type.*

Definition 2 (Port). *A synchronous send, asynchronous send or internal port is a tuple* $(p, x_p, \mathit{dtype}, \mathit{ctype})$ *where:* p *is the port identifier;* $x_p \in \mathit{Vars}$ *is the port variable;* $\mathit{dtype} \in \mathit{DataTypes}$ *is the port data type; and* $\mathit{ctype} \in \mathit{PortTypes}$ *is the port communication type. Similarly, a receive port is a tuple* $(p, x_p, \mathit{dtype}, \mathit{ctype}, \mathit{buff})$ *where* $\mathit{buff} \in \mathit{Data}^*$ *is the port buffer (used to store values).*

Ports are referred to by their identifier. In the rest of the paper, we use the dot notation:

- for a (a)synchronous send or internal port $(p, x_p, ptype, ctype)$ or a receive port $(p, x_p, ptype, ctype, buff)$, $p.\texttt{var}$ (resp. $p.\texttt{dtype}$, $p.\texttt{ctype}$, $p.\texttt{buff}$) refers to x_p (resp. *dtype*, *ctype*, *buff*);
- for a set of ports P, $P.\texttt{var}$ denotes $\{p.\texttt{var} \mid p \in P\}$, the set of variables of the ports in P.

Given a port p, we define the predicate $\texttt{isSSend}(p)$ (resp., `isASend`, `isRecv`, `isInternal`) that holds true iff (the communication type of) p is a synchronous send (resp., asynchronous send, receive, internal) port, i.e., iff $p.\texttt{ctype} = \texttt{ss}$ (resp. $\texttt{as}, \texttt{r}, \texttt{in}$).

To later construct a system, we assume a set of ports $\mathcal{P}$ and a partition of the ports over components: $\mathcal{P} = \cup_{i=1}^{n} P_i$. We define $\mathcal{P}^{\text{ss}} = \{p \in \mathcal{P} \mid \texttt{isSSend}(p)\}$ (resp. $\mathcal{P}^{\text{as}} = \{p \in \mathcal{P} \mid \texttt{isASend}(p)\}$, $\mathcal{P}^{\text{r}} = \{p \in \mathcal{P} \mid \texttt{isRecv}(p)\}$) to be the set of all synchronous send port (resp. asynchronous send ports, receive ports) of the system. Moreover, we denote by $\mathcal{P}_i^{\text{ss}}$ (resp. $\mathcal{P}_i^{\text{as}}$, $\mathcal{P}_i^{\text{r}}$) the set of all synchronous send (resp., asynchronous send, receive) ports of atomic component B_i.

Update functions. Update functions serve to abstract internal computations performed by atomic components.

Definition 3 (Update function). *An update function f over a set of variables $X \subseteq Vars$ is a sequence of assignments, where each assignment is of the form $x := expr_X$, where $x \in X$ and $expr_X \in Expr(X)$. The set of update functions over X is denoted by $\mathcal{F}(X)$.*

For an update function f and a valuation v, executing f on v yields a new valuation v', noted $v' = f(v)$, such that v' is obtained in the usual way by the successive applications of the assignments in f taken in order and where the right-hand side expressions are evaluated with the latest constructed temporary valuation.

4. Illustrating Example

To illustrate our approach, we consider a toy example of a variant of producer-consumer. The example begins by modeling producer-consumer using choreographies (described along with their semantics in Section 5). Then, we show the corresponding component-based distributed implementation (detailed in Section 7) which is synthesized from the choreographies using transformations described in Section 8.

Choreography. The system consists of two components: a producer (`P`) and a consumer (`C`). Initially, `P` has a certain number B of messages to send asynchronously through its interface s. The number of messages that remain to be sent is stored in variable n of port p. `P` sends its messages asynchronously through interface s and `C` receives messages through interface r. While `P` has messages to send ($n > 0$), it applies some computation function f on the message and decrements the value of n. After `P` has finished ($\bullet$) sending ($\rightarrow$), `C`

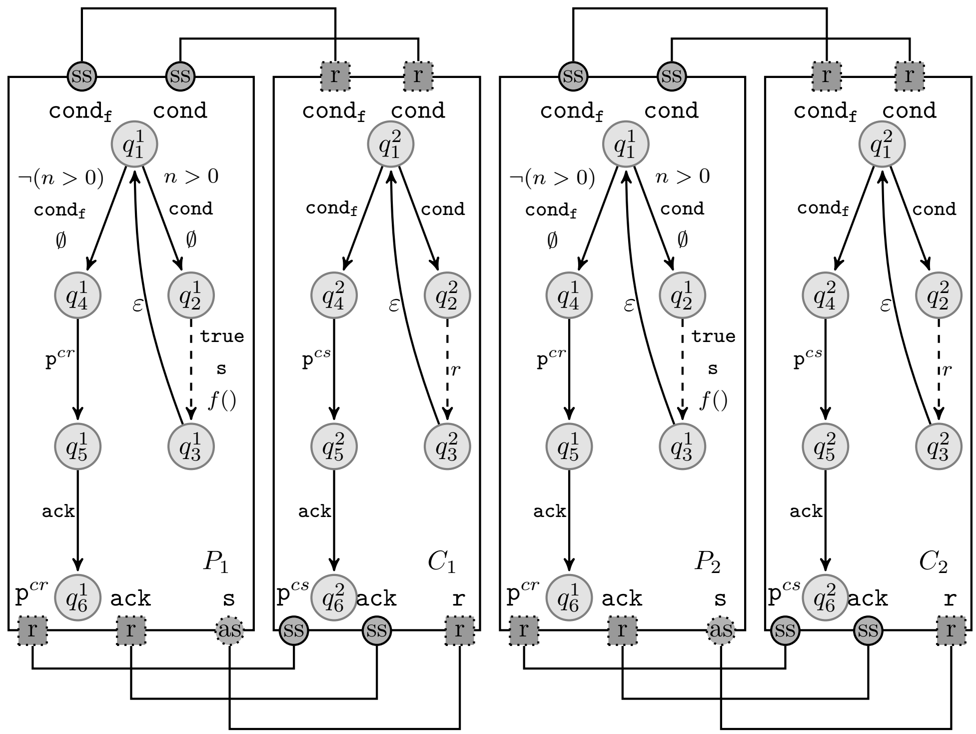

Figure 1: A toy example of a variant of producer-consumer.

sends an acknowledgment message to P. We consider two instances of producers (resp. consumers) $\mathtt{P}_1$ and $\mathtt{P}_2$ (resp. $\mathtt{C}_1$ and $\mathtt{C}_2$), where the two pairs are running in parallel. Below is the choreography modeling (in a simplified syntax) the above scenario and realizing the transmission of message from P to C.

$$(\mathtt{while}(\mathtt{P}_1.\mathtt{cond}[\mathtt{n} > 0])\{\mathtt{P}_1.\mathtt{s}[\mathtt{true}, \mathtt{f}()] \longrightarrow \{\mathtt{C}_1.\mathtt{r}[\emptyset]\}\} \bullet \mathtt{C}_1.\mathtt{ack} \longrightarrow \{\mathtt{P}_1.\mathtt{ack}\})$$
$$\| \ (\mathtt{while}(\mathtt{P}_2.\mathtt{cond}[\mathtt{n} > 0])\{\mathtt{P}_2.\mathtt{s}[\mathtt{true}, \mathtt{f}()] \longrightarrow \{\mathtt{C}_2.\mathtt{r}[\emptyset]\}\} \bullet \mathtt{C}_2.\mathtt{ack} \longrightarrow \{\mathtt{P}_2.\mathtt{ack}\})$$

Synthesized distributed system. The corresponding distributed component-based model is depicted in Figure 1. The system is composed of four components. Component $\mathtt{P}_1$ has three basic interfaces `ack` (for receive), `s` (asynchronous send) and `cond` (synchronous cond). Two other interfaces are generated for control: $\mathtt{cond_f}$ and $\mathtt{p}^{cr}$. Condition $\mathtt{cond_f}$ is enabled when the condition of the while does not hold. $\mathtt{p}^{cr}$ is used to implement the sequential primitive ($\bullet$). The two parallel choreographies are independent and correspond of the parallel execution of $\mathtt{P}_1$ with $\mathtt{C}_1$ and $\mathtt{P}_2$ with $\mathtt{C}_2$. As can be noticed, there is no need of controllers and one can use a process or thread for each component.

Promela model. From the above description of the distributed implementation, we can synthesize Promela processes (one per componenent). Interactions will be modeled as channels in Promela. See Listing 7 for an example.

`ch`	::=	`nil`	# *empty choreography*
	|	`snd` ⟶ {`rcv_list`} : $\langle t \rangle$	# *typed send / receive*
	|	$B \oplus$ {`cont_list`}	# *conditional master branching*
	|	`while(snd) ch end`	# *iterative composition*
	|	`ch` • `ch`	# *sequential composition*
	|	`ch` ‖ `ch`	# *parallel composition*
`snd`	::=	$psas[g, f]$	# *synchronous/asynchronous send ports*
			# *with guard & update function*
`rcv_list`	::=	$pr[f]$ | $pr[f]$, `rcv_list`	# *list of receive ports with update function*
`cont_list`	::=	`snd : ch` | `snd : ch, cont_list`	# *list of continuations*
t	$\in$	*DataTypes*	# *types*
B	$\in$	$\{B_1, \ldots, B_n\}$	# *available components*
$psas$	$\in$	$\mathcal{P}^{ss} \cup \mathcal{P}^{as}$	# *synchronous/asynchronous*
			# *send ports identifiers*
pr	$\in$	$\mathcal{P}^{r}$	# *receive ports*
g	$\in$	$\mathcal{G}(X)$	# *guards*
f	$\in$	$\mathcal{F}(X)$	# *update function*

Figure 2: Abstract grammar defining the syntax of the choreography model.

5. Global Choreography

In this section, we define the global choreography model. Recall that components are seen as interfaces and a choreography serves the purpose of coordinating the communications and computations of components. In choreographies, ports are used with guards and update functions.

We start by defining the syntax and then the semantics of choreographies.

Syntax of choreographies. We introduce the abstract syntax of the global choreography model.

Definition 4 (Abstract syntax of the choreography model). *The abstract grammar in Figure 2 defines the syntax of the choreography model. We denote by Chors the set of choreographies defined by this grammar.*

The definition of choreographies relies on the previously defined concepts such as update functions in $\mathcal{F}(X)$, guards in $\mathcal{G}(X)$, the existing types in *DataTypes*, available components in $\{B_1, \ldots, B_n\}$, and the various types of ports (synchronous and asynchronous send ports in $\mathcal{P}^{ss}$ and $\mathcal{P}^{as}$ and receive ports in $\mathcal{P}^{r}$). It also relies on the definitions of send port augmented with guard and update function and lists of receive ports and continuations. A send port augmented with guard and update function is of the form $psas[g, f]$ where $psas$ is a synchronous or asynchronous send port, g a guard, and f an update function. In a list of receive ports, each element is of the form $pr[g]$ where pr is a receive port identifier and g a guard. In a list of continuations, each element is of the form $psas{:}ch$ where $psas$ is a synchronous or asynchronous send port and ch is a choreography. We extend the dot notation to choreographies and, for a send or receive port augmented with guard and update function, i.e., of the form $psas[g, f]$ or $pr[g]$, we note $psas.\texttt{guard}$ and $pr.\texttt{guard}$ for g and $psas.\texttt{ufct}$ for f.

Base choreographies include the empty choreography (`nil`) and the send/receive communication primitive. Send/receive communications are of the form `snd` $\longrightarrow$ {`rcv_list`} : $\langle t \rangle$ where `snd` is a (synchronous or asynchronous) send port, `rcv_list` is a list of receive ports and : $\langle t \rangle$ is a type annotation with $t \in \mathit{DataTypes}$.

Composite choreographies include the conditional master branching, the iterative, sequential and parallel compositions. Conditional master branching are of the form $B \oplus$ {`cont_list`} where B is a component taking the branching decision and `cont_list` a list of continuations, that is, a list of choreographies guarded by send ports. The iterative composition of a choreography ch is of the form `while(snd)` ch `end` where `snd` defines a send port with a guard and an update function. The component of the send port guides the loop condition. Given two choreographies ch_1 and ch_2, the sequential (resp. parallel) composition of ch_1 and ch_2 is noted $ch_1 \bullet ch_2$ (resp. $ch_1 \parallel ch_2$).

Remark 1. *Guards are not attached to receive ports so as to always permit the reception of data. Such a choice also allows for generating more efficient code with less communication overhead, and, as communication are master triggered, it avoids deadlock situations.*

Typing constraints. Additionally, for a choreography to be well defined, it should respect the following typing constraints:

- In a synchronous/asynchronous send port with guard and update function $psas[g, f]$, the variables used in the guard g should belong to the component of port $psas$.
- In a conditional master branching, the send ports in the continuation list should belong to the component.

Semantics of choreographies. In the following, we consider well-typed choreographies built with the syntax in Definition 4. We define the (structural operational) semantics of choreographies. For this, we consider that states of a choreography are valuations of the component variables in $[X \to \mathit{Data}]$. Recall that variables and ports are partitioned over components. We denote by $\mathit{ChorState}$ the set of choreography states.

Before actually defining the semantics, we need to model the effect of communication on the choreography state. We model the sending through a port to a set of ports with a function send : $\mathit{ChorState} \times (\mathcal{P}^{\mathrm{as}} \cup \mathcal{P}^{\mathrm{s}}) \times 2^{\mathcal{P}^{\mathrm{r}}} \to \mathit{ChorState}$ that takes as input a choreography state and outputs a choreography state when a communication occurs from the (synchronous or asynchronous) send port of a component to the receive ports of some components: $\mathrm{send}(\sigma, \mathit{snd}, \{\mathit{rcv\_list}\})$ is state σ where the value of variable of port snd is used to update the variables attached to ports in $\{\mathit{rcv\_list}\}$. Formally: $\mathrm{send}(\sigma, \mathit{snd}, \{\mathit{rcv\_list}\}) =$

$$\frac{}{(\texttt{nil}, \sigma) \xRightarrow{\tau} \sigma}\ \texttt{(nil)}$$

$$\frac{snd \in \mathcal{P}^{\text{ss}} \quad \sigma \models g \quad rcv\_list = pr_1[f_1], \ldots, pr_k[f_k]}{(snd[g, f] \longrightarrow \{rcv\_list\}, \sigma) \xRightarrow{\{snd, pr_1, \ldots, pr_k\}} f \circ f_k \circ \cdots \circ f_1 \circ \text{send}(\sigma, snd, \{pr_1, \ldots, pr_k\})}\ \texttt{(synch-sendrcv)}$$

$$\frac{snd \in \mathcal{P}^{\text{as}} \quad \sigma \models g}{(snd[g, f] \longrightarrow \{rcv\_list\}, \sigma) \xRightarrow{\{snd\}} (\{rcv\_list\}, f \circ \text{send}(\sigma, snd, rcv\_list))}\ \texttt{(asynch-sendrcv-1)}$$

$$\frac{pr[f] \in \{rcv\_list\}}{(\{rcv\_list\}, \sigma) \xRightarrow{\{pr\}} (\{rcv\_list\} \setminus \{pr[f]\}, f(\sigma))}\ \texttt{(asynch-sendrcv-2)}$$

$$\frac{\sigma \models g_j}{(B \oplus \{snd_1[g_1, f_1] : ch_1, \ldots, snd_k[g_k, f_k] : ch_k\}, \sigma) \xRightarrow{\{snd_j\}} (ch_j, f_j(\sigma))}\ \texttt{(master-branching)}$$

$$\frac{\sigma \models g}{(\texttt{while}(snd[g, f])\ ch\ \texttt{end}, \sigma) \xRightarrow{\{snd\}} (ch \bullet \texttt{while}(snd[g, f])\ ch\ \texttt{end}, f(\sigma))}\ \texttt{(iterative-tt)}$$

$$\frac{\sigma \not\models g}{(\texttt{while}(snd[g, f])\ ch\ \texttt{end}, \sigma) \xRightarrow{\tau} \sigma}\ \texttt{(iterative-ff)}$$

$$\frac{(ch_1, \sigma) \xRightarrow{l_1} (ch_1', \sigma')}{(ch_1 \bullet ch_2, \sigma) \xRightarrow{l_1} (ch_1' \bullet ch_2, \sigma')}\ \texttt{(sequential-1)} \qquad \frac{(ch_1, \sigma) \xRightarrow{l_1} \sigma'}{(ch_1 \bullet ch_2, \sigma) \xRightarrow{l_1} (ch_2, \sigma')}\ \texttt{(sequential-2)}$$

$$\frac{(ch_1, \sigma_1) \xRightarrow{l_1} (ch_1', \sigma_1')}{(ch_1 \parallel ch_2, \sigma_1) \xRightarrow{l_1} (ch_1' \parallel ch_2, \sigma_1')}\ \texttt{(parallel-1)} \qquad \frac{(ch_2, \sigma_2) \xRightarrow{l_2} (ch_2', \sigma_2')}{(ch_1 \parallel ch_2, \sigma_2) \xRightarrow{l_2} (ch_1 \parallel ch_2', \sigma_2')}\ \texttt{(parallel-2)}$$

$$\frac{(ch_1, \sigma_1) \xRightarrow{l_1} \sigma_1'}{(ch_1 \parallel ch_2, \sigma_1) \xRightarrow{l_1} (ch_2, \sigma_1')}\ \texttt{(parallel-3)} \qquad \frac{(ch_2, \sigma_2) \xRightarrow{l_2} \sigma_2'}{(ch_1 \parallel ch_2, \sigma_2) \xRightarrow{l_2} (ch_1, \sigma_2')}\ \texttt{(parallel-4)}$$

Figure 3: Rules defining the transitions in the semantics of choreographies.

$\sigma\,[\{rcv\_list\}.\texttt{var} \mapsto \sigma(snd.\texttt{var})]$, it is state σ where we apply the substitution that assigns all the variables in $\{rcv\_list\}.\texttt{var}$ to $\sigma(snd.\texttt{var})$.

Additionally, to model asynchronous communication, we utilise two rules: the first to execute the send function, and the second to execute the receive function on each port. This requires a transient configuration, which contains the remaining ports for which the receive function needs to be executed. This configuration corresponds to the asynchronous message being "in transit". This state is modeled as a set of pairs of ports with their functions (i.e., $2^{\mathcal{P}^{\text{r}} \times \mathcal{F}(X)}$).

We are now able to define the semantics of choreographies.

Definition 5 (Semantics of choreography model). *The semantics of choreographies is an LTS* $(ChorConf, ChorLab, \Rightarrow)$ *where :*

- $ChorConf \subseteq (Chors \times ChorState) \cup ChorState \cup 2^{\mathcal{P}^r \times \mathcal{F}(X)}$ *is the set of configurations and* $ChorState \subseteq ChorConf$ *is the set of final configurations;*

- $ChorLab \subseteq \left(2^{\mathcal{P}} \setminus \{\emptyset\} \cup \{\tau\}\right)$ *is the set of labels where each label is either a set of ports or label* τ *for silent transitions;*

- $\Rightarrow \subseteq ChorConf \times ChorLab \times ChorConf$ *is the least set of (labelled) transitions satisfying the rules in Figure 3;*

Whenever for two configurations $c, c' \in ChorConf$ and a label $l \in ChorLab$, $(c, l, c') \in \Rightarrow$, we note it $c \overset{l}{\Rightarrow} c'$. The rules in Figure 3 can be intuitively understood as follows:

- Rule `(nil)` states that choreography `nil` terminates in any state σ and produces the terminal configuration σ.

- Rule `(synch-sendrcv)` describes the synchronous send/receive primitive. The component of port snd transfers data to the components with the receive ports in rcv_list whenever the guard g attached to snd holds true from the starting state σ. If the list of receive ports (with update functions) is $pr_1[f_1], \ldots, pr_k[f_k]$, the choreography terminates in a state obtained after the data transfer defined by $\text{send}(\sigma, snd, \{pr_1, \ldots, pr_k\})$ and the applications of the update functions $f, f_1, \ldots, f_k$ of the send and receive ports. Note that the application order does not influence the resulting state as these update functions apply to disjoint variables.

- Rule `(asynch-sendrcv-1)` describes the first part of an asynchronous send/receive primitive. As in the synchronous send/receive primitive, the component of port snd transfers data to the components with the receive ports in rcv_list whenever the guard g attached to snd holds true from the starting state σ. However, the state of the receiving component is only updated with the transferred data (with $\text{send}(\sigma, snd, \{pr_1, \ldots, pr_k\})$) and the receiving components do not apply their update functions.

- Rule `(asynch-sendrcv-2)` describes the second part of an asynchronous send/receive primitive. A receive port $pr[f]$ in the list of receive ports to be executed rcv_list applies the attached updated function f to the current state and is removed from the list of received ports to be executed.

- Rule `(master-branching)` describes the (conditional) master branching from component B on one of its continuations $snd_j[g_j, f_j] : ch_j$ whenever the guard g_j attached to port snd_j holds true. The resulting configuration consists of the choreography ch_j and the state $f_j(\sigma)$ (resulting from the application of the attached update function f_j to σ).

- Rule `(iterative-tt)` describes the first case of the iterative composition of a choreography ch under the condition $snd[g, f]$ (which consists of a send port snd, a guard g, and an update function f). When g holds true in σ, the resulting configuration consists of the choreography ch sequentially composed with the same starting choreography to be executed in state σ updated by f.

- Rule `(iterative-ff)` describes the second case of the iterative composition of a choreography ch under the condition $snd[g, f]$. When g holds false in σ, the choreography terminates in the (unmodified) state σ.

- Rules `(sequential-1)` and `(sequential-2)` describe the possible evolutions of two sequentially composed choreographies ch_1 and ch_2. Rule `(sequential-1)` describes the case where the execution of choreography ch_1 does not terminate and evolves to a configuration (ch_1, σ_1') which leads to the global configuration $(ch_1' \bullet ch_2, \sigma_1')$. Rule `(sequential-2)` describes the case where the execution of choreography ch_1 terminates and evolves to a final configuration σ_1' which leads to the global configuration (ch_2, σ_1') (where the second choreography ch_2 is to be executed in state σ_1').

- Rules `(parallel-1)` to `(parallel-4)` describe the possible evolutions of two choreographies ch_1 and ch_2 composed in parallel. Rules `(parallel-1)` and `(parallel-2)` describe the evolutions where ch_1 performs a computation step and terminates or not. Rules `(parallel-3)` and `(parallel-4)` describe the evolutions where ch_2 performs a computation step.

6. Example: Two-Phase Commit

Overview. The two-phase commit protocol (2PC) is a distributed algorithm that allows distributed processes to perform a transaction atomically. To do so, one process is designated to be the coordinator, the rest we refer to them as workers. The coordinator initiates the transaction by notifying all workers to begin. Each worker then takes the necessary steps to perform the transaction answering the coordinator with either an acknowledgement or requesting an abort on failure. Once all workers have voted, the coordinator then sends the final request to commit or abort the transaction, after which all works acknowledge the commit or rollback.

Components. We model the following protocol using global choreographies (Section 5). In our setting, we have n workers and 1 coordinator.

For each worker $i \in [1..n]$ we associate a worker component $\mathtt{W_i}$. Component $\mathtt{W_i}$ has the following variables: $\mathtt{ok_i}$ and $\mathtt{id_i}$. The variable $\mathtt{ok_i}$ is a boolean used to convey the positive or negative acknowledgement, it is initially set to `false`, while the variable $\mathtt{id_i}$ contains a unique identifier of the worker. Additionally, for each worker component, we associate the ports: $\mathtt{vote_i(id_i, ok_i)}$, $\mathtt{prepare_i}$, $\mathtt{ack_i}$, and $\mathtt{fail_i}$. Port $\mathtt{vote_i}$ is used to send to the coordinator the identifier and a positive or negative acknowledgment. Port $\texttt{start}_\texttt{i}$ is used to prepare the transaction, port $\texttt{ack}_\texttt{i}$ is used to request the final commit, while port $\texttt{fail}_\texttt{i}$ is used to request a rollback.

The coordinator component is denoted by C and has the following variables: rok, rid, cs, and res. Variables rok and rid are used to receive a worker's vote, and are used to store its acknowledgment and identifier. Variable cs is a set of worker identifiers, and is used to keep track of which worker(s) voted, it is initialized to the empty set. Variable res is a boolean, it contains the result of the vote, it is initially set to true. The interface of the coordinator component consists of the following ports: begin, proceed, cond, and recv(rid, rok). Port begin is used to notify workers to prepare the transaction, while port proceed is used to notify them of a commit or failure. Port cond is used for branching between either requesting a commit or a rollback. Port recv is used to receive a worker's vote. To simplify the state reset between communication, we define update function $\texttt{reset}() = [res = \texttt{true}; cs = \emptyset]$.

Choreographies. In order to be general, we assume for each worker process three choreographies: $\texttt{stage}_\texttt{i}$, $\texttt{commit}_\texttt{i}$, and $\texttt{roll}_\texttt{i}$. Choreography $\texttt{stage}_\texttt{i}$ performs the operation before committing, and sets a variable $\texttt{ok}_\texttt{i}$ to true if the operation succeeded or false otherwise. Choreography $\texttt{commit}_\texttt{i}$ is performed when all workers have committed, while choreography $\texttt{roll}_\texttt{i}$ is executed whenever at least one worker failed. We assume the three choreographies do not interfere with $\texttt{ok}_\texttt{i}$ and $\texttt{id}_\texttt{i}$ in any other way.

The protocol is expressed as a sequential composition of two phases, where the second phase depends on the vote of the first phase. For each phase, the coordinator interacts with each worker in parallel.

$$\begin{pmatrix} \texttt{phase1}_1 \parallel \\ \vdots \\ \parallel \texttt{phase1}_n \end{pmatrix} \bullet \quad \texttt{C} \oplus \{\texttt{C.cond}[|\texttt{cs}| = n \wedge \texttt{res}, \texttt{reset}] : \begin{pmatrix} \texttt{phase2a}_1 \parallel \\ \vdots \\ \parallel \texttt{phase2a}_n \end{pmatrix},$$

$$\texttt{C.cond}[\neg(|\texttt{cs}| = n \wedge \texttt{res},), \texttt{reset}] : \begin{pmatrix} \texttt{phase2b}_1 \parallel \\ \vdots \\ \parallel \texttt{phase2b}_n \end{pmatrix}\}$$

$$\begin{array}{lcl} \multicolumn{3}{l}{\forall i \in [1..n]:} \\ \texttt{phase1}_\texttt{i} & = & \{\texttt{C.begin}[\texttt{true}, \emptyset] \longrightarrow \{\texttt{W}_\texttt{i}.\texttt{prepare}_\texttt{i}[\texttt{ok}_\texttt{i} := \texttt{false}]\}\} \bullet \texttt{stage}_\texttt{i} \bullet \\ & & \{\texttt{W}_\texttt{i}.\texttt{vote}_\texttt{i}[\texttt{true}, \emptyset] \longrightarrow \{\texttt{C.recv}[\texttt{res} = \texttt{res} \wedge \texttt{rok}; \texttt{cs} = \texttt{cs} \cup \{\texttt{rid}\}]\}\} \\ \texttt{phase2a}_\texttt{i} & = & \{\texttt{C.proceed}[\texttt{true}, \emptyset] \longrightarrow \{\texttt{W}_\texttt{i}.\texttt{ack}_\texttt{i}[\texttt{ok}_\texttt{i} = \texttt{true}]\}\} \bullet \texttt{commit}_\texttt{i} \bullet \\ & & \{\texttt{W}_\texttt{i}.\texttt{vote}_\texttt{i}[\texttt{true}, \emptyset] \longrightarrow \{\texttt{C.recv}[\texttt{res} = \texttt{res} \wedge \texttt{rok}; \texttt{cs} = \texttt{cs} \cup \{\texttt{rid}\}]\}\} \\ \texttt{phase2b}_\texttt{i} & = & \{\texttt{C.proceed}[\texttt{true}, \emptyset] \longrightarrow \{\texttt{W}_\texttt{i}.\texttt{fail}_\texttt{i}[\texttt{ok}_\texttt{i} = \texttt{true}]\}\} \bullet \texttt{roll}_\texttt{i} \bullet \\ & & \{\texttt{W}_\texttt{i}.\texttt{vote}_\texttt{i}[\texttt{true}, \emptyset] \longrightarrow \{\texttt{C.recv}[\texttt{res} = \texttt{res} \wedge \texttt{rok}; \texttt{cs} = \texttt{cs} \cup \{\texttt{rid}\}]\}\} \end{array}$$

In the first phase ($\texttt{phase1}_\texttt{i}$), the coordinator initiates the transaction ($\texttt{C.begin} \longrightarrow \texttt{W}_\texttt{i}.\texttt{prepare}_\texttt{i}$). Then the worker performs the staging choreography ($\texttt{stage}_\texttt{i}$), and once it is complete, communicates its' result (stored in $\texttt{ok}_\texttt{i}$) and its' identifier to the coordinator (using its interface $\mathtt{W_i.vote_i}$). Upon reception, the coordinator updates the vote by performing a conjunction ($\mathtt{res} = \mathtt{res} \wedge \mathtt{rok}$), so as to ensure *all* workers vote to commit, and updates the workers list by adding the worker identifier ($\mathtt{cs} = \mathtt{cs} \cup \{\mathtt{rid}\}$). We note here, that while there is overlap on the port `C.begin` and the receiving variables `cs` and `res`, that it is easy to resolve such overlap, as the variables are updated using an associative and commutative operators ($\wedge$ and $\cup$) which are not affected by order of reception. (Something to be said about the variables `rok` and `rid` being that each receive binds those, and they cannot be overritten.)

When initiating the second phase, the coordinator branches to verify that all workers voted ($|\mathtt{cs}| = n$), and that their vote was `true` ($\mathtt{res} = \mathtt{true}$). If the condition is satisfied, the coordinator initiates parallel composition of choreographies to commit ($\mathtt{phase2a_i}$). Otherwise it initiates a parallel composition of choreographies to rollback ($\mathtt{phase2b_i}$). For both branches, the coordinator resets the state of the vote (`reset`), to refresh acknowledgments. Each choreography $\mathtt{phase2a_i}$ notifies the port $\mathtt{ack_i}$ which is followed by worker performing $\mathtt{commit_i}$ and returning an acknowledgement. Alternatively, $\mathtt{phase2b_i}$ notifies the port $\mathtt{fail_i}$ which is followed by worker performing $\mathtt{roll_i}$ and returning an acknowledgement.

7. Distributed Component-based Framework

In this section, we introduce a component-based framework, inspired from the Behavior Interaction Priority framework (BIP) [5]. In the BIP framework, atomic components communicate through an interaction model defined on the interface ports of the atomic components. Moreover, all ports have the same type. Unlike BIP, we distinguish between four types of ports: (1) synchronous send; (2) asynchronous send; (3) asynchronous receive; and (4) internal ports. The new port types allow to (1) easily model distributed system communication models; (2) provide efficient code generation, under some constraints, that does not require to build controllers to handle conflicts between multiparty interactions.

7.1. Atomic Components

Atomic components are the main computation blocks. Atomic components are endowed with a set of variables used in their computation. An atomic component is defined as follows.

Definition 6 (Atomic component - syntax). *An atomic component B is a tuple $(P,\ X, L, T)$, where P is a set of ports; X is a set of variables such that $X \subseteq$ Vars and $P.\mathtt{var} \subseteq X$; L is a set of control locations; and $T \subseteq (L \times P \times \mathcal{G}(X) \times \mathcal{F}(X) \times L)$ is a set of transitions.*

Transitions make the system move from one control location to another by executing a port. Transitions are guarded and are associated with the execution of an update function. In a transition $(\ell, p, g, f, \ell') \in T$, ℓ and ℓ' are respectively the source and destination location, p is the executed port, g is the guard, and f is the update function.

The semantics of an atomic component is defined as an LTS. A state of the LTS consists of a location ℓ and valuation v of the variables where a valuation is a function from the variables of the component to a set of values. The atomic component can transition from state (ℓ, v) to state (ℓ', v') using a transition $(\ell, p, d, g, f, \ell') \in T$ if (i) the guard of the transition holds ($g(v)$ holds true) (ii) the application of update function f to valuation v_{pd}/v yields v' where v_{pd} is the valuation associating $p.\mathtt{var}$ with $d \in \mathit{Data}$, which is a value possibly received from other components.

Definition 7 (Atomic component - semantics). *The semantics of an atomic component (P, X, L, T) is a labelled transition system, i.e., a tuple $(Q, \mathcal{P} \times \mathit{Data}, \rightarrow)$, where:*

- *$Q \subseteq L \times [X \rightarrow \mathit{Data}]$ is the set of states,*
- *$\mathcal{P} \times \mathit{Data}$ is the set of labels where a label is a pair made of a port and a value, and*
- *$\rightarrow \subseteq Q \times P \times \mathit{Data} \times Q$ is the set of transitions defined as:*

$$\{((\ell, v), (p, d), (\ell', v')) \mid \exists (\ell, p, g, f, \ell') \in T : g(v) \wedge v' = f(v_{pd}/v)\}.$$

When $(q, (p, d), q') \in T$, we note it $q \xrightarrow{p/d} q'$. Moreover, we use states as functions: for $x \in X$ and $q = (l, v)$, $q(x)$ is a short for $v(x)$.

To later construct a system, we shall use a set of n atomic components $\{B_i = (P_i, Q_i, T_i)\}_{i=1}^{n}$

Synchronization between the atomic components is defined using the notion of interaction.

Definition 8 (Interaction). *An interaction from component B_i to components $\{B_j\}_{j \in J}$, where $i \notin J$, is a pair $(p_i, \{p_j\}_{j \in J})$, where:*

- *p_i is its send port (synchronous or asynchronous) that belongs to the send ports of atomic component B_i, i.e., $p_i \in \mathcal{P}_i^{\mathrm{ss}} \cup \mathcal{P}_i^{\mathrm{as}}$;*
- *$\{p_j\}_{j \in J}$ is the set of receive ports, each of which belongs to the receive ports of atomic component B_j, i.e., $\forall j \in J : p_j \in \mathcal{P}_j^{r}$.*

An interaction $(p_i, \{p_j\}_{j \in J})$ is said to be synchronous (resp. asynchronous) iff $\mathtt{isSSend}(p_i)$ (resp. $\mathtt{isASend}(p_i)$) holds.

7.2. Composite Components

A composite component consists of several atomic components and a set of interactions. The semantics of a composite component is defined as a labeled transition system where the transitions depend on the interaction types.

$$\frac{\begin{array}{l}\texttt{isSSend}(p_i) \\ a = (p_i, \{p_j\}_{j\in J}) \in \gamma \\ d = q_i(p_i.\texttt{var}) \in Data\end{array} \quad \begin{array}{l}\forall k \in J \cup \{i\} : q_k \xrightarrow{p_k/d} q_k' \\ \forall k \notin J \cup \{i\} : q_k = q_k'\end{array} \quad \forall j \in J : q_j(p_j.\texttt{buff}) = \epsilon}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')} \; \texttt{(synch-send)}$$

$$\frac{\begin{array}{l}\texttt{isASend}(p_i) \\ a = (p_i, \{p_j\}_{j\in J}) \in \gamma \\ d = q_i(p_i.\texttt{var}) \in Data\end{array} \quad \begin{array}{l}\forall k \in J \setminus \{i\} : q_k' = q_k \\ q_i \xrightarrow{p_i/d} q_i'\end{array} \quad \begin{array}{l}\forall j \in J : \\ q_j'(p_j.\texttt{buff}) = q_j(p_j.\texttt{buff}) \cdot d\end{array}}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')} \; \texttt{(asynch-send)}$$

$$\frac{\texttt{isRecv}(p_j) \quad \begin{array}{l} q_j \xrightarrow{p_j/d} q_j' \\ \forall k \neq j : q_k = q_k'\end{array} \quad \begin{array}{ll} q_j(p_j.\texttt{buff}) = d \cdot D & d \in Data \\ q_j'(p_j.\texttt{buff}) = D & D \in Data^*\end{array}}{(q_1, \ldots, q_n) \xrightarrow{\tau} (q_1', \ldots, q_n')} \; \texttt{(recv)}$$

$$\frac{\texttt{isInternal}(p_i) \quad q_i \xrightarrow{p_i/\bot_{\mathrm{d}}} q_i' \quad \forall k \neq i : q_k = q_k'}{(q_1, \ldots, q_n) \xrightarrow{\tau} (q_1', \ldots, q_n')} \; \texttt{(internal)}$$

Figure 4: Semantic rules defining the behavior of composite components.

Definition 9 (Composite component). *A composite component built over atomic components $B_1, \ldots, B_n$ and parameterized by a set of interactions γ, noted $\gamma(B_1, \ldots, B_n)$, is defined as a transition system $(Q, \gamma \cup \{\tau\}, \rightarrow)$, where :*

- $Q = \bigotimes_{i=1}^{n} Q_i$ *is the set of configurations,*
- $\gamma \cup \{\tau\}$ *is the set of labels which consist of interactions and τ for silent transitions, and*
- $\rightarrow$ *is the least set of transitions satisfying the rules in Figure 4.*

The semantic rules in Figure 4 can be intuitively understood as follows:

- Rule `(synch-send)` describes synchronous interactions, i.e., the interactions of the form $(p_i, \{p_j\}_{j\in J})$ where $\texttt{isSSend}(p_i)$, where some component B_i synchronously sends to some components $B_j, j \in J$. The variable attached to port p_i of B_i ($p_i.\texttt{var}$) gets evaluated to some value $d \in Data$, which is transmitted. All components B_k, $k \in J \cup \{i\}$, perform a transition $q_k \xrightarrow{p_k/d} q_k'$, and other components do not move ($q_k = q_k'$ for $k \notin J \cup \{i\}$). The rule requires that all the corresponding receive ports have no pending messages (their buffers are empty, i.e., $\forall j \in J : q_j(p_j.\texttt{buff}) = \epsilon$). The states of all the involved components are simultaneously updated through the transition $q_k \xrightarrow{p_k/d} q_k'$, for $j \in J \cup \{i\}$.
- Rule `(asynch-send)` describes asynchronous interactions, i.e., the interactions of the form $(p_i, \{p_j\}_{j\in J})$ where $\texttt{isSSend}(p_i)$, where some component

B_i asynchronously sends to some components $B_j, j \in J$. The rule resembles the previous one, except that it does not require the participation of the receiving components. Only the sending component performs a transition $q_i \xrightarrow{p_i/d} q_i'$ and the receiving components (as well as the other components) do not move. Value $d \in \mathit{Data}$ is appended to the buffer of the corresponding receive ports ($\forall j \in J : q_j'(p_j.\mathtt{buff}) = q_j(p_j.\mathtt{buff}) \cdot d$).

- Rule `(recv)` describes the autonomous execution of receive port p_j of some component B_j. The rule requires that the buffer of port p_j is non-empty ($q_j(p_j.\mathtt{buff}) = d \cdot D$, with $d \in \mathit{Data}$ and $D \in \mathit{Data}^*$). The execution of this interaction makes component B_j perform a transition $q_j \xrightarrow{p_j/d} q_j'$ and consumes value d in buffer $p_i.\mathtt{buff}$.

- Rule `(internal)` describes the autonomous execution of an internal port p_i of component B_i where only the local state of B_i is updated by performing the transition $q_i \xrightarrow{p_i/\perp_d} q_i'$.

Finally, a system is defined as a composite component where we specify the initial states of its atomic components.

Definition 10 (System). *A system is a pair* $(\gamma(B_1, \ldots, B_n), \mathtt{init})$*, made of a composite component and* $\mathtt{init} \in \bigotimes_{i=1}^{n} Q_i$ *its initial state.*

8. Transformations

We start with a composite component consisting of n atomic components $\{B_1, \ldots, B_n\}$ with their interface ports and variables. That is, the behaviors of the input atomic components are empty. Atomic components can be considered as services with their interfaces but with undefined behaviors.

In this section, we define how to automatically synthesize the behavior of atomic components corresponding to a global choreography model `ch`. The distributed system associated with `ch` is noted $[\![\mathtt{ch}]\!]$, and is inductively defined over `ch`. To realize choreographies as atomic components we follow the syntactic structure of the choreography. This facilitates the definition of the transformation from choreographies to components and lead to a clearer implementation.

8.1. Preliminary Notions and Notation

We introduce some preliminary concepts and notations that will serve the realization of choreographies as components. As we are inductively transforming choreographies to components, we need to synchronize the execution of the independently generated choreographies. For this, we define three auxiliary functions that takes a choreography as input and give the components that:

- are involved in the realization of the choreography – function $\mathcal{C}$.

- need to be notified for the choreography to start – function `start`,

- need to terminate for the choreography to terminate – function `end`,

The definitions of the two latter functions follow from the semantics of choreographies (Definition 5). Note, in the following definitions, when referring to a port p with a guard and/or update function involved in a choreography, we note $p[-]$ when the guard and/or update function is irrelevant to the definition.

Function $\mathcal{C}$. We define $\mathcal{C}(\mathtt{ch})$ as the set of indexes of all components involved in choreography `ch`.

Definition 11 (Function $\mathcal{C}$). *Function $\mathcal{C}$: Choreographies $\rightarrow 2^{[1,n]} \setminus \{\emptyset\}$ is inductively defined over choreographies as follows:*

$$\begin{aligned}
\mathcal{C}(psas) =&\ \{i\} \text{ if } \exists i \in [1,n] : psas \in \mathcal{P}_i^{\mathrm{ss}} \cup \mathcal{P}_i^{\mathrm{as}}\\
\mathcal{C}(pr[-]) =&\ \{i\} \text{ if } \exists i \in [1,n] : pr \in \mathcal{P}_i^{\mathrm{r}}\\
\mathcal{C}(pr[-], \mathtt{rcv_list}) =&\ \mathcal{C}(pr[-]) \cup \mathcal{C}(\mathtt{rcv_list})\\
\mathcal{C}(\mathtt{nil}) =&\ \emptyset\\
\mathcal{C}(\mathtt{snd} \longrightarrow \{\mathtt{rcv_list}\}) =&\ \mathcal{C}(\mathtt{snd}) \cup \mathcal{C}(\mathtt{rcv_list})\\
\mathcal{C}(B_i \oplus \{\mathtt{cont_list}\}) =&\ \{i\} \cup \mathcal{C}(\mathtt{cont_list})\\
\mathcal{C}\big(\mathtt{while}(\mathtt{snd})\, \mathtt{ch}\, \mathtt{end}\big) =&\ \mathcal{C}(\mathtt{snd}) \cup \mathcal{C}(\mathtt{ch})\\
\mathcal{C}(\mathtt{ch}_1 \bullet \mathtt{ch}_2) =&\ \mathcal{C}(\mathtt{ch}_1) \cup \mathcal{C}(\mathtt{ch}_2)\\
\mathcal{C}(\mathtt{ch}_1 \parallel \mathtt{ch}_2) =&\ \mathcal{C}(\mathtt{ch}_1) \cup \mathcal{C}(\mathtt{ch}_2)
\end{aligned}$$

Function `start`. We define `start(ch)` as the set of indexes of the components in `ch` that should be notified to trigger the start of `ch`.

Definition 12 (Function `start`). *Function `start` : Choreographies $\rightarrow 2^{[1,n]} \setminus \{\emptyset\}$ is inductively defined over choreographies as follows:*

$$\begin{aligned}
\mathtt{start}(\mathtt{nil}) =&\ \emptyset\\
\mathtt{start}(\mathtt{snd} \longrightarrow \{\mathtt{rcv_list}\}) =&\ \mathcal{C}(\mathtt{snd})\\
\mathtt{start}(B \oplus \{\mathtt{cont_list}\}) =&\ \mathcal{C}(B)\\
\mathtt{start}\big(\mathtt{while}(\mathtt{snd})\, \mathtt{ch}\, \mathtt{end}\big) =&\ \mathcal{C}(\mathtt{snd})\\
\mathtt{start}(\mathtt{ch}_1 \bullet \mathtt{ch}_2) =&\ \mathtt{start}(\mathtt{ch}_1)\\
\mathtt{start}(\mathtt{ch}_1 \parallel \mathtt{ch}_2) =&\ \mathtt{start}(\mathtt{ch}_1) \cup \mathtt{start}(\mathtt{ch}_2)
\end{aligned}$$

Intuitively, to start a simple synchronous or asynchronous send/receive, the component of its corresponding send port should be notified. Conditional master branching choreographies can be started by notifying their corresponding master component. Iterative choreographies can be started by notifying the component of its corresponding send port. A choreography consisting of the sequential composition of two choreographies can be started by notifying the components that can start the first choreography. A choreography consisting of the parallel composition of two choreographies can be started by notifying the components that can start the two choreographies of the composition.

Function `end`. Similarly, we define $\texttt{end}(\texttt{ch})$ as the set of indexes of the components involved in `ch` that need to terminate so that `ch` terminates.

Definition 13 (Function `end`). *Function* $\texttt{end} : \mathit{Choreographies} \to 2^{[1,n]} \setminus \{\emptyset\}$ *is inductively defined over choreographies as follows:*

$$
\begin{aligned}
\texttt{end}(\texttt{nil}) &= \ \emptyset \\
\texttt{end}(\mathit{snd}[-] \longrightarrow \{\texttt{rcv\_list}\}) &= \ \mathcal{C}(\texttt{rcv\_list}) \ \mathit{if}\ \mathit{snd} \in \mathcal{P}^{\mathrm{ss}} \\
\texttt{end}(\mathit{snd}[-] \longrightarrow \{\texttt{rcv\_list}\}) &= \ \mathcal{C}(\mathit{snd}) \ \mathit{if}\ \mathit{snd} \in \mathcal{P}^{\mathrm{as}} \\
\texttt{end}(B \oplus \{\texttt{cont\_list}\}) &= \ \mathcal{C}(\texttt{cont\_list}) \\
\texttt{end}\big(\texttt{while}(\texttt{snd})\,\texttt{ch}\,\texttt{end}\big) &= \ \mathcal{C}(\texttt{snd}) \\
\texttt{end}(\texttt{ch}_1 \bullet \texttt{ch}_2) &= \ \texttt{end}(\texttt{ch}_2) \\
\texttt{end}(\texttt{ch}_1 \parallel \texttt{ch}_2) &= \ \texttt{end}(\texttt{ch}_1) \cup \texttt{end}(\texttt{ch}_2)
\end{aligned}
$$

We consider that a synchronous send/receive is terminated when all the components involved in the sending and receiving ports are terminated. However, if the send part is asynchronous, any subsequent choreography can start after the sending is complete. Conditional master branching choreographies are terminated when the corresponding master component has terminated. Iterative choreographies are terminated when the component of the send port (with its guard used as condition) has terminated. A choreography consisting of the sequential composition of two choreographies has terminated when the second choreography in the composition has terminated. A choreography that consists of the parallel composition of two choreographies has terminated when the first and second choreographies have terminated.

Representing components. In the sequel, we represent receive ports (resp. synchronous send, asynchronous send) using dashed square labeled with r (resp. circle with solid border labeled with ss, circle with dashed border labeled with as). We also omit the border for send ports when synchrony is out of context and label it with s.

8.2. Generation of Distributed CBSs

We consider a global choreography `ch` defined over the set of ports $\mathcal{P} = \cup_{i=1}^{n} P_i$ of a given set of atomic components (with empty behavior) with their corresponding variables. Given a choreography `ch`, we define a set of transformations that allows to generate the behaviors and the corresponding interactions of the distributed components $S = (B, \texttt{init})$. Moreover, as we progressively build system S, we consider that it has a context to denote the current state where a choreography should be appended. For this, $\mathcal{S} = (S, \texttt{context})$ denotes a system with its corresponding context where `context` is a function that takes an atomic component as input and returns a location, i.e., $\texttt{context}(B_i) \in L_i$ to denote the current context of atomic components B_i. The building of the final system is done by induction, following the syntactic structure of the input choreography and uses the continuously updated context. Any step for constructing the component ensures that the context of each component consists of a unique state.

Initially, we consider a system skeleton $\mathcal{S} = (S, \texttt{context})$, where $B = \gamma(B_1, \ldots, B_n)$ with: (1) $\gamma = \emptyset$; (2) $B_i = (P_i, \emptyset, \{l_i\}, \emptyset)$; (3) $init = (l_1^{\texttt{init}}, \ldots, l_n^{\texttt{init}})$; and (4) $\texttt{context}(B_i) = l_i^{\texttt{init}}$; for $i \in [1, n]$. The initial location of the obtained system remains unchanged, i.e., it is *init*. As such, for the sake of clarity, we omit it in our construction. Moreover, all variables are initialized to their default value.

8.2.1. Send/Receive

Send/receive choreography updates the participating components by adding a transition from the current context and labeling it by the corresponding send or receive port from the choreography. In order to avoid inconsistencies between same ports but from different choreographies, we create a copy of each port of the choreography (`copy`). $\texttt{copy}(p)$ is a new port that has the same function and guard, but a different name. We also add the corresponding interaction between the send and the receive ports. Finally, we update the context of the participants to be the corresponding new added states. As such, if the initial context of each component consists of one state, then the resulting system (after applying the send/receive choreography) also guarantees that each of its components also consists of one state. Note that an interaction connected to a synchronous send port and receive ports can be considered as a multiparty interaction with a master trigger, which is the send port. As such, this allows to efficiently implement multiparty interactions.

Remark 2. *Creating a copy for each port per choreography is necessary to generate efficient and correct distributed implementation. As for efficiency, consider the choreography $p_1 \longrightarrow \{p_2\} \bullet p_1 \longrightarrow \{p_3\}$. Its corresponding distributed implementation would require to create two interactions $(p_1, \{p_2\})$ and $(p_1, \{p_3\})$. As such, the component that corresponds to p_1 (B_1) needs to interact B_2 and B_3 to know which interaction must be executed (depending on their current enable ports). However, if we create a copy of the ports, each port will be connected to one and only interaction, hence component B_1 can locally decide, without interacting with other components, on the interaction to be executed. As for correctness, consider the choreography $p_1 \longrightarrow \{p_2, p_3\} \bullet p_1 \longrightarrow \{p_2\}$. According to the choreography semantics, we should first execute $p_1 \longrightarrow \{p_2, p_3\}$ then $p_1 \longrightarrow \{p_2\}$. Consider that we are in a state where p_1 and p_2 are enabled but p_3. This may happen when the component that corresponds to p_3 is still executing the function of the previous transition. In this case, B_1 would interact with B_2 and B_3 to know which interaction to execute. As p_3 is not currently enabled, component B_1 will execute the interaction connected with p_2 only, hence violating the sequential semantics.*

Definition 14 (Send/Receive).

$$[\![psas[g, f] \longrightarrow \{\texttt{rcv\_list}\}]\!](\gamma(B_1, \ldots, B_n), \texttt{context}) = (\gamma'(B'_1, \ldots, B'_n), \texttt{context}'), \textit{with:}$$

- $B'_k = \begin{cases} (P_k, L'_k, T'_k) & \textit{if } k \in \mathcal{C}(psas[g, f]) \cup \mathcal{C}(\texttt{rcv\_list}) \\ B_k & \textit{otherwise} \end{cases}$ *, where:*

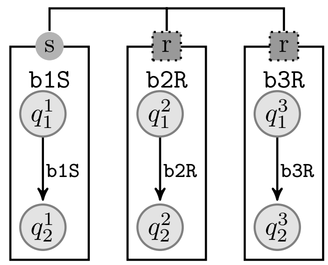

Figure 5: Send/Receive Transformation

- $L'_k = L_k \cup \{l_k^{\mathrm{new}}\}$
- $T'_k = T_k \cup \begin{cases} \{\mathtt{context}(B_k) \xrightarrow{\mathtt{copy}(psas),g,f} l_k^{\mathrm{new}}\} & \textit{if } psas[g,f] \in B_k.\mathcal{P}^{\mathrm{ss}} \cup B_k.\mathcal{P}^{\mathrm{as}} \\ \{\mathtt{context}(B_k) \xrightarrow{\mathtt{copy}(p_k),\mathtt{true},p_k.\mathtt{ufct}} l_k^{\mathrm{new}}\} & \textit{if } p_k \in \mathtt{rcv\_list} \end{cases}$

- $\gamma' = \gamma \cup \{(\mathtt{copy}(psas), \{\mathtt{copy}(p_i) \mid p_i \in \mathtt{rcv\_list}\})\}$,
- $\mathtt{context}'(B'_k) = \begin{cases} l_k^{\mathrm{new}} & \textit{if } k \in \mathcal{C}(psas[g,f]) \cup \mathcal{C}(\mathtt{rcv\_list}) \\ \mathtt{context}(B_k) & \textit{otherwise} \end{cases}$.

Atomics components that do not participate in the send/receive choreography remain unchanged. Atomic components that participate in the send/receive are updated by adding a transition from their context location to a new location (l_k^{new}). We label this transition with a copy of the corresponding port. We create an interaction that connects the send ports to the receive ports. The new context becomes the new created location.

Example 1 (Send/Receive). *Figure 5 shows an abstract example on how to transform a simple send/receive choreography,* `b1S` $\longrightarrow$ {`b2R`, `b3R`}*, into an initial system consisting of three components with interfaces:* `b1S` *(send, synchronous or asynchronous),* `b2R` *(receive), and* `b3R` *(receive), respectively.*

8.2.2. Branching Composition

Recall that conditional master branching of the form $B_i \oplus \{p_i^l[g_i, f_i] : \mathtt{ch}_l\}_{l \in L}$, allows for the modeling of conditional choice between several choreographies. The choice is made by a specific component (B_i), which depending on its internal state would enable some its guards (g_i). Accordingly, it notifies the appropriate components by sending a label (p_i^l), to follow the taken choice (i.e., the corresponding choreography, $\mathtt{ch}_l$). We apply branching by independently integrating the choreography for each choice. This can be done by letting B_i notifying the participants, i.e., $\mathcal{C}(B_i \oplus \{p_i^l[-] : \mathtt{ch}_l\}_{l \in L}) \setminus \{i\}$, of the choreography ($\mathtt{ch}_l$) of that choice ($p_i^l$). For that purpose, we create new receive ports ($\{p_k^{\mathtt{cr}_l}\}_{k \in K}$) to be able to receive the corresponding choice.

For this, we define a union operator, noted `union`, that takes a set of systems with their contexts and (1) unions all of their locations, transitions and ports; then (2) updates the contexts of the obtained components by joining each of their input contexts with internal transitions. Therefore, after applying branching we guarantee that each component will have one and only one context location. Formally, operator `union` is defined as follows.

Definition 15 (Union). *The union of systems with their contexts* $\{(S_l, \texttt{context}_l)\}_{l\in L}$*, where* $S_l = \gamma^l(B_1^l, \ldots, B_n^l)$ *and* $B_i^l = (P_i^l, X_i, L_i^l, T_i^l)$ *for* $i \in [1, n]$ *and* $l \in L$*, noted* $\texttt{union}(\{(S_l, \texttt{context}_l)\}_{l\in L})$*, is defined as the system with context* $(\gamma(B_1, \ldots, B_n), \texttt{context})$*, where:*

- $\gamma = \bigcup_{l\in L} \gamma^l$;
- $B_i = (\bigcup_{l\in L} P_i^l, \bigcup_{l\in L} X_i^l, \bigcup_{l\in L} L_i^l \cup \{l_i^u\}_{l\in L}, \bigcup_{l\in L} T_i^l \cup T_i^{\texttt{merge}})$ *with* l_i^u *a new location and* $T_i^{\texttt{merge}} = \{\texttt{context}_l(B_i^l) \xrightarrow{\epsilon} q_i^c \mid l \in L\}$;
- $\texttt{context}(B_i) = l_i^u$ *for* $i \in [1, n]$.

Then, branching as described by independently applying each choice, then doing the union.

Definition 16 (Branching).

$$\begin{aligned}&[\![B_i \oplus \{p_i^l[g_l, f_l] : \texttt{ch}_l\}_{l\in L}]\!](S, \texttt{context})\\ &\quad = \texttt{union}\left(\{[\![\texttt{ch}_l]\!][\![p_i^l[g_l, f_l] \longrightarrow \{p_k^{\texttt{cr}_l}[\emptyset]\}_{k\in K}]\!](S, \texttt{context})\}_{l\in L}\right)\end{aligned}$$

Where, $K = \mathcal{C}(B_i \oplus \{p_i^l[-] : \texttt{ch}_l\}_{l\in L}) \setminus \{i\}$.

Remark 3. *Note that we require to notify all the participants of a choice and not only the start components. Consider the following choreography (where* α *and* β *denote some choreographies):*

$$B_1 \oplus \{p_1^l[-] : p_2[-] \longrightarrow p_3[-] \bullet \alpha;\ p_2^l[-] : p_2[-] \longrightarrow p_3[-] \bullet \beta\}$$

In this choreography, if we would have not sent the choice made by component 1 *to component* 3*, then component* 3 *cannot know about the decision that was taken by component* 1*. Hence, it cannot decide whether to follow choreography* α *or* β *afterwards.*

Example 2 (Branching). *Figure 6 shows an abstract example on how to apply a branching operation that consists of two choices* $B_1 \oplus \{b1^{l_1}[g_1, f_1] : \texttt{ch}_1, b2^{l_2}[g_2, f_2] : \texttt{ch}_2\}$*. First, we add choice transitions to component* B_1 *and synchronize them with the participants of* $\texttt{ch}_1$ *and* $\texttt{ch}_2$*, e.g.,* B_2 *and* B_3*. Then, we apply the choreographies accordingly. Finally, we merge the contexts with internal transitions.*

8.2.3. Loop Composition

Loop $\texttt{while}(snd[g, f])\{\texttt{ch}\}$, allows for the modeling of a conditional repeated choreograph `ch`. The condition is evaluated by a specific component, which will notify, through the port *snd*, the participants of the choreography to either re-execute it or break.

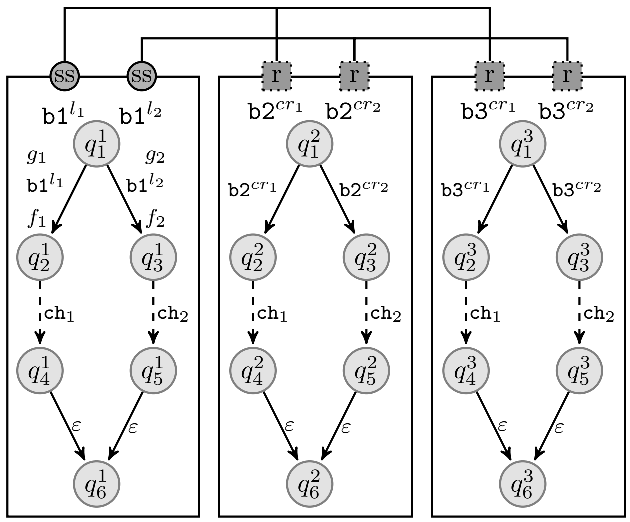

Figure 6: Branching transformation

Definition 17 (Loop).

let $K = \mathcal{C}(\texttt{ch}) \setminus \{i\}$
let $(\gamma^{\texttt{t}}\,(B_1^{\texttt{t}}, \ldots, B_n^{\texttt{t}})\,, \texttt{context}^{\texttt{t}}) = [\![\texttt{ch}]\!][\![snd[g,f] \longrightarrow \{pr_k^{\texttt{cont}}[\emptyset]\}_{k\in K}]\!](S, \texttt{context})$
let $(P_i^{\texttt{t}}, -, L_i^{\texttt{t}}, T_i^{\texttt{t}}) = B_i^{\texttt{t}},$ *for* $i \in [1, n]$
in $[\![\texttt{while}(snd[g,f])\texttt{ch end}]\!](S, \texttt{context}) = (\gamma'\,(B_1', \ldots, B_n')\,, \texttt{context}')$
where:
let $p_j^{\texttt{f}}$ *and* l_j^c *be new synchronous ports and locations, for* $j \in K \cup \{i\}$

- $P_j' = P_j^{\texttt{t}} \cup \begin{cases} \{p_j^{\texttt{f}}\} & \textit{if } j \in K \cup \{i\} \\ \emptyset & \textit{otherwise} \end{cases};$

- $L_j' = L_j^{\texttt{t}} \cup \begin{cases} \{l_j^c\} & \textit{if } j \in K \cup \{i\} \\ \emptyset & \textit{otherwise} \end{cases};$

- $T_j' = T_j^{\texttt{t}} \cup \begin{cases} \{\texttt{context}^{\texttt{t}}(B_j) \xrightarrow{\epsilon} \texttt{context}(B_j), \texttt{context}(B_j) \xrightarrow{p_j^{\texttt{f}}, \texttt{true}, \emptyset} l_j^c\} & \textit{if } j = i \\ \{\texttt{context}^{\texttt{t}}(B_j) \xrightarrow{\epsilon} \texttt{context}(B_j), \texttt{context}(B_j) \xrightarrow{p_j^{\texttt{f}}, \neg g, \emptyset} l_j^c\} & \textit{if } j \in K \setminus \{i\} \\ \emptyset & \textit{otherwise} \end{cases};$

- $\gamma' = \gamma^{\texttt{t}} \cup \{(p_i^{\texttt{f}}, \{p_j^{\texttt{f}}\}_{j\in K})\};$

- $\texttt{context}'(B_j') = \begin{cases} l_j^c & \textit{if } j \in K \cup \{i\} \\ \texttt{context}(B_j) & \textit{otherwise} \end{cases}.$

Transitions are updated by adding the reset and loop transitions. The condition is evaluated by a specific component, which will notify, through the port p_i, the participants of the choreography to either re-execute it or break. The context is updated to be the location associated with the end of the loop.

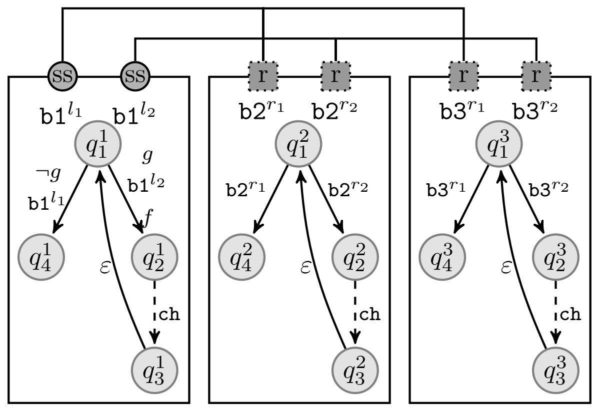

Figure 7: Loop composition transformation

Example 3 (Loop). *Figure 7 shows an example of application of a loop operation guided by component B_1 and where the participants are components B_1, B_2 and B_3.*

8.2.4. Sequential Composition

The binary operator $\bullet$ allows to sequentially compose two choreographies, $\mathtt{ch}_1 \bullet \mathtt{ch}_2$. For this, its semantics is defined by (1) applying $\mathtt{ch}_1$; (2) notifying the start of $\mathtt{ch}_2$; and finally (3) applying $\mathtt{ch}_2$. As we require that $\mathtt{ch}_1$ must terminate before the start of $\mathtt{ch}_2$, we need to synchronize all the end components of $\mathtt{ch}_1$ with all the start components of $\mathtt{ch}_2$. To do so, it is sufficient to pick one of the end components of $\mathtt{ch}_1$ and create a synchronous send port, which is connected to new receive ports added to the remaining end components of $\mathtt{ch}_1$ and start components of $\mathtt{ch}_2$. Moreover, the application of the sequential composition guarantees that each component of the resulting system consists of exactly one state, provided that the context of each component of the initial system consists of one state. Formally, the semantics of the sequential composition is defined as follows.

Definition 18 (Sequential Composition).

$$[\![\mathtt{ch}_1 \bullet \mathtt{ch}_2]\!](S, \mathtt{context}) = [\![\mathtt{ch}_2]\!][\![\mathtt{ch}_{synch}]\!][\![\mathtt{ch}_1]\!](S, \mathtt{context}), \textit{with:}$$

$\mathtt{ch}_{synch} = p_i^{\mathtt{cs}}[\mathtt{true}, \emptyset] \longrightarrow \{p_j^{\mathtt{cr}}[\mathtt{true}, \emptyset]\}_{j \in J}$ *such that: (1)* $i \in \mathtt{end}(\mathtt{ch}_1)$*; (2)* $J = \mathtt{end}(\mathtt{ch}_1) \cup \mathtt{start}(\mathtt{ch}_2) \setminus \{i\}$*; (3)* $p_i^{\mathtt{cs}}$ *is a new synchronous send port to be added to* $\mathcal{P}_i^{ss}$*; and (4)* $\{p_j^{\mathtt{cr}}\}_{j \in J}$ *are new receive ports to be added to* $\mathcal{P}_j^{r}$*.*

Example 4 (Sequential composition). *Figure 8 shows an abstract example on how to transform sequential composition of two choreographies,* $\mathtt{ch}_1 \bullet \mathtt{ch}_2$*, into an initial system consisting of five components. Here we only consider components that are involved in those choreographies, where (1) components* b_1*,* b_2*,* b_3 *and* b_4 *are involved in choreography* $\mathtt{ch}_1$*; and (2) components* b_1*,* b_2*,* b_3

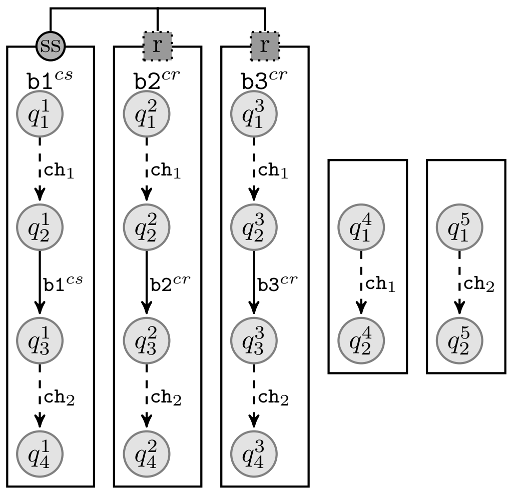

Figure 8: Sequential composition transformation

and b_5 are involved in choreography $\texttt{ch}_2$*. Note, components that are not involved are kept unchanged. The transformation requires to: (1) apply first choreography* $\texttt{ch}_1$ *to its participated components (i.e., b_1, b_2, b_3 and b_4); (2) synchronize the end of choreography* $\texttt{ch}_1$ *(e.g., b_1) with the start of choreography* $\texttt{ch}_2$ *(e.g., b_2 and b_3). To do so, we create a synchronous send port to one of the end components of* $\texttt{ch}_1$ *(e.g., b_1^{cs}) and connect it to all the remaining end components of* $\texttt{ch}_1$ *(e.g., $\emptyset$ and the start components of* $\texttt{ch}_2$ *(e.g., b_2 and b_3); finally (3) we apply choreography* $\texttt{ch}_2$*.*

8.2.5. Parallel Composition

The binary operator $\|$ allows for the parallel compositions of two independent choreographies. Two choreographies are independent if their participating components are disjoint.

Definition 19 (Independent Choreographies). *Two choreographies* $\texttt{ch}_1$ *and* $\texttt{ch}_2$ *are said to be independent iff* $\mathcal{C}(\texttt{ch}_1) \cap \mathcal{C}(\texttt{ch}_2) = \emptyset$*.*

We consider independent choreographies to avoid conflicts and interleaving of executions within components. In addition, this simplifies reasoning and writing choreographies as well as for efficient code generation. Note that parallelizing independent choreographies implies that each component has a single execution flow. In case we have overlap, e.g., $p_1 \longrightarrow \{p_2, p_3\} \;\|\; p_1 \longrightarrow \{p_5\}$, we could split p_1 into two different components. Moreover, it is possible to enforce any arbitrary order of execution. Further, we discuss other possible alternatives for handling this case. This would not reduce the expressiveness of our model as parallel execution flows can be modelled in separate components. The semantics of the parallel composition $\texttt{ch}_1 \| \texttt{ch}_2$ is simply defined by applying $\texttt{ch}_1$ and $\texttt{ch}_2$ in any order, which leads to the same system as the two choreographies are

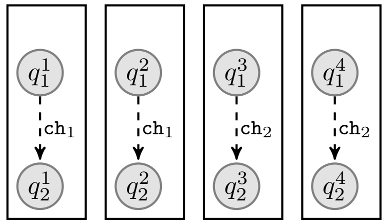

Figure 9: Parallel composition transformation

independent, i.e., they behave on different set of components. Moreover, the application of the parallel composition guarantees that each component of the resulting system consists of exactly one state, provided that the context of each component of the initial system consists of one state.

Definition 20 (Parallel Composition).

$$[\![\texttt{ch}_1 \parallel \texttt{ch}_2]\!](S, \texttt{context}) = [\![\texttt{ch}_2]\!][\![\texttt{ch}_1]\!](S, \texttt{context})$$

Example 5 (Parallel Composition). *Figure 8 shows an abstract example on how to transform parallel composition of two choreographies,* $\texttt{ch}_1 \parallel \texttt{ch}_2$*, into an initial system consisting of five components. Here, we consider that* $\texttt{ch}_1$ *(resp.* $\texttt{ch}_2$*) involves components* B_1 *and* B_2 *(resp.* B_3 *and* B_4*).*

The following proposition is a straightforward consequence of the transformation associated with the $\parallel$ operator and the fact that the transformation of a choreography only modifies the component involved in this choreography.

Proposition 1. *If* $\texttt{ch}_1$ *and* $\texttt{ch}_2$ *are two independent choreographies, then* $[\![\texttt{ch}_1 \parallel \texttt{ch}_2]\!] = [\![\texttt{ch}_2 \parallel \texttt{ch}_1]\!]$.

Consequently, synthesizing distributed systems for parallel choreographies can be done concurrently.

Remark 4. *For parallelizing choreographies that have a component in common (i.e., not independent), we can still apply the parallel composition either by (1) enforcing any arbitrary order of execution. As such, in the case of independent choreographies, true parallelism is achieved, otherwise, we apply them in any order to avoid non-deterministic execution; (2) using of product automata as defined in [36]; (3) use of multiple execution flows (i.e., multi-threading within a component).*

8.3. Discussion on the Correctness of the Synthesis Method

We conjecture that a choreography $\texttt{ch}$ and its corresponding synthesized distributed system obtained by the transformations in this section are weakly bisimilar. Below we give some arguments based on the structure of the choreography. A full proof is left for future work.

- In the case of send/receive choreographies. The execution of choreographies follows rules `(synch-sendrcv)` for synchronous send, `(asynch-sendrcv-1)` and `(asynch-sendrcv-2)` for asynchronous send. The execution of distributed systems follows rule `(synch-send)`. The transformation is implemented by the interaction added in Definition 14; see Figure 5.

- In the case of branching choreographies. The execution of choreographies follows rule `(master-branching)`. The transformation is implemented by Definition 16 where we create the appropriate interactions to implement the master branching rule, as depicted in Figure 6.

- In the case of looping choreographies. The execution of choreographies follows rules `(iterative-tt)` and `(iterative-ff)`. The transformation is implemented by Definition 17 where we create the appropriate interactions and behavior to implement the looping rule, as depicted in Figure 7.

- In the case of sequential choreographies. The execution of choreographies follows rules `(sequential-1)` and `(sequential-2)`. The transformation is implemented by Definition 18 where we add an interaction and behavior to implement the sequential rules and guarantee the sequential execution of the input choreographies, as depicted in Figure 8.

- In the case of parallel choreographies. The execution of choreographies follows rules `(parallel-1)`, `(parallel-2)`, `(parallel-3)`, and `(parallel-4)`. The transformation is implemented by Definition 20 where we transform each choreography independently, as depicted in Figure 9.

9. Code Generation

We describe the principle of how to generate a distributed implementation from the generated components.

Code generation takes as input a choreography and a configuration file containing the list of components with their corresponding interfaces/ports and variables. Clearly, the choreography is defined with respect to the components' ports, with functions and guards defined with respect to the components' variables. We only consider independent choreographies, as described in Definition 19. Note, if the components are not independent, we can follow the strategies described in Remark 4. Code generation then automatically produces the corresponding implementation of each of the components. Following our transformation into Distributed CBS in Section 8.2, the obtained components have the following characteristics: (1) they do not have a location with outgoing send and receive ports; (2) a port is connected to exactly one interaction. As such, there are no conflicting interactions that can run concurrently. Two interactions are said to be conflicting iff they share a common component. Consequently, it is possible to generate fully distributed implementations, with no need for controllers (unlike [7]) for managing multiparty interactions. Hence, the number of exchanged messages will be divided by 2 for each execution of an interaction.

Algorithm 1: Pseudo-code - generated components.

```
initialization();
while true do
    if all outgoing transitions are send then
        port p = select enabled port, i.e., guard true;
        notify all the receivers of the interaction that has port p;
        if p is synchronous then
            wait for ack. from the receivers;
        end
    end
    else if all outgoing transitions are receive then
        wait until a message is ready in one of the outgoing receive ports;
        port p = select message;
        if interaction connected is synchronous then
            send ack. to the corresponding send port;
        end
    updateCurrentState();
end
```

The code structure is depicted in Algorithm 1 that requires only send/receive primitives. After initializing, we distinguish between two possible cases.

Case 1. All outgoing transitions are labeled with send ports.

- We pick a random enabled port, i.e., its guard evaluated to true.
- Then, we notify all the receive ports that are connected to the interaction containing that port.
- If the port is a synchronous send port, the component waits for an acknowledgement from the corresponding receive components.

Case 2. All outgoing transitions are labeled with receive ports.

- The component waits until a message is ready/received in one of the receive ports.
- Upon receiving a message, we acknowledge its receipt if the port is connected to a synchronous interaction.

Finally, we update the current state (update location and execute local function) of the component (`updateCurrentState()`) depending on the current outgoing transition.

It is worth mentioning that it is possible to provide a code generation w.r.t. a communication library (e.g., MPI, Java Message Service). In this case, the code generation can benefit from the features provided by the library, e.g., synchronous communication such as `MPI_Ssend`.

10. Building Micro-Services Using Choreography

Traditionally, distributed applications follow a monolithic architecture, i.e., all the services are embedded within the same application. A new trend is to split complex applications up into smaller micro-services, where each micro-service can live on its own within a container.

We conduct a case study on a micro-service architecture to automatically derive the skeleton of each micro-service. We use choreographies to describe the interactions between services. The system consist of several communicating services to provide clients with system images. Typical services include load balancing, authentication, fault-tolerance, installation, storage, configuration, and deployment. The system also allows clients to request and install packages.

The corresponding global choreography CH is defined in Listing 1.

- CH_1: A client (`c`) sends a request to the *gateway service* (`gs`), which is the only visible micro-service to the client, containing the required version, revision, pool name, and an identifier to the testing data. `gs` forwards the request to the *deploy environment service* (`des`). `des` creates an environment id and returns it back to `gs`, which in turn forwards it back to `c`.

- CH_2: `des` sends to the *deploy application directory service* (`dads`) and the *deploy database service* (`dds`) (i) required version, revision and pool name and (ii) testing data identifier and environment id, respectively. `c` keeps checking if the environment is ready, which is done through the gateway service with the help of the *environment info. service* (`eis`).

- CH_3: `dads` requests from the *machine service* (`ms`) and the *setup service* (`ss`) (i) a machine location from the pool and (ii) the package location, respectively. When `dads` receives the replies from both `ms` and `ss`, it contacts the appropriate *host machine* ($\texttt{hm}_i$) by sending the package location. Then, $\texttt{hm}_i$ sends its status to `des`. `des` upon receiving the status update, it forwards it to the `eis`. `dds` requests from the *dumps service* (`dus`) and the *Database machines services* (`dms`) (i) testing data location, and (ii) a database server, respectively. When `dds` receives the replies from both `dus` and `dbs`, it contacts the appropriate *database server* $\texttt{hd}_j$ by sending the testing data location. Then, $\texttt{hd}_j$ sends its status to `des`. Upon receiving the status update, `des` forwards it to `eis`.

For each micro-service/component `m`, we denote by $\texttt{m}SS$, $\texttt{m}AS$ $\texttt{m}R$ a corresponding synchronous send, asynchronous send and receive port, respectively.

Given the global choreography, we automatically synthesize the code of each component. Note that, in practice, the above choreography may be updated to fulfill new requirements by updating/adding/removing new micro-services. This would require a drastic effort to re-implement the communication logic between components, which is tedious, error-prone and very time-consuming. Using our method, we only require to update the global choreography, and then automatically generate the implementation of the components.

Listing 1: Global choreography

$$
\begin{array}{l}
\mathtt{CH} \;=\; \mathtt{CH}_1 \bullet \mathtt{CH}_2 \bullet \mathtt{CH}_3 \\
\mathtt{CH}_1 = \mathtt{cSS} \longrightarrow \mathtt{gsR} \bullet \mathtt{gsSS} \longrightarrow \mathtt{desR} \bullet \mathtt{desAS} \longrightarrow \mathtt{gsR} \\
\mathtt{CH}_2 = \mathtt{CH}_2^1 \bullet \mathtt{CH}_2^2 \\
\mathtt{CH}_2^1 = \mathtt{gsSS} \longrightarrow \mathtt{cR} \parallel (\mathtt{desAS} \longrightarrow \mathtt{dadsR} \bullet \mathtt{desAS} \longrightarrow \mathtt{dadsR}) \\
\mathtt{CH}_2^2 = \mathtt{while}(\mathtt{cSS})\ \mathtt{cSS} \longrightarrow \mathtt{gsR} \bullet \\
\qquad\qquad\qquad \mathtt{gsSS} \longrightarrow \mathtt{eisR} \bullet \mathtt{eisSS} \longrightarrow \mathtt{gsR} \bullet \mathtt{gsSS} \longrightarrow \mathtt{cR}\ \mathtt{end} \\
\mathtt{CH}_3 = (\mathtt{CH}_4 \parallel \mathtt{CH}_5) \bullet \mathtt{CH}_6 \\
\mathtt{CH}_4 = \mathtt{CH}_4^1 \bullet \mathtt{CH}_4^2 \bullet \mathtt{CH}_4^3 \\
\mathtt{CH}_4^1 = \mathtt{dadsAS} \longrightarrow \mathtt{amsR} \bullet \mathtt{dadsAS} \longrightarrow \mathtt{SSR} \\
\mathtt{CH}_4^2 = \mathtt{amsSS} \longrightarrow \mathtt{dadsR} \parallel \mathtt{ssSS} \longrightarrow \mathtt{dadsR} \\
\mathtt{CH}_4^3 = \mathtt{dads} \oplus \{l_i : \mathtt{dadsSS} \longrightarrow \mathtt{hm_iR} \bullet \mathtt{hm_iSS} \longrightarrow \mathtt{desR}\} \\
\mathtt{CH}_5 = \mathtt{CH}_5^1 \bullet \mathtt{CH}_5^2 \bullet \mathtt{CH}_5^3 \\
\mathtt{CH}_5^1 = \mathtt{ddsAS} \longrightarrow \mathtt{dusR} \bullet \mathtt{ddsAS} \longrightarrow \mathtt{SSR} \\
\mathtt{CH}_5^2 = \mathtt{dusSS} \longrightarrow \mathtt{ddsR} \parallel \mathtt{dmsSS} \longrightarrow \mathtt{dadsR} \\
\mathtt{CH}_5^3 = \mathtt{dds} \oplus \{l_i : \mathtt{ddsSS} \longrightarrow \mathtt{hd_iR} \bullet \mathtt{hd_iSS} \longrightarrow \mathtt{desR}\} \\
\mathtt{CH}_6 = \mathtt{desAS} \longrightarrow \mathtt{eisR}
\end{array}
$$

```
createPromela() {
  createChannels();
  foreach B_i {
    createProcess(i);
  }
}
```

Listing 2: Main Code Generation from System S to `Promela`

11. Transformation to Promela

Overview. Given a system $S = (B, \mathtt{init})$, with $B = \gamma(B_1, \ldots, B_n)$, produced by applying the set of transformations corresponding to a given choreography `ch`, we define a translation of S into `Promela` [21]. The `Promela` version of the system has the same behavior as S but it can be verified with respect to properties specified in Linear Temporal Logic (LTL).

The transformation to `Promela` is realized mainly by two functions (1) `createChannels`, which generates global channels (in Promela) that are used to transfer messages between processes; (2) `createProcess`, which generates the code that corresponds to each of the components. We use the `append` call to add `Promela` code to the generated file. Listing 2 depicts code generation for a system S to `Promela`.

Function `createChannels`. The main skeleton of the `createChannels` is depicted in Listing 3. For every receive port, we create a channel (Promela's

```
createChannels()
  foreach  a ∈ γ, where a = (p_s, {p_r^i}_{i∈I}) {
    foreach  p ∈ {p_r^i}_{i∈I} {
      if(isSSend(p_s))
        append chan channelP = [0] of {ps.dtype};
      else
        append chan channelP = [MAX_LEN] of {ps.dtype};
      end
    end
  end
```

Listing 3: `createChannels` Skeleton

message carrier type). The type of the channel is the data type of the corresponding send port (i.e., p.`dtype`). For synchronous (resp. asynchronous) ports, we use a channel of length 0 (resp. `MAX_LEN`).

Function `createProcess`. The main skeleton of the `createProcess` is depicted in Listing 4. For every component B_i, we create a process in Promela containing: (1) a variable that will hold the current location of the component, which is initialized to the initial location of the component; a (2) the variables of the component; and (3) the code generated of the LTS implementation of the component.

12. Case Study: Synthesizing an Implementation of a Buying System

We consider a system consisting of four components: Buyer 1 (B_1), Buyer 2 (B_2), Seller (S) and Bank (Bk).

12.1. Specification of the Buying System

Buyer 1 sends a book title to the Seller, who replies to both buyers by quoting a price for the given book. Depending on the price, Buyer 1 may try to haggle with Seller for a lower price, in which case Seller may either accept the new price or call off the transaction entirely. At this point, Buyer 2 takes Seller's response and coordinates with Buyer 1 to determine how much each should pay. In case Seller chose to abort, Buyer 2 would also abort. Otherwise, it would keep negotiating with Buyer 1 to determine how much it should pay. Buyer 1, having a limited budget, consults with the bank before replying to Buyer 2. Once Buyer 2 deems the amount to be satisfactory, he will ask the bank to pay the seller the agreed upon amount (Buyer 1 would be doing the same thing *in parallel*).

12.2. Synthesizing the Implementation

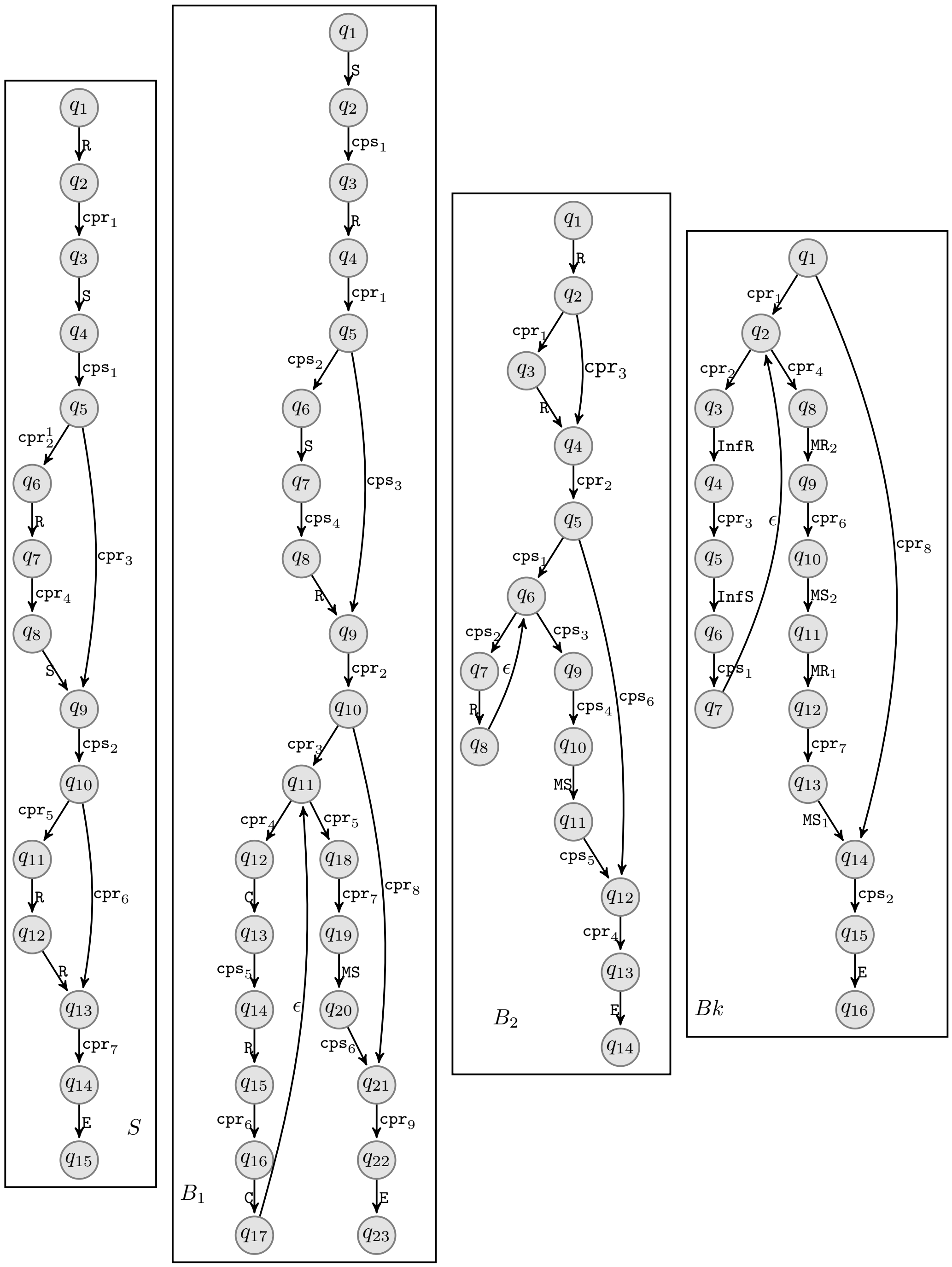

Figure 10: Components generated from the choreography in Listing 5.

```
createProcess(int id) {
  append proctype process(int id) {
    append int currentLocation = initialLocation;
    append currPort = _;
    append do
    append :: if
    append :: (all current outgoing trans. are send) ->
      append p_s = pickEnablePort(); // w.r.t. guard
      append currPort = p_s;
      foreach  p ∈ {p_r^i}_{i∈I}, where ∃a = (p_s, {p_r^i}_{i∈I}) ∈ γ {
        append channelP!(msg);
      }
      append if
      append :: (all outgoing are synchronous send) ->
        foreach  p ∈ {p_r^i}_{i∈I}, where ∃a = (p_s, {p_r^i}_{i∈I}) ∈ γ {
          append channelP?(_);
        }
      append fi;
      append ::  else -> // outgoing transitions are receive
        // listening to all current channels
        append if
         foreach p: currentLocation -p->
            append ::(channelP?(val)) -> currPort = p;
            if(p is connected to synchronous send) {
               append channelP!(ack);
            }
        append fi;
    append fi;
    // Update current location and execute location function
    // of the current outoing transition.
    append updateCurrentState();
  append  od;
  append }
}
```

Listing 4: `createProcess` Skeleton

Choreography. We used the specification of the buying system to write a global choreography `ch` that describes the expected interactions between the buyers and the seller. The choreography is given Listing 5. In the choreography, we prefix the names of the ports by the owning components. Each port maps to a different functionality in the system so that, for example, `Bk.InfR` and `Bk.InfS` represent an interface for handling enquiries. $\texttt{B}_i\texttt{.S}$ and $\texttt{B}_i\texttt{.R}$ represent simple message send/receive interfaces for Buyer i (similarly for `S.S` and `S.R`).

Synthesizing the distributed component-based system. We apply our transformation to the choreography in Listing 5 and obtain the distributed component-based system depicted in Figure 10. The system consists of four components, Listing 5: Global choreography of the Buyer/Seller example

```
CH  = B1.S → S.R • S.S →{B1.R, B2.R} • B1 ⊕{CH1, ϵ} •
    CH2 • CH7
CH1 = B1.S → S.R • S.S →{B1.R, B2.R}
CH2 = B2 ⊕{CH3, nil}
CH3 = while(B2.C) {
        B1.C → Bk.InfR • Bk.InfS → B1.R • B1.C → B2.R
      } • CH4
CH4 = CH5 ∥CH6
CH5 = B2.MS → Bk.MR2 • Bk.MS2 → S.R
CH6 = B1.MS → Bk.MR1 • Bk.MS1 → S.R
CH7 = B1.E →nil ∥ B2.E →nil ∥ Bk.E →nil ∥ S.E →nil
```

```
#define recv(ch) ch?value
#define recvAck(ch) ch?(_)
#define send(ch) ch!value
#define sendAck(ch) ch!ack
#define synchRecv(ch) ch?value; sendAck(ch)
```

Listing 6: Promela Macros

one for each process involved in the choreography. Ports prefixed with `cp` are controlled ports generated for synchronization following the transformations in Section 8. Interactions are used by the components to synchronize and communicate, e.g., (1) $(B_1.S, \{S.R\})$, which allows buyer B_1 to request a quote from the seller; (2) $(B_2.cps_1, \{B_1.cpr_3, Bk.cpr_1, S.cpr_5\})$, which is used to broadcast the choice made by buyer B_2. In total, we generate 27 interactions. Otherwise, the components evolve independently. The components do not require controllers to execute; this ensures the efficiency of the implementation at runtime.

`Promela` version of the implementation. To verify that the distributed implementation respects some desired properties, we apply our transformation of distributed component-based systems to `Promela` which constitutes a translation of the choreography behavior.

Because of the absence of procedures in `Promela`, we define the macros in Listing 6 for convenience and clarity. All of these macros accept a `Promela` channel (`ch`). We assume that `value` is a variable that contains the value that should be sent.

With the macros defined in Listing 6, the `Promela` code generated is depicted in Listing 7.

`updateCurrentState` is a macro that updates the current location and execute the location function of the current outgoing transition. The result of this computation would then be stored in the variable `value`.

12.3. Verifying the Implementation

We verify the generated implementation of the buying system against LTL [33][1] properties specifying its expected behavior. In the following descriptions of properties, we prefix variables local to processes with the the name of the process.

Correct termination. The correct termination property require that "all processes terminate if any of them terminate". Let the ports suffixed by `E` represent the termination interface/port of the corresponding process. Moreover, we consider the following atomic propositions $\texttt{currPort}_1 = \texttt{Buyer1.currPort}$, $\texttt{currPort}_2 = \texttt{Buyer2.currPort}$, $\texttt{currPort}_3 = \texttt{Bank.currPort}$, and $\texttt{currPort}_4 = \texttt{Seller.currPort}$. Then, correct termination can be expressed as the following LTL formula:

$$\mathbf{G}\left(\bigvee_{i=1}^{4}(\texttt{currPort}_i = E_i) \implies \mathbf{F}\bigwedge_{i=1}^{4}(\texttt{currPort}_i = E_i)\right)$$

where $\texttt{E}_\texttt{i}$ represents the ending interface of the appropriate process.

Uniqueness of interface calls. An interface should *only be called once.* In each run, money is only withdrawn once by each process. Let the port $\texttt{Bk.MS}_1$ (resp. $\texttt{Bk.MS}_2$) represent the withdrawal of money by process 1 (resp. process 2). Then, specifying that money is withdrawn once per process can be expressed as the LTL formula:

$$\bigwedge_{i=1}^{2}\mathbf{G}((\texttt{Bank.currPort} = \texttt{Bk.MS}_i) \implies \mathbf{XG}(\neg\texttt{Bank.currPort} = \texttt{Bk.MS}_i))$$

Correct transaction. Money is only withdrawn *after* either Buyer1 or Buyer 2 makes a request. Let the ports $\texttt{Bk.MS}_i$ be as above and let $\texttt{B}_i\texttt{.MS}$ represent money transfer requests by Buyer i. Then specifying the order of execution is represented by the following LTL formula:

$$\bigwedge_{i=1}^{2}\mathbf{G}\big((\neg(\texttt{Bank.currPort} = \texttt{Bk.MS}_i))\ \mathbf{U}\ (\texttt{B}_\texttt{i}\texttt{.currPort} = \texttt{B}_\texttt{i}\texttt{.MS})\big)$$

13. Related Work

Many coordination models exist to simplify the modeling of interactions in concurrent and distributed systems, such as in [1, 5]. Using these models requires the definition of the local behaviors of the processes and use of the communication model to implement the interactions between them. This is in

[1] We recall the intuitive meaning of LTL operators: $\mathbf{G}\varphi$ (resp. $\mathbf{F}\varphi$, $\mathbf{X}\varphi$) stands for globally (resp. eventually, next) φ, and $\varphi_1\mathbf{U}\varphi_2$ stands for φ_1 until φ_2.

contrast to our case where we automatically synthesize the local code of the processes.

Moreover, in order to reason about the correctness of coordinated processes, session types [6, 22, 8, 37, 18, 11] and choreographies [36] have been proposed to statically verify the implementations of communication protocols based on the following methodology: (1) define communication protocol between processes using a *global protocol*; (2) automatically synthesize *local types* which are the projection of global protocol w.r.t. processes; (3) develop the code of processes; (4) statically type-check the code of the processes w.r.t. local types. Consequently, the distributed software follows the stipulated global protocol. In our case, we automatically generate a more refined version of processes that embeds all the communication and synchronization logic as well as control flows, and which is (conjectured to be) correct-by-construction with respect to the global choreography.

In [9], the authors present a deadlock-freedom by design method for choreographies communicating using multiparty asynchronous interactions. The method allows to efficiently verify and reason at the choreography level. Although, (1) the method is not concerned about synthesizing distributed implementation; and (2) the communication model only supports asynchronous interactions; using this approach can help us to verify and reason about our choreographies. Moreover, we can use a similar approach introduced in [35] to efficiently verify our choreographies.

In [10], the notion of Linear Compositional Choreographies (LCC) is presented. In LCC, choreographies and processes can be combined, so that, for example, a choreography can be combined with existing process code (e.g., from a software library) to produce a new choreography. LCC is a genrealization of intuitionistic linear logic, and proof transformations in LCC yield procedures of endpoint projection and also of choreography extraction (using the standard Curry-Howard interpretation of proofs-as-program). It is also shown that all internal communications can be reduced, so that LCC programs are deadlock-free by construction. In [34], the authors present a notion of choreography that permits dynamic updates at run time. These can be compiled into distributed programs in the Jolie programming language. In [3] choreographies are implemented by the automatic synthesis of distributed Coordination Delegates (CDs), which are extra processes added to the basic participant services, and which enforce the choreography specification.

In [25, 26], the authors present a method to synthesize a global choreography from a set of local types. The global view allows for the reasoning and analysis of distributed systems. In our approach, we consider the inverse of that transformation, i.e., we create a template with all the necessary communication and control flows of the endpoint processes starting from a global choreography.

In [2, 16], the authors introduce syntactic transformations to refine distributed system programs starting from high-level specifications. In [2], the proposed specification differs from our choreography model as it is not possible to express multiparty interactions, or guarded loop, which makes it impractical in the context of distributed systems. In [16], the paper mainly targets multiparty interactions, where the main objective is to loosening synchronous multiparty interaction while preserving its semantics. In our case, as we automatically synthesize code for multiply interactions, there is no need for loosening technique. Add to that, we also support asynchronous ports that allow to loosening interactions. Additionally, in [2, 16], it is not clear how to automatically generate code from the refined programs.

BPMN [31] (Business Process Model and Notation) is an industry standard that allows modeling process choreographies. An extension of BPMN was introduced in [20, 28] to automatically derive a local choreography from a global one. Nonetheless, the extension only considers exchange of messages and does not formally define other composition operators such as synchronous multiparty communications, parallelism, choice, sequential and loop. The method proposed in [30] allows deriving RESTful choreographies from process choreographies, whereas in this paper we synthesize the code of the processes given global choreography. Moreover, the model is restricted to RESTful architecture. In [19], the authors introduce a framework for the verification and design of choreographies, however, the communication model only allows for one send and one receive per interaction.

14. Conclusion and Future Work

Conclusion. This paper deals with the synthesis of distributed implementations of local processes (control flows, synchronization, notification, acknowledgment, computations embedding), starting from a global choreography. The method presented in this paper allows one to automatically verify the communication protocols and drastically simplify the synthesis of the distributed implementation. Moreover, the language is used to model a real case study provided by Murex S.A.L. services industry. We used the choreography language and the method to synthesize actual micro-services architectures. The synthesized micro-services can be verified against any Linear Temporal Logic formula thanks to a translation to Promela. We illustrated the translation and the verification on a simplified version of an application at Murex for which we synthesized the micro-service implementation.

Future work. In addition to formally prove the weak bisimilarity between choreographies and the synthesized distributed systems (sketched in Section 8.3), future work comprises several directions. First, we consider augmenting our choreography model by adding fault-tolerance primitives. That is, we aim to specify the number of replicas of each process and automatically embed a consensus protocol between them such as Paxos [24] or Raft [32]. Second, we consider integrating our framework with Spring Boot to allow for the automatic generation of RESTful web services starting from global choreography. Third, we consider augmenting our code generation with features provided by *Istio* [23] and *Linkerd* [27], which are used for routing, failure handling, service discovery, the integration of micro-services, the traffic-flow management and enforcing policies. Fourth, we consider defining a specific model checker for our distributed component-based framework. Finally, we consider using complementary verification techniques operating at runtime such as runtime verification [4, 15] and runtime enforcement [12] for which we defined approaches in the case of non-distributed component-based systems [14, 13].

Acknowledgment

The authors warmly thank the reviewers for their helpful comments on a preliminary version of this paper. The work presented in this paper is supported by the Murex Grant Award 103456 and University Research Board at the American University of Beirut.

References

[1] Agha, G. A., Kim, W., 1999. Actors: A unifying model for parallel and distributed computing. Journal of Systems Architecture 45 (15), 1263–1277. URL `https://doi.org/10.1016/S1383-7621(98)00067-8`

[2] Attie, P. C., Das, C., 1997. Automating the refinement of specifications for distributed systems via syntactic transformations. Int. J. Systems Science 28 (11), 1129–1144.

[3] Autili, M., Inverardi, P., Tivoli, M., August 2018. Choreography realizability enforcement through the automatic synthesis of distributed coordination delegates. Science of computer programming, 3–29.

[4] Bartocci, E., Falcone, Y. (Eds.), 2018. Lectures on Runtime Verification - Introductory and Advanced Topics. Vol. 10457 of Lecture Notes in Computer Science. Springer. URL `https://doi.org/10.1007/978-3-319-75632-5`

[5] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J., 2011. Rigorous component-based system design using the BIP framework. IEEE Software 28 (3), 41–48.

[6] Bejleri, A., Yoshida, N., 2009. Synchronous multiparty session types. Electr. Notes Theor. Comput. Sci. 241, 3–33.

[7] Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J., 2012. A framework for automated distributed implementation of component-based models. Distributed Computing 25 (5), 383–409.

[8] Bonelli, E., Compagnoni, A. B., 2007. Multipoint session types for a distributed calculus. In: Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers. pp. 240–256.

[9] Carbone, M., Montesi, F., 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 263–274.
URL `https://doi.org/10.1145/2429069.2429101`

[10] Carbone, M., Montesi, F., Schürmann, C., Feb. 2018. Choreographies, logically. Distrib. Comput. 31 (1), 51–67.
URL `https://doi.org/10.1007/s00446-017-0295-1`

[11] Charalambides, M., Dinges, P., Agha, G. A., 2016. Parameterized, concurrent session types for asynchronous multi-actor interactions. Sci. Comput. Program. 115-116, 100–126.

[12] Falcone, Y., 2010. You should better enforce than verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G. J., Rosu, G., Sokolsky, O., Tillmann, N. (Eds.), Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Vol. 6418 of Lecture Notes in Computer Science. Springer, pp. 89–105.
URL `https://doi.org/10.1007/978-3-642-16612-9_9`

[13] Falcone, Y., Jaber, M., 2017. Fully automated runtime enforcement of component-based systems with formal and sound recovery. STTT 19 (3), 341–365.
URL `https://doi.org/10.1007/s10009-016-0413-6`

[14] Falcone, Y., Jaber, M., Nguyen, T., Bozga, M., Bensalem, S., 2015. Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Software and System Modeling 14 (1), 173–199.
URL `https://doi.org/10.1007/s10270-013-0323-y`

[15] Falcone, Y., Krstic, S., Reger, G., Traytel, D., 2018. A taxonomy for classifying runtime verification tools. In: Colombo, C., Leucker, M. (Eds.), Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings. Vol. 11237 of Lecture Notes in Computer Science. Springer, pp. 241–262.

[16] Francez, N., Forman, I. R., 1991. Synchrony loosening transformations for interacting processes. In: CONCUR '91, 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 26-29, 1991, Proceedings. pp. 203–219.

[17] Francez, N., Forman, I. R., 2018. From global choreography to efficient distributed implementation. In: HPCS - 4PAD International Symposium on Formal Approaches to Parallel and Distributed Systems.

[18] Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N., Caldeira, A. Z., 2010. Modular session types for distributed object-oriented programming. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 299–312.

[19] Güdemann, M., Poizat, P., Salaün, G., Ye, L., 2016. Verchor: A framework for the design and verification of choreographies. IEEE Trans. Services Computing 9 (4), 647–660.
URL https://doi.org/10.1109/TSC.2015.2413401

[20] Hofreiter, B., Huemer, C., 2008. A model-driven top-down approach to inter-organizational systems: From global choreography models to executable BPEL. In: 10th IEEE International Conference on E-Commerce Technology (CEC 2008) / 5th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (EEE 2008), July 21-14, 2008, Washington, DC, USA. pp. 136–145.
URL https://doi.org/10.1109/CECandEEE.2008.129

[21] Holzmann, G. J., 1997. The model checker SPIN. IEEE Trans. Software Eng. 23 (5), 279–295.
URL https://doi.org/10.1109/32.588521

[22] Honda, K., Yoshida, N., Carbone, M., 2008. Multiparty asynchronous session types. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 273–284.

[23] Istio, . https://github.com/istio/istio/.

[24] Lamport, L., December 2001. Paxos made simple, 51–58.
URL https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[25] Lange, J., Tuosto, E., 2012. Synthesising choreographies from local session types. In: CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings. pp. 225–239.

[26] Lange, J., Tuosto, E., Yoshida, N., 2015. From communicating machines to graphical choreographies. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 221–232.

[27] Linkerd, . https://linkerd.io.

[28] Meyer, A., Pufahl, L., Batoulis, K., Fahland, D., Weske, M., 2015. Automating data exchange in process choreographies. Inf. Syst. 53, 296–329.
URL https://doi.org/10.1016/j.is.2015.03.008

[29] Murex, . https://www.murex.com.

[30] Nikaj, A., Weske, M., Mendling, J., 2019. Semi-automatic derivation of restful choreographies from business process choreographies. Software and System Modeling 18 (2), 1195–1208.
URL https://doi.org/10.1007/s10270-017-0653-2

[31] OMG, B. P. M., Notation (BPMN), V. ., 2011. http://www.omg.org/spec/BPMN/2.0/.

[32] Ongaro, D., Ousterhout, J. K., 2014. In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014. pp. 305–319.

[33] Pnueli, A., 1977. The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. IEEE Computer Society, pp. 46–57.

[34] Preda, M. D., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J., Apr. 2017. Dynamic choreographies: Theory and implementation. Logical Methods in Computer Science Volume 13, Issue 2.
URL https://lmcs.episciences.org/3263

[35] Scalas, A., Yoshida, N., 2019. Less is more: multiparty session types revisited. PACMPL 3 (POPL), 30:1–30:29.
URL https://doi.org/10.1145/3290343

[36] Tuosto, E., Guanciale, R., 2018. Semantics of global view of choreographies. J. Log. Algebr. Meth. Program. 95, 17–40.
URL https://doi.org/10.1016/j.jlamp.2017.11.002

[37] Vallecillo, A., Vasconcelos, V. T., Ravara, A., 2006. Typing the behavior of software components using session types. Fundam. Inform. 73 (4), 583–598.

```
proctype Seller() {
  int currentLocation = q1;
  currPort = _;
  int value;
  do
  ::if
    ::(currentLocation == q1) -> synchRecv(S.R); currPort = S.R;
        currentLocation = q2;
    ::(currentLocation == q2) -> synchRecv(S.cpr1); currPort =
        S.cpr1; q3;
    ::(currentLocation == q3) -> send(B1.R); send(B2.R);
        recvAck(B1.R); recvAck(B2.R); currPort = S.S;
        currentLocation = q4;
    ::(currentLocation == q4) -> send(B1.cpr1); recvAck(B1.cpr1);
        currPort = S.cps1 currentLocation = q5;
    ::(currentLocation == q5) ->
      if
      ::recv(S.cpr2) -> sendAck(S.cpr2); currPort = S.cpr2;
          currentLocation = q6;
      ::recv(S.cpr3) -> sendAck(S.cpr3); currPort = S.cpr3;
          currentLocation = q9;
      fi;
    ::(currentLocation == q6) -> synchRecv(S.R); currPort = S.R;
        currentLocation = q7;
    ::(currentLocation == q7) -> synchRecv(S.cpr4); currPort =
        S.cpr4; currentLocation = q8;
    ::(currentLocation == q8) -> send(B1.R); send(B2.R);
        recvAck(B1.R); recvAck(B2.R); currPort = S.S;
        currentLocation = q9;
    ::(currentLocation == q9) -> send(B2.cpr2); recvAck(B2.cpr2);
        currPort = S.cps2; currentLocation = q10;
    ::(currentLocation == q10) ->
      if
      ::recv(S.cpr5) -> sendAck(S.cpr5); currPort = S.cpr5;
          currentLocation = q11
      ::recv(S.cpr6) -> sendAck(S.cpr5); currPort = S.cpr6;
          currentLocation = q14
      fi;
    ::(currentLocation == q11) -> synchRecv(S.R); currPort = S.R;
        currentLocation = q12;
    ::(currentLocation == q12) -> synchRecv(S.R); currPort = S.R;
        currentLocation = q13;
    ::(currentLocation == q13) -> synchRecv(S.cpr7); currPort =
        S.cpr7; currentLocation = q14;
    ::(currentLocation == q14) -> currPort = S.E; currentLocation =
        end;
    ::(currentLocation == end) -> break;
    fi;
    updateCurrentState();
  od;
}
```

Listing 7: Seller Process in Promela