



HAL
open science

From global choreographies to verifiable efficient distributed implementations

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

► **To cite this version:**

Mohamad Jaber, Yliès Falcone, Paul Attie, Al-Abbass Khalil, Rayan Hallal, et al.. From global choreographies to verifiable efficient distributed implementations. *Journal of Logical and Algebraic Methods in Programming*, 2020, 115, pp.1-24. 10.1016/j.jlamp.2020.100577 . hal-03113398

HAL Id: hal-03113398

<https://inria.hal.science/hal-03113398>

Submitted on 18 Jan 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

2 Developing correct distributed software is notoriously difficult. This is mainly
3 due to their complex structure that consists of interactions between distributed
4 processes. We mainly distinguish two possible directions to cope with the com-
5 plexity of the interaction model: (1) high-level modeling frameworks [7]; and
6 (2) session types [6, 22, 8, 37, 18, 11]. The former facilitates expressing the
7 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` (Al-Abbass Khalil), `rah74@aub.edu.lb` (Rayan Hallal),
`antoine.el-hokayem@univ-grenoble-alpes.fr` (Antoine El-Hokayem)

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

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

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

32 *Contributions.* In this paper, we introduce a new framework which allows the
33 automatic synthesis of the local code of the processes starting from a global
34 choreography. First, inspired from the Behavior Interaction Priority framework
35 (BIP) [5], we consider a set of components/processes with their interfaces and
36 a configuration file that defines the variables of each component as well as the
37 mapping between ports and their computation blocks. Then, given a global
38 choreography, which is defined on the set of ports of the components and which
39 models coordination and composition operators, we automatically synthesize
40 the local code of the processes, which embeds all communication and control
41 flow logic. The choreography allows us to define: (1) multiparty interaction;
42 (2) branching; (3) loop; (4) sequential composition; and (5) parallel composi-
43 tion. Without loss of generality, as in most distributed system applications,
44 we consider master-based protocols. In master-based protocols, each interac-
45 tion has a master component deciding whether it can take place and what are
46 the components involved in the interaction. This allows for the generation of
47 fully distributed implementations, i.e., without the need of controllers, hence
48 reducing the need for communication at runtime. Moreover, we discuss some
49 correctness arguments about the behavior of the synthesized implementations

50 following the semantics of choreographies. Furthermore, we define a transla-
51 tion of the distributed implementations to equivalent `Promela` versions. Such a
52 translation allows us to verify user-defined properties on the implementations.
53 We use the SPIN model-checker to verify properties. Our transformations are
54 implemented in a Java tool that we applied to automatically synthesize micro-
55 service architectures starting from global protocols.

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

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

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

88 2. Notation

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

92 $\{x \in \mathbb{N} \mid x \geq a \wedge x \leq b\}$. A sequence of elements over a set E of length $n \in \mathbb{N}$ is
93 formally defined as a (total) function from $[1, n]$ to E . The empty sequence over
94 E (function from \emptyset to E) is denoted by ϵ_E (or ϵ when clear from the context).
95 The length of a sequence s is denoted by $|s|$. The set of (finite) sequences over
96 E is denoted by E^* . The (usual) concatenation of a sequence s to a sequence
97 s' is the sequence denoted by $s \cdot s'$. Given two sets E and F , we denote by
98 $[E \rightarrow F]$ the set of functions from E to F . Given some function $f \in [E \rightarrow F]$
99 and an element $e \in E$, we denote by $f(e)$ the element in F associated with e
100 according to f .

101 3. Preliminary Notions

102 To later construct a system, we assume an architecture with n components
103 $\{B_i\}_{i=1}^n$, with $n \in \mathbb{N}^+$. At this stage, components are just interfaces with
104 ports for communication. To each port of a component is attached a (unique)
105 variable. In this section, we define these notions common to choreographies and
106 component-based systems, later defined in Section 5 and Section 7 respectively.

107 *Types, variables, expressions, and functions.* We use a set of data types, $DataTypes$,
108 including the set of usual types found in programming languages $\{\mathbf{int}, \mathbf{str}, \mathbf{bool}, \dots\}$
109 and a set of (typed) variables $Vars$. Variables are partitioned over components,
110 i.e., $Vars = \bigcup_{i=1}^n Vars_i$ and $\forall i, j \in [1, n] : i \neq j \implies Vars_i \cap Vars_j = \emptyset$. Vari-
111 ables take values in a general data domain $Data$ containing all values associated
112 with the types in $DataTypes$ plus a neutral communication element denoted
113 by \perp_d . We call any function with codomain $Data$ a valuation. Moreover, for
114 two valuations v and v' , v'/v denotes the valuation where values in v' have
115 priority over those in v . For a set of variables $X \subseteq Vars$, we denote by $\mathcal{G}(X)$
116 (resp. $Expr(X)$) the set of boolean (resp. all, i.e., boolean and arithmetic)
117 expressions over X , constructed in the usual manner. Expressions can be used
118 as function descriptions, and, for an expression $e \in Expr(X)$ and a valuation
119 $v \in [X \rightarrow Data]$, we note $e(v)$ the value in $Data$ of expression e according to v .

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

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

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

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

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

136 Given a port p , we define the predicate $isSSend(p)$ (resp., $isASend$, $isRecv$,
 137 $isInternal$) that holds true iff (the communication type of) p is a synchronous
 138 send (resp., asynchronous send, receive, internal) port, i.e., iff $p.ctype = ss$
 139 (resp. as, r, in).

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

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

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

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

157 4. Illustrating Example

158 To illustrate our approach, we consider a toy example of a variant of producer-
 159 consumer. The example begins by modeling producer-consumer using chore-
 160 ographies (described along with their semantics in Section 5). Then, we show the
 161 corresponding component-based distributed implementation (detailed in Sec-
 162 tion 7) which is synthesized from the choreographies using transformations de-
 163 scribed 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 asyn-
 chronously 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 mes-
 sage and decrements the value of n . After P has finished (\bullet) sending (\rightarrow), C

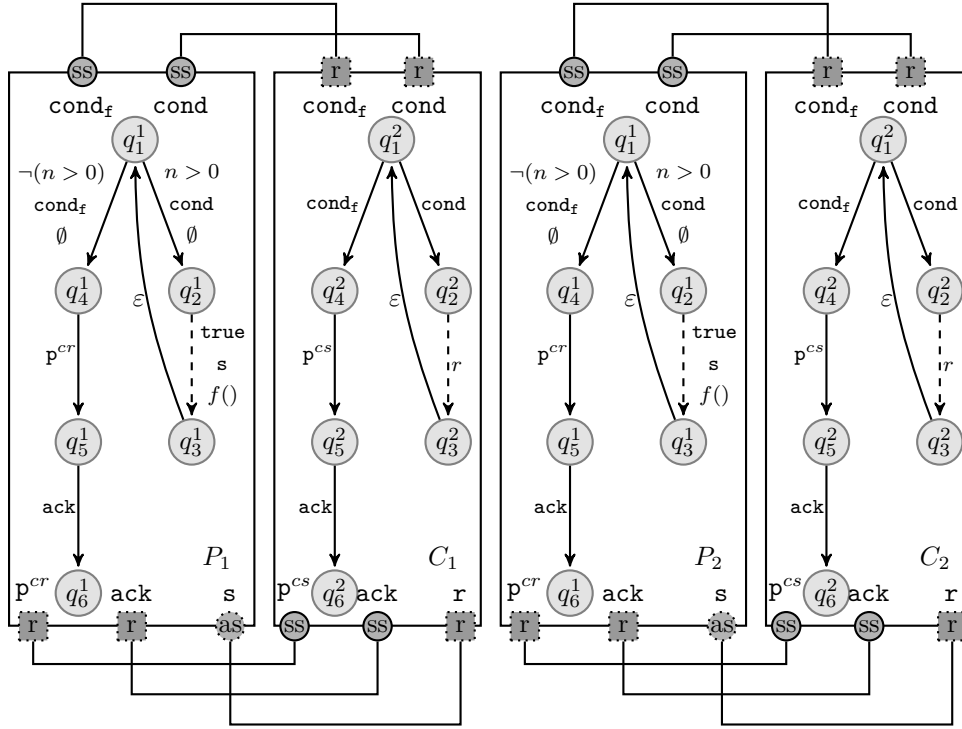


Figure 1: A toy example of a variant of producer-consumer.

sends an acknowledgment message to P. We consider two instances of producers (resp. consumers) P_1 and P_2 (resp. C_1 and 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.

$$\begin{aligned}
 & (\text{while}(P_1.\text{cond}[n > 0])\{P_1.s[\text{true}, f()]\} \rightarrow \{C_1.r[\emptyset]\} \bullet C_1.\text{ack} \rightarrow \{P_1.\text{ack}\}) \\
 \parallel & (\text{while}(P_2.\text{cond}[n > 0])\{P_2.s[\text{true}, f()]\} \rightarrow \{C_2.r[\emptyset]\} \bullet C_2.\text{ack} \rightarrow \{P_2.\text{ack}\})
 \end{aligned}$$

164 *Synthesized distributed system.* The corresponding distributed component-based
 165 model is depicted in Figure 1. The system is composed of four components.
 166 Component P_1 has three basic interfaces `ack` (for receive), `s` (asynchronous
 167 send) and `cond` (synchronous cond). Two other interfaces are generated for
 168 control: `condf` and `pcr`. Condition `condf` is enabled when the condition of the
 169 while does not hold. `pcr` is used to implement the sequential primitive (\bullet). The
 170 two parallel choreographies are independent and correspond of the parallel ex-
 171 ecution of P_1 with C_1 and P_2 with C_2 . As can be noticed, there is no need of
 172 controllers and one can use a process or thread for each component.

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

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

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

176 5. Global Choreography

177 In this section, we define the global choreography model. Recall that compo-
178 nents are seen as interfaces and a choreography serves the purpose of coordinat-
179 ing the communications and computations of components. In choreographies,
180 ports are used with guards and update functions.

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

182 *Syntax of choreographies.* We introduce the abstract syntax of the global chore-
183 ography model.

184 **Definition 4 (Abstract syntax of the choreography model).** *The abstract*
185 *grammar in Figure 2 defines the syntax of the choreography model. We denote*
186 *by \mathcal{Chors} the set of choreographies defined by this grammar.*

187 The definition of choreographies relies on the previously defined concepts such
188 as update functions in $\mathcal{F}(X)$, guards in $\mathcal{G}(X)$, the existing types in *DataTypes*,
189 available components in $\{B_1, \dots, B_n\}$, and the various types of ports (syn-
190 chronous and asynchronous send ports in \mathcal{P}^{ss} and \mathcal{P}^{as} and receive ports in \mathcal{P}^r).
191 It also relies on the definitions of send port augmented with guard and update
192 function and lists of receive ports and continuations. A send port augmented
193 with guard and update function is of the form $psas[g, f]$ where $psas$ is a syn-
194 chronous or asynchronous send port, g a guard, and f an update function. In
195 a list of receive ports, each element is of the form $pr[g]$ where pr is a receive
196 port identifier and g a guard. In a list of continuations, each element is of the
197 form $psas:ch$ where $psas$ is a synchronous or asynchronous send port and ch is

198 a choreography. We extend the dot notation to choreographies and, for a send
 199 or receive port augmented with guard and update function, i.e., of the form
 200 $psas[g, f]$ or $pr[g]$, we note $psas.guard$ and $pr.guard$ for g and $psas.ufct$ for f .

201 Base choreographies include the empty choreography (`nil`) and the send/re-
 202 ceive communication primitive. Send/receive communications are of the form
 203 $snd \rightarrow \{rcv_list\} : \langle t \rangle$ where snd is a (synchronous or asynchronous) send
 204 port, rcv_list is a list of receive ports and $: \langle t \rangle$ is a type annotation with
 205 $t \in DataTypes$.

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

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

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

- 221 • In a synchronous/asynchronous send port with guard and update func-
 222 tion $psas[g, f]$, the variables used in the guard g should belong to the
 223 component of port $psas$.
- 224 • In a conditional master branching, the send ports in the continuation list
 225 should belong to the component.

226 *Semantics of choreographies.* In the following, we consider well-typed chore-
 227 ographies built with the syntax in Definition 4. We define the (structural op-
 228 erational) semantics of choreographies. For this, we consider that states of a
 229 choreography are valuations of the component variables in $[X \rightarrow Data]$. Re-
 230 call that variables and ports are partitioned over components. We denote by
 231 $ChorState$ the set of choreography states.

232 Before actually defining the semantics, we need to model the effect of com-
 233 munication on the choreography state. We model the sending through a port to
 234 a set of ports with a function $send : ChorState \times (\mathcal{P}^{as} \cup \mathcal{P}^s) \times 2^{\mathcal{P}^r} \rightarrow ChorState$
 235 that takes as input a choreography state and outputs a choreography state when
 236 a communication occurs from the (synchronous or asynchronous) send port of
 237 a component to the receive ports of some components: $send(\sigma, snd, \{rcv_list\})$
 238 is state σ where the value of variable of port snd is used to update the vari-
 239 ables attached to ports in $\{rcv_list\}$. Formally: $send(\sigma, snd, \{rcv_list\}) =$

$$\begin{array}{c}
\frac{}{(\mathbf{nil}, \sigma) \xrightarrow{\tau} \sigma} \text{ (nil)} \\
\frac{snd \in \mathcal{P}^{ss} \quad \sigma \models g \quad rcv_list = pr_1[f_1], \dots, pr_k[f_k]}{(snd[g, f] \longrightarrow \{rcv_list\}, \sigma) \xrightarrow{\{snd, pr_1, \dots, pr_k\}} f \circ f_k \circ \dots \circ f_1 \circ \text{send}(\sigma, snd, \{pr_1, \dots, pr_k\})} \text{ (synch-sendrcv)} \\
\frac{snd \in \mathcal{P}^{as} \quad \sigma \models g}{(snd[g, f] \longrightarrow \{rcv_list\}, \sigma) \xrightarrow{\{snd\}} (\{rcv_list\}, f \circ \text{send}(\sigma, snd, rcv_list))} \text{ (asynch-sendrcv-1)} \\
\frac{pr[f] \in \{rcv_list\}}{(\{rcv_list\}, \sigma) \xrightarrow{\{pr\}} (\{rcv_list\} \setminus \{pr[f]\}, f(\sigma))} \text{ (asynch-sendrcv-2)} \\
\frac{\sigma \models g_j}{(B \oplus \{snd_1[g_1, f_1] : ch_1, \dots, snd_k[g_k, f_k] : ch_k\}, \sigma) \xrightarrow{\{snd_j\}} (ch_j, f_j(\sigma))} \text{ (master-branching)} \\
\frac{\sigma \models g}{(\mathbf{while}(snd[g, f]) \mathbf{ch\ end}, \sigma) \xrightarrow{\{snd\}} (ch \bullet \mathbf{while}(snd[g, f]) \mathbf{ch\ end}, f(\sigma))} \text{ (iterative-tt)} \\
\frac{\sigma \not\models g}{(\mathbf{while}(snd[g, f]) \mathbf{ch\ end}, \sigma) \xrightarrow{\tau} \sigma} \text{ (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')} \text{ (sequential-1)} \quad \frac{(ch_1, \sigma) \xrightarrow{l_1} \sigma'}{(ch_1 \bullet ch_2, \sigma) \xrightarrow{l_1} (ch_2, \sigma')} \text{ (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)} \text{ (parallel-1)} \quad \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)} \text{ (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)} \text{ (parallel-3)} \quad \frac{(ch_2, \sigma_2) \xrightarrow{l_2} \sigma'_2}{(ch_1 \parallel ch_2, \sigma_2) \xrightarrow{l_2} (ch_1, \sigma'_2)} \text{ (parallel-4)}
\end{array}$$

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

240 $\sigma[\{rcv_list\}.\mathbf{var} \mapsto \sigma(snd.\mathbf{var})]$, it is state σ where we apply the substitution
241 that assigns all the variables in $\{rcv_list\}.\mathbf{var}$ to $\sigma(snd.\mathbf{var})$.

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

248 We are now able to define the semantics of choreographies.

249 **Definition 5 (Semantics of choreography model).** *The semantics of chore-*
 250 *ographies is an LTS $(ChorConf, ChorLab, \Rightarrow)$ where :*

- 251 • $ChorConf \subseteq (Chors \times ChorState) \cup ChorState \cup 2^{\mathcal{P}^r \times \mathcal{F}(X)}$ is the set of
 252 configurations and $ChorState \subseteq ChorConf$ is the set of final configura-
 253 tions;
- 254 • $ChorLab \subseteq (2^{\mathcal{P}} \setminus \{\emptyset\}) \cup \{\tau\}$ is the set of labels where each label is either
 255 a set of ports or label τ for silent transitions;
- 256 • $\Rightarrow \subseteq ChorConf \times ChorLab \times ChorConf$ is the least set of (labelled) tran-
 257 sitions satisfying the rules in Figure 3;

258 Whenever for two configurations $c, c' \in ChorConf$ and a label $l \in ChorLab$,
 259 $(c, l, c') \in \Rightarrow$, we note it $c \xRightarrow{l} c'$. The rules in Figure 3 can be intuitively under-
 260 stood as follows:

- 261 • Rule (**nil**) states that choreography **nil** terminates in any state σ and
 262 produces the terminal configuration σ .
- 263 • Rule (**synch-sendrcv**) describes the synchronous send/receive primitive.
 264 The component of port snd transfers data to the components with the
 265 receive ports in rcv_list whenever the guard g attached to snd holds
 266 true from the starting state σ . If the list of receive ports (with update
 267 functions) is $pr_1[f_1], \dots, pr_k[f_k]$, the choreography terminates in a state
 268 obtained after the data transfer defined by $send(\sigma, snd, \{pr_1, \dots, pr_k\})$
 269 and the applications of the update functions f, f_1, \dots, f_k of the send and
 270 receive ports. Note that the application order does not influence the re-
 271 sulting state as these update functions apply to disjoint variables.
- 272 • Rule (**asynch-sendrcv-1**) describes the first part of an asynchronous send/re-
 273 ceive primitive. As in the synchronous send/receive primitive, the compo-
 274 nent of port snd transfers data to the components with the receive ports
 275 in rcv_list whenever the guard g attached to snd holds true from the
 276 starting state σ . However, the state of the receiving component is only
 277 updated with the transferred data (with $send(\sigma, snd, \{pr_1, \dots, pr_k\})$) and
 278 the receiving components do not apply their update functions.
- 279 • Rule (**asynch-sendrcv-2**) describes the second part of an asynchronous
 280 send/receive primitive. A receive port $pr[f]$ in the list of receive ports
 281 to be executed rcv_list applies the attached updated function f to the
 282 current state and is removed from the list of received ports to be executed.
- 283 • Rule (**master-branching**) describes the (conditional) master branching from
 284 component B on one of its continuations $snd_j[g_j, f_j] : ch_j$ whenever the
 285 guard g_j attached to port snd_j holds true. The resulting configuration
 286 consists of the choreography ch_j and the state $f_j(\sigma)$ (resulting from the
 287 application of the attached update function f_j to σ).

- 288 • Rule (**iterative-tt**) describes the first case of the iterative composition of
 289 a choreography ch under the condition $snd[g, f]$ (which consists of a send
 290 port snd , a guard g , and an update function f). When g holds true in
 291 σ , the resulting configuration consists of the choreography ch sequentially
 292 composed with the same starting choreography to be executed in state σ
 293 updated by f .
- 294 • Rule (**iterative-ff**) describes the second case of the iterative composition
 295 of a choreography ch under the condition $snd[g, f]$. When g holds false in
 296 σ , the choreography terminates in the (unmodified) state σ .
- 297 • Rules (**sequential-1**) and (**sequential-2**) describe the possible evolu-
 298 tions of two sequentially composed choreographies ch_1 and ch_2 . Rule
 299 (**sequential-1**) describes the case where the execution of choreography
 300 ch_1 does not terminate and evolves to a configuration (ch_1, σ'_1) which
 301 leads to the global configuration $(ch'_1 \bullet ch_2, \sigma'_1)$. Rule (**sequential-2**) de-
 302 scribes the case where the execution of choreography ch_1 terminates and
 303 evolves to a final configuration σ'_1 which leads to the global configuration
 304 (ch_2, σ'_1) (where the second choreography ch_2 is to be executed in state
 305 σ'_1).
- 306 • Rules (**parallel-1**) to (**parallel-4**) describe the possible evolutions of two
 307 choreographies ch_1 and ch_2 composed in parallel. Rules (**parallel-1**) and
 308 (**parallel-2**) describe the evolutions where ch_1 performs a computation
 309 step and terminates or not. Rules (**parallel-3**) and (**parallel-4**) describe
 310 the evolutions where ch_2 performs a computation step.

311 6. Example: Two-Phase Commit

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

321 *Components.* We model the following protocol using global choreographies (Sec-
 322 tion 5). In our setting, we have n workers and 1 coordinator.

323 For each worker $i \in [1..n]$ we associate a worker component W_i . Component
 324 W_i has the following variables: ok_i and id_i . The variable ok_i is a boolean used
 325 to convey the positive or negative acknowledgement, it is initially set to **false**,
 326 while the variable id_i contains a unique identifier of the worker. Additionally,
 327 for each worker component, we associate the ports: $vote_i(id_i, ok_i)$, $prepare_i$,
 328 ack_i , and $fail_i$. Port $vote_i$ is used to send to the coordinator the identifier

329 and a positive or negative acknowledgment. Port \mathbf{start}_i is used to prepare the
 330 transaction, port \mathbf{ack}_i is used to request the final commit, while port \mathbf{fail}_i is
 331 used to request a rollback.

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

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

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

$$\begin{aligned}
 & \left(\begin{array}{c} \mathbf{phase1}_1 \parallel \\ \vdots \\ \parallel \mathbf{phase1}_n \end{array} \right) \bullet \mathbf{C} \oplus \{ \mathbf{C.cond}[|\mathbf{cs}| = n \wedge \mathbf{res}, \mathbf{reset}] : \left(\begin{array}{c} \mathbf{phase2a}_1 \parallel \\ \vdots \\ \parallel \mathbf{phase2a}_n \end{array} \right), \\
 & \qquad \qquad \qquad \mathbf{C.cond}[\neg(|\mathbf{cs}| = n \wedge \mathbf{res}), \mathbf{reset}] : \left(\begin{array}{c} \mathbf{phase2b}_1 \parallel \\ \vdots \\ \parallel \mathbf{phase2b}_n \end{array} \right) \}
 \end{aligned}$$

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

357 In the first phase ($\mathbf{phase1}_i$), the coordinator initiates the transaction ($\mathbf{C.begin}$
 358 $\longrightarrow \mathbf{W}_i.\mathbf{prepare}_i$). Then the worker performs the staging choreography (\mathbf{stage}_i),
 359 and once it is complete, communicates its' result (stored in \mathbf{ok}_i) and its' iden-
 360

361 tifier to the coordinator (using its interface $W_i.\text{vote}_i$). Upon reception, the
 362 coordinator updates the vote by performing a conjunction ($\text{res} = \text{res} \wedge \text{rok}$),
 363 so as to ensure *all* workers vote to commit, and updates the workers list by
 364 adding the worker identifier ($\text{cs} = \text{cs} \cup \{\text{rid}\}$). We note here, that while there
 365 is overlap on the port $C.\text{begin}$ and the receiving variables cs and res , that it is
 366 easy to resolve such overlap, as the variables are updated using an associative
 367 and commutative operators (\wedge and \cup) which are not affected by order of recep-
 368 tion. (Something to be said about the variables rok and rid being that each
 369 receive binds those, and they cannot be overwritten.)

370 When initiating the second phase, the coordinator branches to verify that
 371 all workers voted ($|\text{cs}| = n$), and that their vote was **true** ($\text{res} = \text{true}$). If the
 372 condition is satisfied, the coordinator initiates parallel composition of choreo-
 373 graphs to commit (phase2a_i). Otherwise it initiates a parallel composition
 374 of choreographies to rollback (phase2b_i). For both branches, the coordinator
 375 resets the state of the vote (reset), to refresh acknowledgments. Each choreog-
 376 raphy phase2a_i notifies the port ack_i which is followed by worker performing
 377 commit_i and returning an acknowledgement. Alternatively, phase2b_i notifies
 378 the port fail_i which is followed by worker performing roll_i and returning an
 379 acknowledgement.

380 7. Distributed Component-based Framework

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

391 7.1. Atomic Components

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

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

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

402 the source and destination location, p is the executed port, g is the guard, and
 403 f is the update function.

404 The semantics of an atomic component is defined as an LTS. A state of the
 405 LTS consists of a location ℓ and valuation v of the variables where a valuation
 406 is a function from the variables of the component to a set of values. The atomic
 407 component can transition from state (ℓ, v) to state (ℓ', v') using a transition
 408 $(\ell, p, d, g, f, \ell') \in T$ if (i) the guard of the transition holds ($g(v)$ holds true) (ii)
 409 the application of update function f to valuation v_{pd}/v yields v' where v_{pd} is the
 410 valuation associating $p.\text{var}$ with $d \in \text{Data}$, which is a value possibly received
 411 from other components.

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

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

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

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

421 To later construct a system, we shall use a set of n atomic components
 422 $\{B_i = (P_i, Q_i, T_i)\}_{i=1}^n$

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

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

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

431 An interaction $(p_i, \{p_j\}_{j \in J})$ is said to be synchronous (resp. asynchronous) iff
 432 $\text{isSSend}(p_i)$ (resp. $\text{isASend}(p_i)$) holds.

433 7.2. Composite Components

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

$$\begin{array}{c}
\text{isSSend}(p_i) \\
a = (p_i, \{p_j\}_{j \in J}) \in \gamma \\
d = q_i(p_i.\text{var}) \in \text{Data} \\
\hline
\forall k \in J \cup \{i\} : q_k \xrightarrow{p_k/d} q'_k \quad \forall j \in J : q_j(p_j.\text{buff}) = \epsilon \\
\forall k \notin J \cup \{i\} : q_k = q'_k \\
\hline
(q_1, \dots, q_n) \xrightarrow{a} (q'_1, \dots, q'_n) \quad (\text{synch-send})
\end{array}$$

$$\begin{array}{c}
\text{isASend}(p_i) \\
a = (p_i, \{p_j\}_{j \in J}) \in \gamma \\
d = q_i(p_i.\text{var}) \in \text{Data} \\
\hline
\forall k \in J \setminus \{i\} : q'_k = q_k \quad \forall j \in J : \\
q_i \xrightarrow{p_i/d} q'_i \quad q'_j(p_j.\text{buff}) = q_j(p_j.\text{buff}) \cdot d \\
\hline
(q_1, \dots, q_n) \xrightarrow{a} (q'_1, \dots, q'_n) \quad (\text{asynch-send})
\end{array}$$

$$\begin{array}{c}
\text{isRecv}(p_j) \quad q_j \xrightarrow{p_j/d} q'_j \quad q_j(p_j.\text{buff}) = d \cdot D \quad d \in \text{Data} \\
\forall k \neq j : q_k = q'_k \quad q'_j(p_j.\text{buff}) = D \quad D \in \text{Data}^* \\
\hline
(q_1, \dots, q_n) \xrightarrow{\tau} (q'_1, \dots, q'_n) \quad (\text{recv})
\end{array}$$

$$\begin{array}{c}
\text{isInternal}(p_i) \quad q_i \xrightarrow{p_i/\perp d} q'_i \quad \forall k \neq i : q_k = q'_k \\
\hline
(q_1, \dots, q_n) \xrightarrow{\tau} (q'_1, \dots, q'_n) \quad (\text{internal})
\end{array}$$

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

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

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

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

- 445 • Rule **(synch-send)** describes synchronous interactions, i.e., the interactions
446 of the form $(p_i, \{p_j\}_{j \in J})$ where $\text{isSSend}(p_i)$, where some component B_i
447 synchronously sends to some components $B_j, j \in J$. The variable attached
448 to port p_i of B_i ($p_i.\text{var}$) gets evaluated to some value $d \in \text{Data}$, which
449 is transmitted. All components $B_k, k \in J \cup \{i\}$, perform a transition
450 $q_k \xrightarrow{p_k/d} q'_k$, and other components do not move ($q_k = q'_k$ for $k \notin J \cup \{i\}$).
451 The rule requires that all the corresponding receive ports have no pending
452 messages (their buffers are empty, i.e., $\forall j \in J : q_j(p_j.\text{buff}) = \epsilon$). The
453 states of all the involved components are simultaneously updated through
454 the transition $q_k \xrightarrow{p_k/d} q'_k$, for $j \in J \cup \{i\}$.
455 • Rule **(asynch-send)** describes asynchronous interactions, i.e., the interac-
456 tions of the form $(p_i, \{p_j\}_{j \in J})$ where $\text{isSSend}(p_i)$, where some component

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

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

468 • Rule (**internal**) describes the autonomous execution of an internal port p_i
 469 of component B_i where only the local state of B_i is updated by performing
 470 the transition $q_i \xrightarrow{p_i/\perp_d} q'_i$.

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

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

475 8. Transformations

476 We start with a composite component consisting of n atomic components
 477 $\{B_1, \dots, B_n\}$ with their interface ports and variables. That is, the behaviors of
 478 the input atomic components are empty. Atomic components can be considered
 479 as services with their interfaces but with undefined behaviors.

480 In this section, we define how to automatically synthesize the behavior of
 481 atomic components corresponding to a global choreography model ch . The
 482 distributed system associated with ch is noted $\llbracket \text{ch} \rrbracket$, and is inductively defined
 483 over ch . To realize choreographies as atomic components we follow the syntactic
 484 structure of the choreography. This facilitates the definition of the transforma-
 485 tion from choreographies to components and lead to a clearer implementation.

486 8.1. Preliminary Notions and Notation

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

- 492 • are involved in the realization of the choreography – function \mathcal{C} .
- 493 • need to be notified for the choreography to start – function **start**,

494 • need to terminate for the choreography to terminate – function **end**,

495 The definitions of the two latter functions follow from the semantics of chore-
 496 ographies (Definition 5). Note, in the following definitions, when referring to a
 497 port p with a guard and/or update function involved in a choreography, we note
 498 $p[-]$ when the guard and/or update function is irrelevant to the definition.

499 *Function \mathcal{C} .* We define $\mathcal{C}(\text{ch})$ as the set of indexes of all components involved
 500 in choreography ch .

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

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

503 *Function **start**.* We define $\text{start}(\text{ch})$ as the set of indexes of the components
 504 in ch that should be notified to trigger the start of ch .

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

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

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

516 *Function end.* Similarly, we define $\text{end}(\text{ch})$ as the set of indexes of the compo-
 517 nents involved in ch that need to terminate so that ch terminates.

518 **Definition 13 (Function end).** *Function $\text{end} : \text{Choreographies} \rightarrow 2^{[1,n]} \setminus \{\emptyset\}$*
 519 *is inductively defined over choreographies as follows:*

$$\begin{aligned}
 \text{end}(\text{nil}) &= \emptyset \\
 \text{end}(\text{snd}[-] \longrightarrow \{\text{rcv_list}\}) &= \mathcal{C}(\text{rcv_list}) \text{ if } \text{snd} \in \mathcal{P}^{\text{ss}} \\
 \text{end}(\text{snd}[-] \longrightarrow \{\text{rcv_list}\}) &= \mathcal{C}(\text{snd}) \text{ if } \text{snd} \in \mathcal{P}^{\text{as}} \\
 \text{end}(B \oplus \{\text{cont_list}\}) &= \mathcal{C}(\text{cont_list}) \\
 \text{end}(\text{while}(\text{snd}) \text{ch} \text{end}) &= \mathcal{C}(\text{snd}) \\
 \text{end}(\text{ch}_1 \bullet \text{ch}_2) &= \text{end}(\text{ch}_2) \\
 \text{end}(\text{ch}_1 \parallel \text{ch}_2) &= \text{end}(\text{ch}_1) \cup \text{end}(\text{ch}_2)
 \end{aligned}$$

520 We consider that a synchronous send/receive is terminated when all the compo-
 521 nents involved in the sending and receiving ports are terminated. However,
 522 if the send part is asynchronous, any subsequent choreography can start af-
 523 ter the sending is complete. Conditional master branching choreographies are
 524 terminated when the corresponding master component has terminated. Itera-
 525 tive choreographies are terminated when the component of the send port (with
 526 its guard used as condition) has terminated. A choreography consisting of the
 527 sequential composition of two choreographies has terminated when the second
 528 choreography in the composition has terminated. A choreography that consists
 529 of the parallel composition of two choreographies has terminated when the first
 530 and second choreographies have terminated.

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

536 8.2. Generation of Distributed CBSs

537 We consider a global choreography ch defined over the set of ports $\mathcal{P} =$
 538 $\cup_{i=1}^n \mathcal{P}_i$ of a given set of atomic components (with empty behavior) with their
 539 corresponding variables. Given a choreography ch , we define a set of transforma-
 540 tions that allows to generate the behaviors and the corresponding interactions
 541 of the distributed components $S = (B, \text{init})$. Moreover, as we progressively
 542 build system S , we consider that it has a context to denote the current state
 543 where a choreography should be appended. For this, $\mathcal{S} = (S, \text{context})$ denotes
 544 a system with its corresponding context where context is a function that takes
 545 an atomic component as input and returns a location, i.e., $\text{context}(B_i) \in L_i$
 546 to denote the current context of atomic components B_i . The building of the
 547 final system is done by induction, following the syntactic structure of the input
 548 choreography and uses the continuously updated context. Any step for con-
 549 structing the component ensures that the context of each component consists
 550 of a unique state.

551 Initially, we consider a system skeleton $\mathcal{S} = (S, \text{context})$, where $B =$
552 $\gamma(B_1, \dots, B_n)$ with: (1) $\gamma = \emptyset$; (2) $B_i = (P_i, \emptyset, \{l_i\}, \emptyset)$; (3) $init = (l_1^{\text{init}}, \dots, l_n^{\text{init}})$;
553 and (4) $\text{context}(B_i) = l_i^{\text{init}}$; for $i \in [1, n]$. The initial location of the obtained
554 system remains unchanged, i.e., it is *init*. As such, for the sake of clarity, we
555 omit it in our construction. Moreover, all variables are initialized to their default
556 value.

557 8.2.1. Send/Receive

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

572 **Remark 2.** *Creating a copy for each port per choreography is necessary to*
573 *generate efficient and correct distributed implementation. As for efficiency,*
574 *consider the choreography $p_1 \longrightarrow \{p_2\} \bullet p_1 \longrightarrow \{p_3\}$. Its corresponding dis-*
575 *tributed implementation would require to create two interactions $(p_1, \{p_2\})$ and*
576 *$(p_1, \{p_3\})$. As such, the component that corresponds to p_1 (B_1) needs to interact*
577 *B_2 and B_3 to know which interaction must be executed (depending on their cur-*
578 *rent enable ports). However, if we create a copy of the ports, each port will be*
579 *connected to one and only interaction, hence component B_1 can locally decide,*
580 *without interacting with other components, on the interaction to be executed. As*
581 *for correctness, consider the choreography $p_1 \longrightarrow \{p_2, p_3\} \bullet p_1 \longrightarrow \{p_2\}$. Accord-*
582 *ing to the choreography semantics, we should first execute $p_1 \longrightarrow \{p_2, p_3\}$ then*
583 *$p_1 \longrightarrow \{p_2\}$. Consider that we are in a state where p_1 and p_2 are enabled but*
584 *p_3 . This may happen when the component that corresponds to p_3 is still exe-*
585 *cuting the function of the previous transition. In this case, B_1 would interact*
586 *with B_2 and B_3 to know which interaction to execute. As p_3 is not currently en-*
587 *abled, component B_1 will execute the interaction connected with p_2 only, hence*
588 *violating the sequential semantics.*

Definition 14 (Send/Receive).

$\llbracket psas[g, f] \longrightarrow \{\text{rcv_list}\} \rrbracket(\gamma(B_1, \dots, B_n), \text{context}) = (\gamma'(B'_1, \dots, B'_n), \text{context}')$, with:

589 $\bullet B'_k = \begin{cases} (P_k, L'_k, T'_k) & \text{if } k \in \mathcal{C}(psas[g, f]) \cup \mathcal{C}(\text{rcv_list}) \\ B_k & \text{otherwise} \end{cases}$, where:

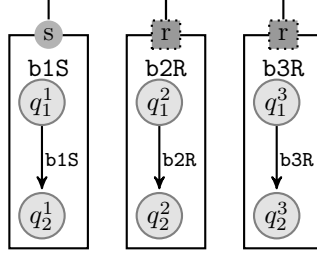


Figure 5: Send/Receive Transformation

$$\begin{aligned}
590 \quad & - L'_k = L_k \cup \{l_k^{\text{new}}\} \\
591 \quad & - T'_k = T_k \cup \begin{cases} \{\text{context}(B_k) \xrightarrow{\text{copy}(psas), g, f} l_k^{\text{new}}\} & \text{if } psas[g, f] \in B_k.\mathcal{P}^{\text{ss}} \cup B_k.\mathcal{P}^{\text{as}} \\ \{\text{context}(B_k) \xrightarrow{\text{copy}(p_k), \text{true}, p_k.\text{ufct}} l_k^{\text{new}}\} & \text{if } p_k \in \text{rcv_list} \end{cases} \\
592 \quad & \bullet \gamma' = \gamma \cup \{(\text{copy}(psas), \{\text{copy}(p_i) \mid p_i \in \text{rcv_list}\})\}, \\
593 \quad & \bullet \text{context}'(B'_k) = \begin{cases} l_k^{\text{new}} & \text{if } k \in \mathcal{C}(psas[g, f]) \cup \mathcal{C}(\text{rcv_list}) \\ \text{context}(B_k) & \text{otherwise} \end{cases} .
\end{aligned}$$

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

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

604 8.2.2. Branching Composition

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

615 For this, we define a union operator, noted **union**, that takes a set of systems
616 with their contexts and (1) unions all of their locations, transitions and ports;
617 then (2) updates the contexts of the obtained components by joining each of their

618 input contexts with internal transitions. Therefore, after applying branching
 619 we guarantee that each component will have one and only one context location.
 620 Formally, operator **union** is defined as follows.

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

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

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

Definition 16 (Branching).

$$\begin{aligned} & \llbracket B_i \oplus \{p_i^l[g_l, f_l] : \text{ch}_l\}_{l \in L} \rrbracket(S, \text{context}) \\ & = \text{union}(\{\llbracket \text{ch}_l \rrbracket \llbracket p_i^l[g_l, f_l] \longrightarrow \{p_k^{\text{ctrl}}[\emptyset]\}_{k \in K} \rrbracket(S, \text{context})\}_{l \in L}) \end{aligned}$$

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

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

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

635 *In this choreography, if we would have not sent the choice made by component
 636 1 to component 3, then component 3 cannot know about the decision that was
 637 taken by component 1. Hence, it cannot decide whether to follow choreography
 638 α or β afterwards.*

639 **Example 2 (Branching).** *Figure 6 shows an abstract example on how to apply
 640 a branching operation that consists of two choices $B_1 \oplus \{b_1^{l_1}[g_1, f_1] : \text{ch}_1, b_2^{l_2}[g_2, f_2] :$
 641 $\text{ch}_2\}$. First, we add choice transitions to component B_1 and synchronize them
 642 with the participants of ch_1 and ch_2 , e.g., B_2 and B_3 . Then, we apply the choreo-
 643 graphies accordingly. Finally, we merge the contexts with internal transitions.*

644 8.2.3. Loop Composition

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

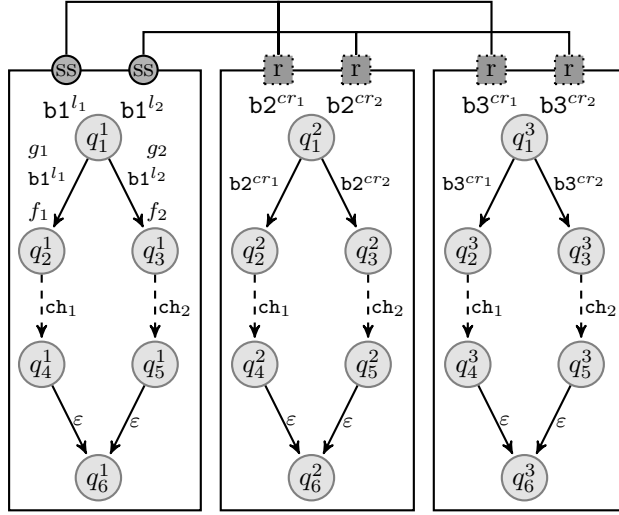


Figure 6: Branching transformation

Definition 17 (Loop).

let $K = \mathcal{C}(\text{ch}) \setminus \{i\}$

let $(\gamma^t(B_1^t, \dots, B_n^t), \text{context}^t) = \llbracket \text{ch} \rrbracket \llbracket \text{snd}[g, f] \longrightarrow \{pr_k^{\text{cont}}[\emptyset]\}_{k \in K} \rrbracket (S, \text{context})$

let $(P_i^t, -, L_i^t, T_i^t) = B_i^t$, for $i \in [1, n]$

in $\llbracket \text{while}(\text{snd}[g, f]) \text{ch end} \rrbracket (S, \text{context}) = (\gamma'(B'_1, \dots, B'_n), \text{context}')$

where:

let p_j^f and l_j^c be new synchronous ports and locations, for $j \in K \cup \{i\}$

649 • $P'_j = P_j^t \cup \begin{cases} \{p_j^f\} & \text{if } j \in K \cup \{i\} \\ \emptyset & \text{otherwise} \end{cases} ;$

650 • $L'_j = L_j^t \cup \begin{cases} \{l_j^c\} & \text{if } j \in K \cup \{i\} \\ \emptyset & \text{otherwise} \end{cases} ;$

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

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

653 • $\text{context}'(B'_j) = \begin{cases} l_j^c & \text{if } j \in K \cup \{i\} \\ \text{context}(B_j) & \text{otherwise} \end{cases} .$

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

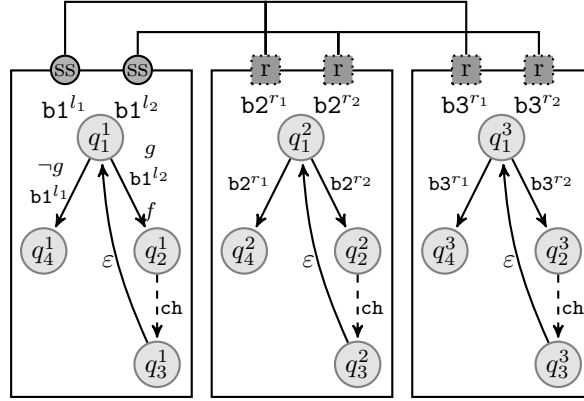


Figure 7: Loop composition transformation

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

661 8.2.4. Sequential Composition

662 The binary operator \bullet allows to sequentially compose two choreographies,
 663 $\text{ch}_1 \bullet \text{ch}_2$. For this, its semantics is defined by (1) applying ch_1 ; (2) notifying the
 664 start of ch_2 ; and finally (3) applying ch_2 . As we require that ch_1 must terminate
 665 before the start of ch_2 , we need to synchronize all the end components of ch_1
 666 with all the start components of ch_2 . To do so, it is sufficient to pick one of the
 667 end components of ch_1 and create a synchronous send port, which is connected
 668 to new receive ports added to the remaining end components of ch_1 and start
 669 components of ch_2 . Moreover, the application of the sequential composition
 670 guarantees that each component of the resulting system consists of exactly one
 671 state, provided that the context of each component of the initial system consists
 672 of one state. Formally, the semantics of the sequential composition is defined
 673 as follows.

Definition 18 (Sequential Composition).

$$\llbracket \text{ch}_1 \bullet \text{ch}_2 \rrbracket(S, \text{context}) = \llbracket \text{ch}_2 \rrbracket \llbracket \text{ch}_{\text{synch}} \rrbracket \llbracket \text{ch}_1 \rrbracket(S, \text{context}), \text{ with:}$$

674 $\text{ch}_{\text{synch}} = p_i^{\text{cs}}[\text{true}, \emptyset] \longrightarrow \{p_j^{\text{cr}}[\text{true}, \emptyset]\}_{j \in J}$ such that: (1) $i \in \text{end}(\text{ch}_1)$; (2)
 675 $J = \text{end}(\text{ch}_1) \cup \text{start}(\text{ch}_2) \setminus \{i\}$; (3) p_i^{cs} is a new synchronous send port to be
 676 added to $\mathcal{P}_i^{\text{ss}}$; and (4) $\{p_j^{\text{cr}}\}_{j \in J}$ are new receive ports to be added to \mathcal{P}_j^{r} .

677 **Example 4 (Sequential composition).** Figure 8 shows an abstract example
 678 on how to transform sequential composition of two choreographies, $\text{ch}_1 \bullet \text{ch}_2$,
 679 into an initial system consisting of five components. Here we only consider
 680 components that are involved in those choreographies, where (1) components b_1 ,
 681 b_2 , b_3 and b_4 are involved in choreography ch_1 ; and (2) components b_1 , b_2 , b_3

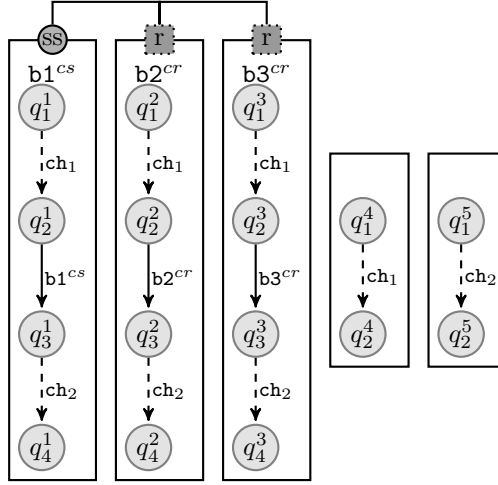


Figure 8: Sequential composition transformation

682 and b_5 are involved in choreography ch_2 . Note, components that are not involved
 683 are kept unchanged. The transformation requires to: (1) apply first choreography
 684 ch_1 to its participated components (i.e., b_1 , b_2 , b_3 and b_4); (2) synchronize the
 685 end of choreography ch_1 (e.g., b_1) with the start of choreography ch_2 (e.g., b_2 and
 686 b_3). To do so, we create a synchronous send port to one of the end components
 687 of ch_1 (e.g., b_1^{cs}) and connect it to all the remaining end components of ch_1
 688 (e.g., \emptyset and the start components of ch_2 (e.g., b_2 and b_3); finally (3) we apply
 689 choreography ch_2 .

690 8.2.5. Parallel Composition

691 The binary operator \parallel allows for the parallel compositions of two indepen-
 692 dent choreographies. Two choreographies are independent if their participating
 693 components are disjoint.

694 **Definition 19 (Independent Choreographies).** Two choreographies ch_1 and
 695 ch_2 are said to be independent iff $\mathcal{C}(\text{ch}_1) \cap \mathcal{C}(\text{ch}_2) = \emptyset$.

696 We consider independent choreographies to avoid conflicts and interleaving of
 697 executions within components. In addition, this simplifies reasoning and writing
 698 choreographies as well as for efficient code generation. Note that parallelizing
 699 independent choreographies implies that each component has a single execution
 700 flow. In case we have overlap, e.g., $p_1 \longrightarrow \{p_2, p_3\} \parallel p_1 \longrightarrow \{p_5\}$, we could
 701 split p_1 into two different components. Moreover, it is possible to enforce any
 702 arbitrary order of execution. Further, we discuss other possible alternatives for
 703 handling this case. This would not reduce the expressiveness of our model as
 704 parallel execution flows can be modelled in separate components. The semantics
 705 of the parallel composition $\text{ch}_1 \parallel \text{ch}_2$ is simply defined by applying ch_1 and ch_2
 706 in any order, which leads to the same system as the two choreographies are

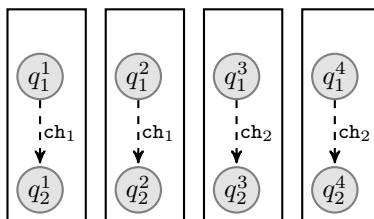


Figure 9: Parallel composition transformation

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

Definition 20 (Parallel Composition).

$$\llbracket \text{ch}_1 \parallel \text{ch}_2 \rrbracket (S, \text{context}) = \llbracket \text{ch}_2 \rrbracket \llbracket \text{ch}_1 \rrbracket (S, \text{context})$$

711 **Example 5 (Parallel Composition).** Figure 8 shows an abstract example on
 712 how to transform parallel composition of two choreographies, $\text{ch}_1 \parallel \text{ch}_2$, into an
 713 initial system consisting of five components. Here, we consider that ch_1 (resp.
 714 ch_2) involves components B_1 and B_2 (resp. B_3 and B_4).

715 The following proposition is a straightforward consequence of the transforma-
 716 tion associated with the \parallel operator and the fact that the transformation of a
 717 choreography only modifies the component involved in this choreography.

718 **Proposition 1.** If ch_1 and ch_2 are two independent choreographies, then $\llbracket \text{ch}_1 \parallel$
 719 $\text{ch}_2 \rrbracket = \llbracket \text{ch}_2 \parallel \text{ch}_1 \rrbracket$.

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

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

729 8.3. Discussion on the Correctness of the Synthesis Method

730 We conjecture that a choreography ch and its corresponding synthesized
 731 distributed system obtained by the transformations in this section are weakly
 732 bisimilar. Below we give some arguments based on the structure of the choreo-
 733 graphy. A full proof is left for future work.

- 734 • In the case of send/receive choreographies. The execution of choreogra-
735 phies follows rules (`synch-sendrcv`) for synchronous send, (`asynch-sendrcv-1`)
736 and (`asynch-sendrcv-2`) for asynchronous send. The execution of dis-
737 tributed systems follows rule (`synch-send`). The transformation is imple-
738 mented by the interaction added in Definition 14; see Figure 5.
- 739 • In the case of branching choreographies. The execution of choreographies
740 follows rule (`master-branching`). The transformation is implemented by
741 Definition 16 where we create the appropriate interactions to implement
742 the master branching rule, as depicted in Figure 6.
- 743 • In the case of looping choreographies. The execution of choreographies
744 follows rules (`iterative-tt`) and (`iterative-ff`). The transformation is
745 implemented by Definition 17 where we create the appropriate interactions
746 and behavior to implement the looping rule, as depicted in Figure 7.
- 747 • In the case of sequential choreographies. The execution of choreographies
748 follows rules (`sequential-1`) and (`sequential-2`). The transformation is
749 implemented by Definition 18 where we add an interaction and behavior
750 to implement the sequential rules and guarantee the sequential execution
751 of the input choreographies, as depicted in Figure 8.
- 752 • In the case of parallel choreographies. The execution of choreographies
753 follows rules (`parallel-1`), (`parallel-2`), (`parallel-3`), and (`parallel-4`).
754 The transformation is implemented by Definition 20 where we transform
755 each choreography independently, as depicted in Figure 9.

756 9. Code Generation

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

759 Code generation takes as input a choreography and a configuration file con-
760 taining the list of components with their corresponding interfaces/ports and
761 variables. Clearly, the choreography is defined with respect to the components'
762 ports, with functions and guards defined with respect to the components' vari-
763 ables. We only consider independent choreographies, as described in Defini-
764 tion 19. Note, if the components are not independent, we can follow the strate-
765 gies described in Remark 4. Code generation then automatically produces the
766 corresponding implementation of each of the components. Following our trans-
767 formation into Distributed CBS in Section 8.2, the obtained components have
768 the following characteristics: (1) they do not have a location with outgoing send
769 and receive ports; (2) a port is connected to exactly one interaction. As such,
770 there are no conflicting interactions that can run concurrently. Two interactions
771 are said to be conflicting iff they share a common component. Consequently, it
772 is possible to generate fully distributed implementations, with no need for con-
773 trollers (unlike [7]) for managing multiparty interactions. Hence, the number of
774 exchanged messages will be divided by 2 for each execution of an interaction.

Algorithm 1: Pseudo-code - generated components.

```
1 initialization();
2 while true do
3   if all outgoing transitions are send then
4     port p = select enabled port, i.e., guard true;
5     notify all the receivers of the interaction that has port p;
6     if p is synchronous then
7       | wait for ack. from the receivers;
8     end
9   end
10  else if all outgoing transitions are receive then
11    wait until a message is ready in one of the outgoing receive ports;
12    port p = select message;
13    if interaction connected is synchronous then
14      | send ack. to the corresponding send port;
15    end
16  updateCurrentState();
17 end
```

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

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

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

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

- 784 • The component waits until a message is ready/received in one of the
785 receive ports.
- 786 • Upon receiving a message, we acknowledge its receipt if the port is con-
787 nected to a synchronous interaction.

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

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

795 10. Building Micro-Services Using Choreography

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

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

806 The corresponding global choreography CH is defined in Listing 1.

- 807 • CH_1 : A client (c) sends a request to the *gateway service* (gs), which is
808 the only visible micro-service to the client, containing the required version,
809 revision, pool name, and an identifier to the testing data. gs forwards the
810 request to the *deploy environment service* (des). des creates an envi-
811 ronment id and returns it back to gs , which in turn forwards it back to
812 c .
- 813 • CH_2 : des sends to the *deploy application directory service* ($dads$) and the
814 *deploy database service* (dds) (i) required version, revision and pool name
815 and (ii) testing data identifier and environment id, respectively. c keeps
816 checking if the environment is ready, which is done through the gateway
817 service with the help of the *environment info. service* (eis).
- 818 • CH_3 : $dads$ requests from the *machine service* (ms) and the *setup service*
819 (ss) (i) a machine location from the pool and (ii) the package location,
820 respectively. When $dads$ receives the replies from both ms and ss , it con-
821 tacts the appropriate *host machine* (hm_i) by sending the package location.
822 Then, hm_i sends its status to des . des upon receiving the status update,
823 it forwards it to the eis . dds requests from the *dumps service* (dus) and
824 the *Database machines services* (dms) (i) testing data location, and (ii) a
825 database server, respectively. When dds receives the replies from both dus
826 and dms , it contacts the appropriate *database server* hd_j by sending the
827 testing data location. Then, hd_j sends its status to des . Upon receiving
828 the status update, des forwards it to eis .

829 For each micro-service/component m , we denote by mSS , mAS mR a correspond-
830 ing synchronous send, asynchronous send and receive port, respectively.

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

Listing 1: Global choreography

```

CH = CH1 • CH2 • CH3
CH1 = cSS → gsR • gsSS → desR • desAS → gsR
CH2 = CH21 • CH22
CH21 = gsSS → cR || (desAS → dadsR • desAS → dadsR)
CH22 = while(cSS) cSS → gsR •
          gsSS → eisR • eisSS → gsR • gsSS → cR end
CH3 = (CH4 || CH5) • CH6
CH4 = CH41 • CH42 • CH43
CH41 = dadsAS → amsR • dadsAS → SSR
CH42 = amsSS → dadsR || ssSS → dadsR
CH43 = dads ⊕ {li : dadsSS → hmiR • hmiSS → desR}
CH5 = CH51 • CH52 • CH53
CH51 = ddsAS → dusR • ddsAS → SSR
CH52 = dusSS → ddsR || dmsSS → dadsR
CH53 = dds ⊕ {li : ddsSS → hdiR • hdiSS → desR}
CH6 = desAS → eisR

```

```

createPromela() {
  createChannels();
  foreach Bi {
    createProcess(i);
  }
}

```

Listing 2: Main Code Generation from System S to Promela

838 11. Transformation to Promela

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

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

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

```

1 createChannels()
2   foreach a ∈ γ, where a = (ps, {pri}i∈I) {
3     foreach p ∈ {pri}i∈I {
4       if (isSSend(ps))
5         append chan channelP = [0] of {ps.dtype};
6       else
7         append chan channelP = [MAX_LEN] of {ps.dtype};
8       end
9     end
10  end

```

Listing 3: createChannels Skeleton

852 message carrier type). The type of the channel is the data type of the corre-
853 sponding send port (i.e., $p.dtype$). For synchronous (resp. asynchronous) ports,
854 we use a channel of length 0 (resp. `MAX_LEN`).

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

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

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

864 12.1. Specification of the Buying System

865 Buyer 1 sends a book title to the Seller, who replies to both buyers by quoting
866 a price for the given book. Depending on the price, Buyer 1 may try to haggle
867 with Seller for a lower price, in which case Seller may either accept the new
868 price or call off the transaction entirely. At this point, Buyer 2 takes Seller's
869 response and coordinates with Buyer 1 to determine how much each should pay.
870 In case Seller chose to abort, Buyer 2 would also abort. Otherwise, it would
871 keep negotiating with Buyer 1 to determine how much it should pay. Buyer
872 1, having a limited budget, consults with the bank before replying to Buyer 2.
873 Once Buyer 2 deems the amount to be satisfactory, he will ask the bank to pay
874 the seller the agreed upon amount (Buyer 1 would be doing the same thing *in*
875 *parallel*).

876 12.2. Synthesizing the Implementation

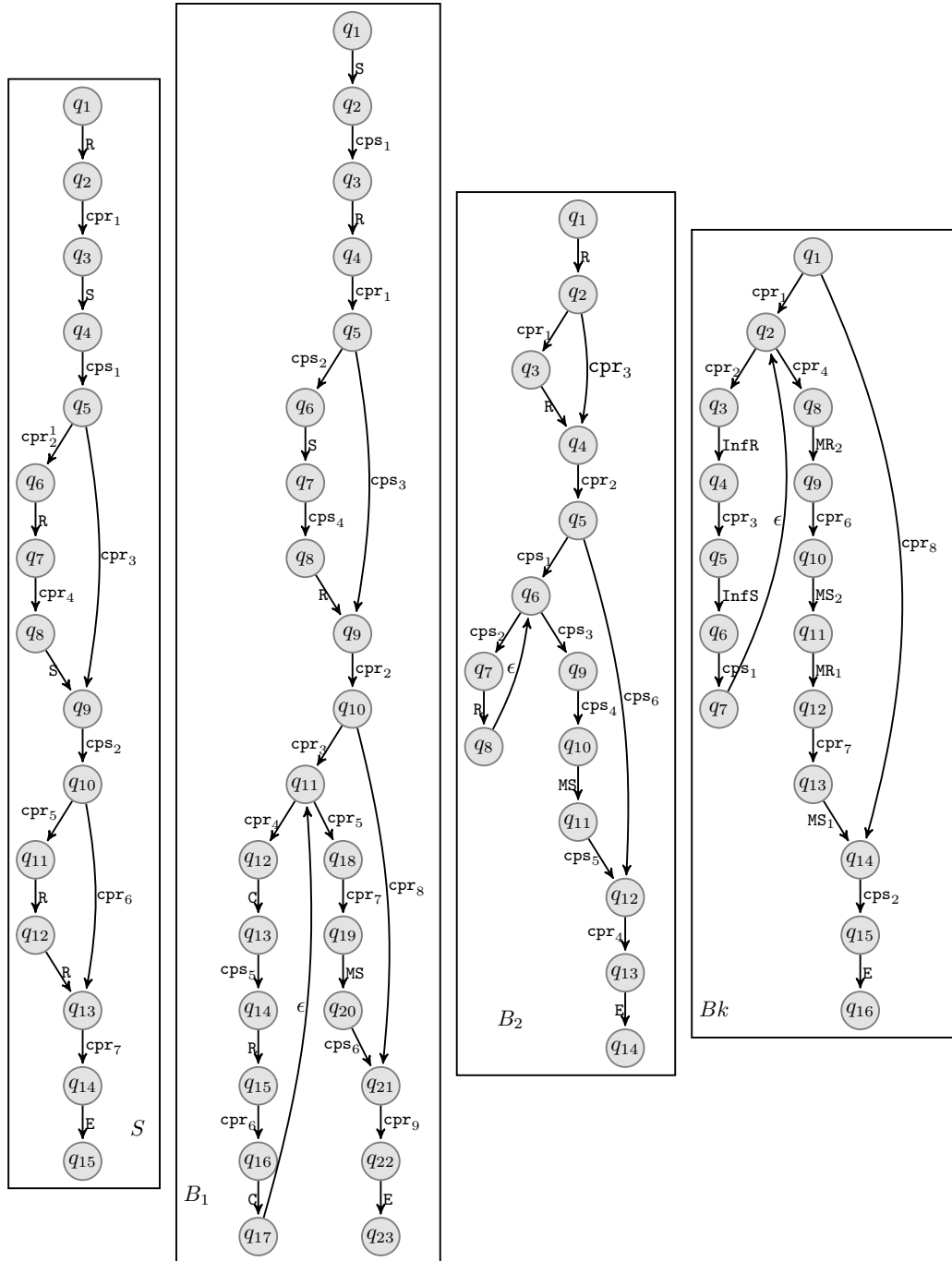


Figure 10: Components generated from the choreography in Listing 5.


```

1 createProcess(int id) {
2   append proctype process(int id) {
3     append int currentLocation = initialLocation;
4     append currPort = _;
5     append do
6     append :: if
7     append :: (all current outgoing trans. are send) ->
8     append ps = pickEnablePort(); // w.r.t. guard
9     append currPort = ps;
10    foreach p ∈ {pri}i∈I, where ∃a = (ps, {pri}i∈I) ∈ γ {
11      append channelP!(msg);
12    }
13    append if
14    append :: (all outgoing are synchronous send) ->
15    foreach p ∈ {pri}i∈I, where ∃a = (ps, {pri}i∈I) ∈ γ {
16      append channelP?(_);
17    }
18    append fi;
19    append :: else -> // outgoing transitions are receive
20    // listening to all current channels
21    append if
22      foreach p: currentLocation  $\xrightarrow{p}$ 
23      append ::(channelP?(val)) -> currPort = p;
24      if(p is connected to synchronous send) {
25        append channelP!(ack);
26      }
27      append fi;
28    append fi;
29    // Update current location and execute location function
30    // of the current outgoing transition.
31    append updateCurrentState();
32  append od;
33  append }
34 }

```

Listing 4: createProcess Skeleton

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

884 *Synthesizing the distributed component-based system.* We apply our transfor-
885 mation to the choreography in Listing 5 and obtain the distributed component-
886 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

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

895 *Promela version of the implementation.* To verify that the distributed imple-
896 mentation respects some desired properties, we apply our transformation of dis-
897 tributed component-based systems to **Promela** which constitutes a translation
898 of the choreography behavior.

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

903 With the macros defined in Listing 6, the **Promela** code generated is depicted
904 in Listing 7.

905 `updateCurrentState` is a macro that updates the current location and exe-
906 cutes the location function of the current outgoing transition. The result of this
907 computation would then be stored in the variable `value`.

908 *12.3. Verifying the Implementation*

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

913 *Correct termination.* The correct termination property require that “all pro-
 914 cesses terminate if any of them terminate”. Let the ports suffixed by **E** rep-
 915 resent the termination interface/port of the corresponding process. Moreover,
 916 we consider the following atomic propositions $\text{currPort}_1 = \text{Buyer1.currPort}$,
 917 $\text{currPort}_2 = \text{Buyer2.currPort}$, $\text{currPort}_3 = \text{Bank.currPort}$, and currPort_4
 918 $= \text{Seller.currPort}$. Then, correct termination can be expressed as the follow-
 919 ing LTL formula:

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

920 where E_i represents the ending interface of the appropriate process.

921 *Uniqueness of interface calls.* An interface should *only be called once*. In each
 922 run, money is only withdrawn once by each process. Let the port Bk.MS_1 (resp.
 923 Bk.MS_2) represent the withdrawal of money by process 1 (resp. process 2).
 924 Then, specifying that money is withdrawn once per process can be expressed as
 925 the LTL formula:

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

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

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

930 **13. Related Work**

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

¹We 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 .

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

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

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

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

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

976 In [2, 16], the authors introduce syntactic transformations to refine dis-
977 tributed system programs starting from high-level specifications. In [2], the
978 proposed specification differs from our choreography model as it is not possible
979 to express multiparty interactions, or guarded loop, which makes it impracti-
980 cal in the context of distributed systems. In [16], the paper mainly targets

981 multiparty interactions, where the main objective is to loosening synchronous
982 multiparty interaction while preserving its semantics. In our case, as we auto-
983 matically synthesize code for multiply interactions, there is no need for loosening
984 technique. Add to that, we also support asynchronous ports that allow to loos-
985 ening interactions. Additionally, in [2, 16], it is not clear how to automatically
986 generate code from the refined programs.

987 BPMN [31] (Business Process Model and Notation) is an industry standard
988 that allows modeling process choreographies. An extension of BPMN was in-
989 troduced in [20, 28] to automatically derive a local choreography from a global
990 one. Nonetheless, the extension only considers exchange of messages and does
991 not formally define other composition operators such as synchronous multi-
992 party communications, parallelism, choice, sequential and loop. The method
993 proposed in [30] allows deriving RESTful choreographies from process chore-
994 ographies, whereas in this paper we synthesize the code of the processes given
995 global choreography. Moreover, the model is restricted to RESTful architec-
996 ture. In [19], the authors introduce a framework for the verification and design
997 of choreographies, however, the communication model only allows for one send
998 and one receive per interaction.

999 **14. Conclusion and Future Work**

1000 *Conclusion.* This paper deals with the synthesis of distributed implementations
1001 of local processes (control flows, synchronization, notification, acknowledgment,
1002 computations embedding), starting from a global choreography. The method
1003 presented in this paper allows one to automatically verify the communication
1004 protocols and drastically simplify the synthesis of the distributed implemen-
1005 tation. Moreover, the language is used to model a real case study provided
1006 by Murex S.A.L. services industry. We used the choreography language and
1007 the method to synthesize actual micro-services architectures. The synthesized
1008 micro-services can be verified against any Linear Temporal Logic formula thanks
1009 to a translation to Promela. We illustrated the translation and the verification
1010 on a simplified version of an application at Murex for which we synthesized the
1011 micro-service implementation.

1012 *Future work.* In addition to formally prove the weak bisimilarity between chore-
1013 ographies and the synthesized distributed systems (sketched in Section 8.3),
1014 future work comprises several directions. First, we consider augmenting our
1015 choreography model by adding fault-tolerance primitives. That is, we aim to
1016 specify the number of replicas of each process and automatically embed a con-
1017 sensus protocol between them such as Paxos [24] or Raft [32]. Second, we
1018 consider integrating our framework with Spring Boot to allow for the automatic
1019 generation of RESTful web services starting from global choreography. Third,
1020 we consider augmenting our code generation with features provided by *Istio* [23]
1021 and *Linkerd* [27], which are used for routing, failure handling, service discovery,
1022 the integration of micro-services, the traffic-flow management and enforcing poli-
1023 cies. Fourth, we consider defining a specific model checker for our distributed

1024 component-based framework. Finally, we consider using complementary ver-
1025 ification techniques operating at runtime such as runtime verification [4, 15]
1026 and runtime enforcement [12] for which we defined approaches in the case of
1027 non-distributed component-based systems [14, 13].

1028 *Acknowledgment*

1029 The authors warmly thank the reviewers for their helpful comments on a
1030 preliminary version of this paper. The work presented in this paper is sup-
1031 ported by the Murex Grant Award 103456 and University Research Board at
1032 the American University of Beirut.

1033 **References**

- 1034 [1] Agha, G. A., Kim, W., 1999. Actors: A unifying model for parallel and
1035 distributed computing. *Journal of Systems Architecture* 45 (15), 1263–1277.
1036 URL [https://doi.org/10.1016/S1383-7621\(98\)00067-8](https://doi.org/10.1016/S1383-7621(98)00067-8)
- 1037 [2] Attie, P. C., Das, C., 1997. Automating the refinement of specifications for
1038 distributed systems via syntactic transformations. *Int. J. Systems Science*
1039 28 (11), 1129–1144.
- 1040 [3] Autili, M., Inverardi, P., Tivoli, M., August 2018. Choreography realizabil-
1041 ity enforcement through the automatic synthesis of distributed coordination
1042 delegates. *Science of computer programming*, 3–29.
- 1043 [4] Bartocci, E., Falcone, Y. (Eds.), 2018. *Lectures on Runtime Verification -*
1044 *Introductory and Advanced Topics*. Vol. 10457 of *Lecture Notes in Com-*
1045 *puter Science*. Springer.
1046 URL <https://doi.org/10.1007/978-3-319-75632-5>
- 1047 [5] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.,
1048 Sifakis, J., 2011. Rigorous component-based system design using the BIP
1049 framework. *IEEE Software* 28 (3), 41–48.
- 1050 [6] Bejleri, A., Yoshida, N., 2009. Synchronous multiparty session types. *Electr.*
1051 *Notes Theor. Comput. Sci.* 241, 3–33.
- 1052 [7] Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J., 2012. A
1053 framework for automated distributed implementation of component-based
1054 models. *Distributed Computing* 25 (5), 383–409.
- 1055 [8] Bonelli, E., Compagnoni, A. B., 2007. Multipoint session types for a dis-
1056 tributed calculus. In: *Trustworthy Global Computing, Third Symposium,*
1057 *TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected*
1058 *Papers*. pp. 240–256.

- 1059 [9] Carbone, M., Montesi, F., 2013. Deadlock-freedom-by-design: multiparty
1060 asynchronous global programming. In: The 40th Annual ACM SIGPLAN-
1061 SIGACT Symposium on Principles of Programming Languages, POPL '13,
1062 Rome, Italy - January 23 - 25, 2013. pp. 263–274.
1063 URL <https://doi.org/10.1145/2429069.2429101>
- 1064 [10] Carbone, M., Montesi, F., Schürmann, C., Feb. 2018. Choreographies, log-
1065 ically. *Distrib. Comput.* 31 (1), 51–67.
1066 URL <https://doi.org/10.1007/s00446-017-0295-1>
- 1067 [11] Charalambides, M., Dinges, P., Agha, G. A., 2016. Parameterized, concu-
1068 rent session types for asynchronous multi-actor interactions. *Sci. Comput.*
1069 *Program.* 115–116, 100–126.
- 1070 [12] Falcone, Y., 2010. You should better enforce than verify. In: Barringer,
1071 H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G. J., Rosu,
1072 G., Sokolsky, O., Tillmann, N. (Eds.), *Runtime Verification - First Inter-*
1073 *national Conference, RV 2010, St. Julians, Malta, November 1-4, 2010.*
1074 *Proceedings.* Vol. 6418 of *Lecture Notes in Computer Science.* Springer,
1075 pp. 89–105.
1076 URL https://doi.org/10.1007/978-3-642-16612-9_9
- 1077 [13] Falcone, Y., Jaber, M., 2017. Fully automated runtime enforcement of
1078 component-based systems with formal and sound recovery. *STTT* 19 (3),
1079 341–365.
1080 URL <https://doi.org/10.1007/s10009-016-0413-6>
- 1081 [14] Falcone, Y., Jaber, M., Nguyen, T., Bozga, M., Bensalem, S., 2015. Run-
1082 time verification of component-based systems in the BIP framework with
1083 formally-proved sound and complete instrumentation. *Software and System*
1084 *Modeling* 14 (1), 173–199.
1085 URL <https://doi.org/10.1007/s10270-013-0323-y>
- 1086 [15] Falcone, Y., Krstic, S., Reger, G., Traytel, D., 2018. A taxonomy for clas-
1087 sifying runtime verification tools. In: Colombo, C., Leucker, M. (Eds.),
1088 *Runtime Verification - 18th International Conference, RV 2018, Limassol,*
1089 *Cyprus, November 10-13, 2018, Proceedings.* Vol. 11237 of *Lecture Notes*
1090 *in Computer Science.* Springer, pp. 241–262.
- 1091 [16] Francez, N., Forman, I. R., 1991. Synchrony loosening transformations for
1092 interacting processes. In: *CONCUR '91, 2nd International Conference on*
1093 *Concurrency Theory, Amsterdam, The Netherlands, August 26-29, 1991,*
1094 *Proceedings.* pp. 203–219.
- 1095 [17] Francez, N., Forman, I. R., 2018. From global choreography to efficient
1096 distributed implementation. In: *HPCS - 4PAD International Symposium*
1097 *on Formal Approaches to Parallel and Distributed Systems.*

- 1098 [18] Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N., Caldeira, A. Z.,
1099 2010. Modular session types for distributed object-oriented programming.
1100 In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Prin-
1101 ciples of Programming Languages, POPL 2010, Madrid, Spain, January
1102 17-23, 2010. pp. 299–312.
- 1103 [19] Gudemann, M., Poizat, P., Salain, G., Ye, L., 2016. Verchor: A frame-
1104 work for the design and verification of choreographies. IEEE Trans. Ser-
1105 vices Computing 9 (4), 647–660.
1106 URL <https://doi.org/10.1109/TSC.2015.2413401>
- 1107 [20] Hofreiter, B., Huemer, C., 2008. A model-driven top-down approach to
1108 inter-organizational systems: From global choreography models to exe-
1109 cutable BPEL. In: 10th IEEE International Conference on E-Commerce
1110 Technology (CEC 2008) / 5th IEEE International Conference on Enter-
1111 prise Computing, E-Commerce and E-Services (EEE 2008), July 21-14,
1112 2008, Washington, DC, USA. pp. 136–145.
1113 URL <https://doi.org/10.1109/CECandEEE.2008.129>
- 1114 [21] Holzmann, G. J., 1997. The model checker SPIN. IEEE Trans. Software
1115 Eng. 23 (5), 279–295.
1116 URL <https://doi.org/10.1109/32.588521>
- 1117 [22] Honda, K., Yoshida, N., Carbone, M., 2008. Multiparty asynchronous ses-
1118 sion types. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Sympo-
1119 sium on Principles of Programming Languages, POPL 2008, San Francisco,
1120 California, USA, January 7-12, 2008. pp. 273–284.
- 1121 [23] Istio, . <https://github.com/istio/istio/>.
- 1122 [24] Lamport, L., December 2001. Paxos made simple, 51–58.
1123 URL <https://www.microsoft.com/en-us/research/publication/paxos-made-simple/>
- 1124 [25] Lange, J., Tuosto, E., 2012. Synthesising choreographies from local session
1125 types. In: CONCUR 2012 - Concurrency Theory - 23rd International Con-
1126 ference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012.
1127 Proceedings. pp. 225–239.
- 1128 [26] Lange, J., Tuosto, E., Yoshida, N., 2015. From communicating machines
1129 to graphical choreographies. In: Proceedings of the 42nd Annual ACM
1130 SIGPLAN-SIGACT Symposium on Principles of Programming Languages,
1131 POPL 2015, Mumbai, India, January 15-17, 2015. pp. 221–232.
- 1132 [27] Linkerd, . <https://linkerd.io>.
- 1133 [28] Meyer, A., Pufahl, L., Batoulis, K., Fahland, D., Weske, M., 2015. Au-
1134 tomating data exchange in process choreographies. Inf. Syst. 53, 296–329.
1135 URL <https://doi.org/10.1016/j.is.2015.03.008>
- 1136 [29] Murex, . <https://www.murex.com>.

- 1137 [30] Nikaj, A., Weske, M., Mendling, J., 2019. Semi-automatic derivation of
1138 restful choreographies from business process choreographies. *Software and*
1139 *System Modeling* 18 (2), 1195–1208.
1140 URL <https://doi.org/10.1007/s10270-017-0653-2>
- 1141 [31] OMG, B. P. M., Notation (BPMN), V. ., 2011.
1142 <http://www.omg.org/spec/BPMN/2.0/>.
- 1143 [32] Ongaro, D., Ousterhout, J. K., 2014. In search of an understandable consen-
1144 sus algorithm. In: 2014 USENIX Annual Technical Conference, USENIX
1145 ATC '14, Philadelphia, PA, USA, June 19-20, 2014. pp. 305–319.
- 1146 [33] Pnueli, A., 1977. The temporal logic of programs. In: 18th Annual Sym-
1147 posium on Foundations of Computer Science, Providence, Rhode Island,
1148 USA, 31 October - 1 November 1977. IEEE Computer Society, pp. 46–57.
- 1149 [34] Preda, M. D., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J., Apr.
1150 2017. Dynamic choreographies: Theory and implementation. *Logical Meth-*
1151 *ods in Computer Science* Volume 13, Issue 2.
1152 URL <https://lmcs.episciences.org/3263>
- 1153 [35] Scalas, A., Yoshida, N., 2019. Less is more: multiparty session types revis-
1154 ited. *PACMPL* 3 (POPL), 30:1–30:29.
1155 URL <https://doi.org/10.1145/3290343>
- 1156 [36] Tuosto, E., Guanciale, R., 2018. Semantics of global view of choreographies.
1157 *J. Log. Algebr. Meth. Program.* 95, 17–40.
1158 URL <https://doi.org/10.1016/j.jlamp.2017.11.002>
- 1159 [37] Vallecillo, A., Vasconcelos, V. T., Ravara, A., 2006. Typing the behavior of
1160 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