

Connectors meet Choreographies

Farhad Arbab¹ Luís Cruz-Filipe² Sung-Shik Jongmans^{3,4}
 Fabrizio Montesi²

¹ Centrum Wiskunde & Informatica, the Netherlands

² Dept. Mathematics and Computer Science, Univ. Southern Denmark

³ Dept. Computer Science, Open University of the Netherlands

⁴ Dept. Computing, Imperial College London

Abstract

We present Cho-Reo-graphies (CR), a new language model that unites two powerful programming paradigms for concurrent software based on communicating processes: Choreographic Programming and Exogenous Coordination. In CR, programmers specify the desired communications among processes using a *choreography*, and define how communications should be concretely animated by *connectors* given as constraint automata (e.g., synchronous barriers and asynchronous multi-casts). CR is the first choreography calculus where different communication semantics (determined by connectors) can be freely mixed; since connectors are user-defined, CR also supports many communication semantics that were previously unavailable for choreographies. We develop a static analysis that guarantees that a choreography in CR and its user-defined connectors are compatible, define a compiler from choreographies to a process calculus based on connectors, and prove that compatibility guarantees deadlock-freedom of the compiled process implementations.

1 Introduction

Background. Programming concurrent software is hard: given the specification of the desired interactions among processes, implementation is error-prone even for experienced programmers [35, 38]. The challenge of concurrent programming has driven decades of research on new programming models. A particularly fruitful idea was to provide a native language abstraction for interaction, rather than modelling it as a side-effect. Two research lines in particular were built upon this idea, but following very different directions.

The first research line is that on *Choreographic Programming* [39]. Central to Choreographic Programming is the *choreography*, a programming artefact that specifies a concurrent system in terms of the interactions among its constituent processes, by using an “Alice and Bob” notation that disallows writing mismatched I/O actions (e.g., a send without a corresponding receive). Through *EndPoint Projection* (EPP), choreographies can be compiled to faithful and deadlock-free implementations in process languages [17].

Previous work studied models for choreographies with different interaction semantics (e.g., synchronous, asynchronous, multi-cast). Common to all existing models of Choreographic Programming is the fact that the nature of interactions is “hardcoded”: each model proposes new syntax and semantics, so results have to be proven from scratch every time and cannot be combined. For example, none of the existing models supports mixing synchronous with asynchronous interactions within the same choreography. This is a serious limitation, which raises the challenge of finding a unifying framework. Furthermore, there is still no indication of how other interesting interaction semantics (e.g., barriers) can be introduced to choreographies.

The second research line is that on *Exogenous Coordination* [2, 3, 34], where interaction protocols and process code are developed separately. Process programs are then modularly composed

with protocols, given as *connectors*, which dictate interactions by accepting/offering messages from/to processes. Connectors offer an elegant way of programming different semantics for interactions, starting from basic instances (e.g., synchronous and asynchronous point-to-point channels) and then composing them to create more sophisticated protocols.

The limitation of Exogenous Coordination is that we do not have a global system view when composing processes with connectors: programmers cannot define the intended flow of information among processes when a connector is used many times, or if many connectors are used in different parts of a system. Hence, incompatibilities between process code and connectors may cause deadlocks, which cannot happen in Choreographic Programming. The two paradigms thus have complementary strengths.

Contribution. We develop the first integration of the best aspects of Choreographic Programming and Exogenous Coordination: a new calculus of choreographies, called *Cho-Reo-graphies* (CR), whereby the process interactions specified in a choreography are animated by arbitrary, user-defined, connectors based on the Exogenous Coordination language Reo. CR allows for mixing different connectors in the same choreography, making it for the first time possible to write choreographies where different interactions can have different communication semantics. Furthermore, by tapping into the expressivity of connectors, we can endow choreographies with hitherto unexplored communication semantics, such as alternators or barriers. This makes CR more expressive and a generalisation of existing models of Choreographic Programming. Through EPP, choreographies can be compiled into concurrent implementations in a process language. In these implementations, the same connectors as in the original choreography animate interactions among processes. We show that these processes are deadlock-free, provided that the original choreography is compatible with the connectors.

Mixing different communication semantics (i.e., connectors) in choreographies produces new challenges of formalisation and decidability. Solving these is our main technical contribution. On the formalisation front, we have to balance expressiveness and comprehensibility: the formal semantics of the calculus should be easy enough to explain and understand, without sacrificing the expressive power of connectors. We address this challenge with a new labelled reduction semantics for choreographic interactions, where labels act as the interface with connectors. On the decidability front, we prove that (1) deadlock-freedom is generally undecidable in our calculus, but (2) we can establish deadlock-freedom for a large subset of the language. Our proof of (1) shows that undecidability is a direct consequence of the expressive power that connectors introduce to choreographies. We address (2) by designing a new decidable static analysis (compatibility) that is made possible by the careful design of labelled reductions for interactions.

By leveraging existing work, distributed implementations of Reo connectors can be automatically generated and deployed on distributed systems (e.g., in Scala [41, 42] or Java [31]); our approach is compatible with this work. As such, a practical tool based on CR can in principle be built on top of existing code generators for distributed implementations of Reo connectors. In general, we believe that our results represent the beginning of an interesting research line on concurrent programming; we discuss future directions in § 6.

Structure. We motivate our work with an example in § 2, and introduce connectors in § 3. CR is described in § 4 together with results on deadlock-freedom. In § 5 we introduce the target process language for EndPoint Projection and prove correctness of synthesised process implementations. We conclude in § 6, discussing directions in which this work can be extended. § 6 concludes, discussing directions in which this work can be extended.

Related work. We already covered the main references to previous work and how it falls short of serving our aim. We briefly recap related work.

Choreographies have been studied in different settings with fixed communication semantics, including synchronous [15, 22, 32], asynchronous point-to-point [17, 24, 27, 40], one-to-many [18, 20], and many-to-one [16, 36]. In our model, these semantics are simply instances of what we

can do. But since we tap into the generality of Reo connectors, we can also do more (e.g., we illustrate how to use barriers). CR is not the first model where choreographies may deadlock: this is common for settings with realistic communication semantics (e.g., [23, 37]). However, it is the first model with arbitrary communication semantics. Our development also extends the line of work on out-of-order execution for choreographies, initiated in [17], where non-interfering interactions may proceed concurrently. This style allows for more safe behaviours in the semantics of choreographies (by swapping non-interfering communications), which the programmer gets for free (concurrency is inferred). As in many other works, out-of-order execution also simplifies our syntax: we do not need to provide for a parallel operator in choreographies, since most parallel behaviour is already captured by out-of-order execution (cf. [17] for details). As in previous work, CR supports asynchronous behaviour without requiring the programmer to reason about it in choreographies: communications are still specified as atomic interactions, which may be asynchronously reduced in a safe way (cf. [17, 26, 40]).

In Exogenous Coordination, interaction protocols for communicating processes, called connectors, are programmed separately from the internal code of each process. This enables a compositional approach for developing protocols, where complex protocols can be built by assembling simpler ones. Exogenous Coordination has been studied extensively over the last two decades [2, 3, 34]. Examples of models of Exogenous Coordination are the algebras of connectors [11, 12], the algebra of stateless connectors [13], and constraint automata [8]; examples of languages are (interactions in) BIP [9, 10], Ptolemy [14, 43], and Reo [4, 5].

2 Motivating Example & Approach

We present an example to introduce the concept of choreographies and the problem we are interested in studying. This example will be used as running example throughout the whole article.

Example 1 (Book sale). *Alice (a) wants to buy a book from seller Carol (c), facilitated by a bank (b) and a shipper (s). First, Alice sends the title of the book to Carol. Carol then replies to Alice with the price of the book. If Alice is happy with the price, she notifies Carol, the bank, and the shipper that the purchase proceeds. Alice subsequently sends the money to the bank (who transfers it to Carol’s account), and Carol sends the book to the shipper (who dispatches it to Alice). The choreography for this scenario looks as follows:*

1. $a\langle\text{title}\rangle \rightarrow c; c\langle\text{price}\rangle \rightarrow a;$
2. $\text{if } a.\text{happy} \text{ then } (a \rightarrow c[\text{ok}]; a \rightarrow b[\text{ok}]; a \rightarrow s[\text{ok}]; a\langle\text{money}\rangle \rightarrow b; c\langle\text{book}\rangle \rightarrow s)$
 $\quad \text{else } (a \rightarrow c[\text{ko}]; a \rightarrow b[\text{ko}]; a \rightarrow s[\text{ko}]) \quad \square$

In previous choreography models, the nature of the interactions is fixed: depending on the model, these interactions are either all synchronous or all asynchronous. This is not flexible enough, because requirements may be different for each interaction.

Example 2 (Book sale). *The book sale scenario has the following requirements:*

- *Because there are no strict timing constraints, it is reasonable for Alice and Carol to communicate asynchronously in line 1.*
- *Because the same label (ok or ko) is sent from Alice to Carol, the bank, and the shipper, it makes sense to combine these communications in a multi-cast.*
- *It is better for Alice to send her money to the bank as late as possible (e.g., because she receives interest on her money). Therefore, Alice does not want to send her money until she knows the others have received her ok-label. Thus, the multi-cast of this label should be synchronous (i.e., handshake between Alice, Carol, the bank, and the shipper).*

- *Alice and Carol may not trust each other: Alice does not want to send money before Carol has sent the book, and vice versa. To resolve this impasse, Alice and Carol should synchronously (barrier-like) send money and book, to ensure each of them holds her end of the bargain; the receives by the bank and the shipper may subsequently proceed asynchronously.* \square

As we show in the next sections, our model is powerful enough to express all these interactions (and more). The key idea is to tag interactions with the name of the particular connector through which they transpire. For instance, a value communication from p to q through a synchronous channel `sync` is expressed as $p \rightarrow q \text{ thru sync}$, while the same communication through an asynchronous channel `async` is expressed as $p \rightarrow q \text{ thru async}$. If all interactions in a choreography transpire through the same (type of) connector, as in all existing choreography approaches, tags become redundant and can be omitted.

3 Reo and Constraint Automata

We view processes in a concurrent system as black boxes with interfaces consisting of *ports*. For a process to send (receive) a message to (from) another process, it performs an output action (input action) on one of its own output ports (input ports), but without specifying a receiver (sender). Instead, a separate connector, connected to the ports of the processes, decides how messages flow from senders' output ports to receivers' input ports.

Reo [4, 5] is a graphical, dataflow-inspired language for connectors. As we focus primarily on semantics in the rest of this paper, we only present *Reo*'s semantic formalism: *constraint automata* [8]. The transitions of a constraint automaton model possible synchronous message flows (i.e., interactions) through a connector in a particular state. Constraint automata are parametrised over a language of constraints Φ to specify transition labels. Depending on the required level of expressiveness, different instantiations of Φ may be considered. In this work, we consider $\Phi_{P,M}$ to be a language of constraints over two sets P and M , of ports and *memory cells* (storage space local to a connector). Constraints are finite sets of formulas of the form $p_1 \rightarrow p_2$ (port p_1 passes a message to port p_2), $m_1 \rightarrow m_2$ (cell m_1 passes a message to cell m_2), $p \rightarrow m$ (port p passes a message to cell m), and $m \rightarrow p$ (cell m passes a message to port p).¹ Furthermore, we require that all terms on the right-hand side of formulas in the same constraint in a transition label be distinct, to ensure that every port/cell is assigned a unique message (e.g., $p_1 \rightarrow m \wedge p_2 \rightarrow m$ is forbidden, but $p \rightarrow m_1 \wedge p \rightarrow m_2$ is allowed).

Definition 1. *A constraint automaton is a tuple $(S, P, M, \rightarrow, s_0, \mu_0)$, where S is a finite set of states, P is a finite set of ports, M is a finite set of memory cells, $\rightarrow \subseteq S \times \Phi_{P,M} \times S$ is a transition relation, $s_0 \in S$ is an initial state, and μ_0 is an initial memory snapshot mapping the memory cells in M to their initial content.*

A transition $(s, \phi, s') \in \rightarrow$, which we write $s \xrightarrow{\phi} s'$ for short, means that, from state s , a subset of the ports in P can interact according to ϕ and the automaton goes to state s' . Constraints in $\Phi_{P,M}$ also control the evolution of memory snapshots as transitions are made. Instead of formalizing this separately (e.g., in terms of runs and languages [29]), we combine it directly in the semantics of our choreographies, in § 4.

In this work, we restrict ourselves to a subset of constraint automata that satisfy two additional assumptions on their transitions. First, the occurrences of each port in transition labels are either all as sender (the port occurs only on the left-hand side of constraints) or all as receiver (it occurs only on the right-hand side of constraints); this constraint is imposed on the whole automaton. For instance, a constraint automaton cannot have two transitions where one is labelled by $p_1 \rightarrow p_2$ and the other by $p_2 \rightarrow p_3$. Secondly, the transition relation of each automaton is deterministic on the ports used, i.e., for any given state s , if there are two distinct transitions $s \xrightarrow{\phi_1} s_1$ and $s \xrightarrow{\phi_2} s_2$, then the sets of ports used in ϕ_1 and ϕ_2 must be distinct (but they can overlap, or one be a strict subset of the other). The first assumption simplifies defining the semantics of choreographies in

¹We can formalise Φ as an equational theory as in [8, 29], but this is unnecessary for our development.

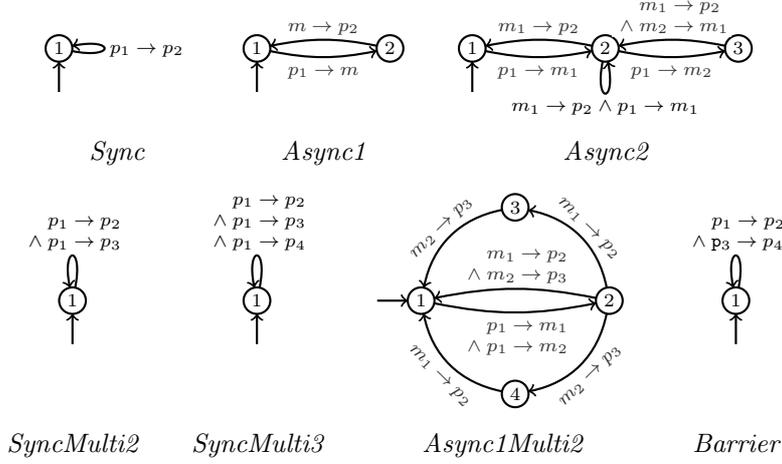


Figure 1: Example constraint automata.

§ 4; the second assumption ensures that this semantics is deterministic, in line with previous work on choreographies.

Port automata are minimalistic, meaning that a port automaton defines only the orders in which ports may be used. On the contrary, a port automaton says nothing about the actual messages being exchanged through ports. Notably, a port automaton does not define which particular message (sent through an output port by a sender) a receiver receives (through an input port). Port automata, thus, *underspecify* connector behaviour.

Example 3. Fig. 1 shows example constraint automata for useful connectors. *Sync* models a synchronous channel, which indefinitely lets two processes synchronously send and receive a message through ports p_1 and p_2 . *Async1* models an asynchronous channel with a 1-capacity buffer (using a memory cell m). Indefinitely, first, this connector lets a process send a message through port p_1 ; subsequently, it lets a process receive the message through port p_2 . *Async2* models an asynchronous channel with a 2-capacity buffer. *SyncMulti2* and *SyncMulti3* model synchronous multi-cast connectors for two and three receivers. Indefinitely, this connector lets three processes synchronously send and receive a message, from port p_1 to ports p_2 and p_3 . *Async1Multi2* models an asynchronous multi-cast connector for two receivers. Indefinitely, first, the connector lets a process send a message (lower middle transition); then, it lets two processes receive the message, either one after the other (top or bottom transitions) or simultaneously (upper middle transition). *Barrier* models a barrier send/receive connector. Indefinitely, this connector lets two processes synchronously send and receive a message through ports p_1 and p_2 , while it synchronously lets processes synchronously send and receive another message through ports p_3 and p_4 . \square

Choreography programmers can model connectors by explicitly defining constraint automata. Alternatively, programmers can model connectors by *composing* constraint automata from basic primitive constraint automata, using a synchronous product operator [8]. A significant advantage of this latter approach is that there exists an intuitive and user-friendly graphical syntax for constraint automaton product expressions. In this graphical syntax, called *Reo* [4, 5], programmers draw constraint automaton product expressions as data-flow graphs between (ports of) processes. As an example, Figure 2 shows *Reo* connectors for the constraint automata in Figure 1. *Reo* conveniently hides from choreography programmers the intimidating act of explicitly composing constraint automata, without sacrificing generality: *Reo* is complete for constraint automata (under the instantiation of Φ considered in this paper), meaning that every constraint automaton can be expressed as a *Reo* connector [6, 7]. Moreover, tooling exists to animate flows of messages through *Reo* connectors (<http://reoproject.cwi.nl>).

We use (constraint) automata and (Reo) connectors interchangeably: at the semantics level, we use automata, while at the syntax level, we use connectors.

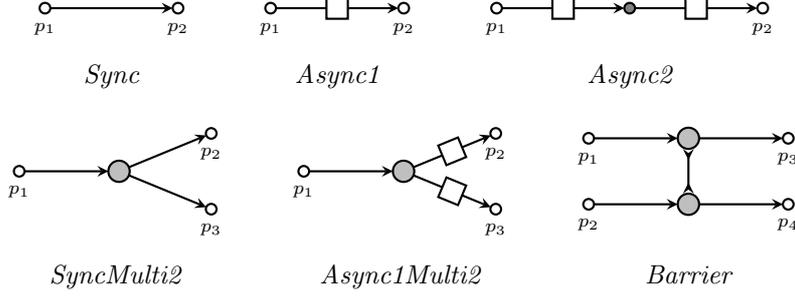


Figure 2: Example Reo connectors for the constraint automata in Fig. 1.

$$\begin{aligned}
C &::= \tilde{\eta} \text{ thru } \gamma; C \mid \text{if } p.e \text{ then } C_1 \text{ else } C_2 \mid \text{def } X = C_2 \text{ in } C_1 \mid X \mid \mathbf{0} \\
\eta &::= p\langle e \rangle \rightarrow q.x \mid p \rightarrow q[\ell] \mid \boxed{q.x?v} \mid \boxed{q[\ell]}
\end{aligned}$$

Figure 3: Cho-Reo-graphies, syntax. The boxed terms are runtime terms, necessary for defining the semantics, but which are not meant to be used by programmers.

4 Cho-Reo-graphies

We now present *Cho-Reo-graphies* (CR): our choreography calculus that combines choreographies with Reo connectors. A choreography describes the behaviour of a set of processes. For simplicity, we assume that values are untyped; treating value types is straightforward and analogous to [15, 17, 23].

The syntax of CR is displayed in Fig. 3. We range over choreographies using C, C', \dots ; over processes using p, q, \dots ; over interactions using η, η', \dots (and over sets of them using $\tilde{\eta}, \tilde{\eta}', \tilde{\eta}_1, \dots$); over connector names using γ, γ', \dots ; over procedure names using X ; over (side-effect free) expressions using e, e', \dots ; over values using v, v', \dots ; and over selection labels using ℓ, ℓ', \dots . Each process p owns a finite set of local variables $\text{var}_p = \{x_1, \dots, x_n\}$, and expressions are assumed to be inductively defined including var_p as base cases.

We comment briefly on label selections. It is standard practice in Choreographic Programming to distinguish value communications, which are used to exchange data, from label selections, which are used to propagate decisions regarding control flow. In the choreography in Ex. 1, Alice uses label selections (*ok* or *ko*) to communicate her choice of whether to buy the book or not to Carol and the bank. Although this can be encoded using value communications [22], it is useful to distinguish them, as they are usually treated differently in implementations. Also, as discussed in § 5, label selections are instrumental in generating process implementations automatically. The syntax of labels is unspecified.

We write $p\langle e \rangle \rightarrow \{q_1.x_1, \dots, q_n.x_n\}$ (value multicast) and $p \rightarrow \{q_1, \dots, q_n\}[\ell]$ (label multicast) to abbreviate $\{p\langle e \rangle \rightarrow q_1.x_1, \dots, p\langle e \rangle \rightarrow q_n.x_n\}$ and $\{p \rightarrow q_1[\ell], \dots, p \rightarrow q_n[\ell]\}$, respectively. Also, we write $p\langle x \rangle \rightarrow q$ for $p\langle x \rangle \rightarrow q.x$ (i.e., if sender p sends the value in local variable x , and if receiver q stores the received value in a local variable with the same name, we omit the name). If $\tilde{\eta}$ is a singleton, we omit curly braces.

Example 4 (Book sale). *The choreography for our running example (Ex. 1) can be written as follows in CR.*

1. $a\langle \text{title} \rangle \rightarrow c \text{ thru } a2c;$
2. $c\langle \text{price} \rangle \rightarrow a \text{ thru } c2a;$
3. $\text{if } a.\text{happy} \text{ then } (a \rightarrow \{c, b, s\}[\text{ok}] \text{ thru } a2cbs;$
4. $\quad \{a\langle \text{money} \rangle \rightarrow b, c\langle \text{book} \rangle \rightarrow s\} \text{ thru } ac2bs; \mathbf{0})$
5. $\quad \text{else } (a \rightarrow \{c, b, s\}[\text{ko}] \text{ thru } a2cbs; \mathbf{0})$ □

Our notation suggests that Alice multi-casts *ok* (or *ko*) to Carol, the bank, and the shipper. It is important to understand, though, that this notation is really just syntactic sugar. Ultimately, only the behaviour of connector `a2cbs` determines how exactly the three communications represented by $a \rightarrow \{c, b, s\}[\ell]$ transpire.

The semantics of most terms is standard. In `if p.e then C1 else C2`, process `p` evaluates expression `e`; if this results in *true*, the choreography proceeds as `C1`, and otherwise, as `C2`. In `def X = C2 in C1`, procedure `X` is defined as `C2`; it can then be invoked as `X` from both `C1` and `C2`. `0` indicates successful termination.

The interesting new part is the semantics of `ñ thru γ`, which informally specifies that all communications in `ñ` occur through connector `γ`. More precisely, in `p⟨e⟩ → q.x thru γ`, process `p` (the *sender*) evaluates expression `e` and offers the resulting value to connector `γ`. The connector eventually accepts and passes it to process `q` (the *receiver*), who stores it in its local variable `x`. The behaviour of `p → q[ℓ] thru γ` is similar, except that `p` offers a label instead of a normal value. Label selections do not change the state of the receiving process; their role in synthesizing process implementations is discussed in § 5.

The boxed terminals in Fig. 3 are *runtime terms*, meant to be used only in defining the formal semantics and not by programmers. They arise because connectors may have a multi-step semantics (they do not necessarily synchronise sends with receives). In particular, `q.x?v` is obtained when a communication `p⟨e⟩ → q.x` is partially executed, and `p` has already sent its value, but `q` has not yet received it; `q[ℓ]` arises similarly – see Ex. 8 below.

For the semantics of `ñ thru γ` to be well-defined, `ñ` must satisfy two conditions. First, all interactions in `ñ` must have *distinct receivers*: if `p1⟨e1⟩ → q1.x1 ∈ ñ` and `p2⟨e2⟩ → q2.x2 ∈ ñ`, then `q1 ≠ q2`. This ensures that the value received by a receiver is uniquely defined. Second, all sends must be *consistent*: if a process `p` is involved in multiple interactions in the same set, then they are either all communications of the same expression or all selections of the same label.

Example 5 (Book sale). *In the context of our running example, the following interaction sets are allowed (✓) or disallowed (✗) by our conditions for distinct receivers and consistent sends.*

- ✓ $\{a\langle money \rangle \rightarrow b, c\langle book \rangle \rightarrow s\}$ – Alice sends money to the bank, while Carol sends a book to the shipper (distinct receivers; consistent sends).
- ✗ $\{a\langle money \rangle \rightarrow b, c\langle money \rangle \rightarrow b\}$ – Both Alice and Carol send money to the bank, but the bank can receive only from one sender at a time (receivers are not distinct).
- ✗ $\{c\langle price \rangle \rightarrow a, c\langle book \rangle \rightarrow s\}$ – Carol sends both the price to Alice and the book to the shipper, but Carol can send only one value at a time (sends are not consistent).
- ✓ $\{a \rightarrow b[ok], a \rightarrow c[ok], a \rightarrow s[ok]\}$ – Alice sends label *ok* to Carol, the bank, and the shipper (distinct receivers; consistent sends). □

Example 6 (Book sale). *Returning to our running example, suppose that Alice wants to buy the book titled Foo.*

Let \mathcal{G} denote the connector mapping in Ex. 7, let C' denote lines 2–3 in Ex. 4, let σ_0 denote the initial choreography state function such that $\sigma_0(a.title) = \text{"foo"}$, let $\sigma'_0 = \sigma_0[c.title \mapsto \text{"foo"}]$, and let $\mathcal{A}_0 = \{a2c \mapsto \langle 1, \{m \mapsto \perp\} \rangle\} \cup \mathcal{A}_0^{rest}$ denote the initial automaton state function, where:

$$\mathcal{A}_0^{rest} = \{c2a \mapsto \langle 1, \{m \mapsto \perp\} \rangle, a2cbs \mapsto \langle 1, \emptyset \rangle, ac2bs \mapsto \langle 1, \emptyset \rangle\}$$

Initially, thus, all connectors are in their initial state (state 1). Furthermore, connectors `a2c` and `c2a` have an empty memory cell ($m \mapsto \perp$); connectors `a2cbs` and `ac2bs` have no memory cells (memory snapshot \emptyset). By rule $\lfloor C \rfloor Com$ (presented after this example), the choreography in Ex. 4

reduces under \mathcal{G} as follows:

$$\begin{array}{l}
\mathbf{a}(\mathit{title}) \rightarrow \mathbf{c} \mathit{thru} \mathbf{a2c}; C', \quad \sigma_0, \quad \{\mathbf{a2c} \mapsto \langle 1, \{m \mapsto \perp\} \rangle\} \cup \mathcal{A}_0^{\mathit{rest}} \\
\rightsquigarrow_{\mathcal{G}} \boxed{\text{In state 1, according to } \mathcal{G}, \text{ connector } \mathbf{a2c} \text{ only has a transition that allows Alice to send (asyn-} \\
\text{chronously) to Carol. By performing such a send, Alice enables the choreography to make a} \\
\text{reduction, in which the first half of the communication completes; the (asynchronous) receive re-} \\
\text{mains. In the same step, } \mathbf{a2c} \text{ moves to state 2, and the value sent by Alice is stored in } \mathbf{a2c}'\text{s} \\
\text{internal memory cell (} \mathit{title} \text{ evaluates to "foo", based on } \sigma_0 \text{).}} \\
\mathbf{c}.\mathit{title}?\text{"foo"} \mathit{thru} \mathbf{a2c}; C', \quad \sigma_0, \quad \{\mathbf{a2c} \mapsto \langle 2, \{m \mapsto \text{"foo"}\} \rangle\} \cup \mathcal{A}_0^{\mathit{rest}} \\
\rightsquigarrow_{\mathcal{G}} \boxed{\text{In state 2, according to } \mathcal{G}, \text{ connector } \mathbf{a2c} \text{ only has a transition that allows Carol to receive (asyn-} \\
\text{chronously) from Alice. By performing such a receive, Carol enables the choreography to make a} \\
\text{reduction in which the whole communication completes. In the same step, } \mathbf{a2c} \text{ moves to state 1} \\
\text{(the internal memory cell is not cleared).}} \\
\emptyset \mathit{thru} \mathbf{a2c}; C', \quad \sigma'_0, \quad \{\mathbf{a2c} \mapsto \langle 1, \{m \mapsto \text{"foo"}\} \rangle\} \cup \mathcal{A}_0^{\mathit{rest}}
\end{array}$$

□

4.1 Formal semantics

The semantics of CR is a reduction semantics parametrised over a *connector mapping*: a function \mathcal{G} from connector names to automata. Intuitively, $\mathcal{G}(\gamma)$ denotes the automaton that models connector γ used in the choreography; the set P of ports in each $\mathcal{G}(\gamma)$ is simply a one-to-one mapping (i.e., a renaming of) the set of processes that use the connector.²

Example 7 (Book sale). *The connector names in Ex. 4 are $\mathbf{a2c}$, $\mathbf{c2a}$, $\mathbf{a2cbs}$, and $\mathbf{ac2bs}$. Thus, the requirements in Ex. 2 give rise to the following connector mapping:*

$$\left\{ \begin{array}{ll} \mathbf{a2c} \mapsto \mathit{Async1}[\mathbf{a}/p_1, \mathbf{c}/p_2], & \mathbf{c2a} \mapsto \mathit{Async1}[\mathbf{c}/p_1, \mathbf{a}/p_2], \\ \mathbf{a2cbs} \mapsto \mathit{SyncMulti3}[\mathbf{a}/p_1, \mathbf{b}/p_2, \mathbf{c}/p_3, \mathbf{s}/p_4], & \mathbf{ac2bs} \mapsto \mathit{Barrier}[\mathbf{a}/p_1, \mathbf{b}/p_2, \mathbf{c}/p_3, \mathbf{s}/p_4] \end{array} \right\}$$

where $\mathit{Async1}[\mathbf{a}/p_1, \mathbf{c}/p_2]$ denotes automaton $\mathit{Async1}$ in Fig. 1, with \mathbf{a} substituted for p_1 , and \mathbf{c} for p_2 (and likewise in the other mappings). Under this connector mapping, thus, Alice and Carol communicate via asynchronous channels ($\mathbf{a2c}$ and $\mathbf{c2a}$) to exchange *title* and *price*; Alice, Carol, the bank, and the shipper communicate via synchronous multi-cast ($\mathbf{a2cbs}$) to exchange *ok* or *ko*, and via barrier sends/receives ($\mathbf{ac2bs}$) to exchange *money* and *book*. □

Remark 1. *The book sale scenario illustrates an important design decision, namely the separation between intention and realisation: a choreography defines what interactions are intended (e.g., communications of the money from Alice to the Bank and the book from Carol to the shipper), while the connectors define how these communications are realised (e.g., synchronously or asynchronously). As a result, every interaction has to be expressed in two places, serving two complementary purposes: as “specifications” in the choreography and as “implementations” in the connectors (automaton transitions). As usual, implementations should respect specifications; we address this in § 4.3.* □

The reduction relation for CR under a given \mathcal{G} is denoted as $\rightsquigarrow_{\mathcal{G}}$; it ranges over triples C, σ, \mathcal{A} , where C is a choreography, σ is a choreography state function (mapping each process to a mapping of its variables to values, i.e., $\sigma(\mathbf{p}.x)$ is the value stored at variable x in process \mathbf{p}), and \mathcal{A} is an automaton state function (mapping each connector name γ in the domain of \mathcal{G} to a pair $\langle s, \mu \rangle$ of the state and memory snapshot of the automaton $\mathcal{G}(\gamma)$). Before introducing the formal rule for communications, we give an example that discusses the intuition.

Example 8 (Book sale). *Returning to our running example, suppose that Alice wants to buy the book titled *Foo*.*

²This means that each process can interact at most through one port in each automaton.

Let \mathcal{G} denote the connector mapping in Ex. 7, let C' denote lines 2–3 in Ex. 4, let σ_0 denote the initial choreography state function such that $\sigma_0(\mathbf{a}.title) = \text{"foo"}$, let $\sigma'_0 = \sigma_0[\mathbf{c}.title \mapsto \text{"foo"}]$, and let $\mathcal{A}_0 = \{\mathbf{a}2\mathbf{c} \mapsto \langle 1, \{m \mapsto \perp\}\rangle\} \cup \mathcal{A}_0^{rest}$ denote the initial automaton state function, where:

$$\mathcal{A}_0^{rest} = \{\mathbf{c}2\mathbf{a} \mapsto \langle 1, \{m \mapsto \perp\}\rangle, \mathbf{a}2\mathbf{c}\mathbf{b}\mathbf{s} \mapsto \langle 1, \emptyset\rangle, \mathbf{a}\mathbf{c}2\mathbf{b}\mathbf{s} \mapsto \langle 1, \emptyset\rangle\}$$

Initially, thus, all connectors are in their initial state (state 1). Furthermore, connectors $\mathbf{a}2\mathbf{c}$ and $\mathbf{c}2\mathbf{a}$ have an empty memory cell ($m \mapsto \perp$); connectors $\mathbf{a}2\mathbf{c}\mathbf{b}\mathbf{s}$ and $\mathbf{a}\mathbf{c}2\mathbf{b}\mathbf{s}$ have no memory cells (memory snapshot \emptyset). By rule $[C|Com]$ (presented after this example), the choreography in Ex. 4 reduces under \mathcal{G} as follows:

$$\begin{array}{l} \mathbf{a}\langle title \rangle \rightarrow \mathbf{c} \text{ thru } \mathbf{a}2\mathbf{c}; C', \quad \sigma_0, \quad \{\mathbf{a}2\mathbf{c} \mapsto \langle 1, \{m \mapsto \perp\}\rangle\} \cup \mathcal{A}_0^{rest} \\ \rightsquigarrow_{\mathcal{G}} \boxed{\text{In state 1, according to } \mathcal{G}, \text{ connector } \mathbf{a}2\mathbf{c} \text{ only has a transition that allows Alice to send (asyn-} \\ \text{chronously) to Carol. By performing such a send, Alice enables the choreography to make a} \\ \text{reduction, in which the first half of the communication completes; the (asynchronous) receive re-} \\ \text{mains. In the same step, } \mathbf{a}2\mathbf{c} \text{ moves to state 2, and the value sent by Alice is stored in } \mathbf{a}2\mathbf{c}'\text{s} \\ \text{internal memory cell (title evaluates to "foo", based on } \sigma_0\text{).}} \\ \mathbf{c}.title? \text{"foo"} \text{ thru } \mathbf{a}2\mathbf{c}; C', \quad \sigma_0, \quad \{\mathbf{a}2\mathbf{c} \mapsto \langle 2, \{m \mapsto \text{"foo"}\}\rangle\} \cup \mathcal{A}_0^{rest} \\ \rightsquigarrow_{\mathcal{G}} \boxed{\text{In state 2, according to } \mathcal{G}, \text{ connector } \mathbf{a}2\mathbf{c} \text{ only has a transition that allows Carol to receive (asyn-} \\ \text{chronously) from Alice. By performing such a receive, Carol enables the choreography to make a} \\ \text{reduction in which the whole communication completes. In the same step, } \mathbf{a}2\mathbf{c} \text{ moves to state 1} \\ \text{(the internal memory cell is not cleared).}} \\ \emptyset \text{ thru } \mathbf{a}2\mathbf{c}; C', \quad \sigma'_0, \quad \{\mathbf{a}2\mathbf{c} \mapsto \langle 1, \{m \mapsto \text{"foo"}\}\rangle\} \cup \mathcal{A}_0^{rest} \end{array}$$

□

These intuitions are captured in the rule for communications $[C|Com]$, which is the key rule defining $\rightsquigarrow_{\mathcal{G}}$. We discuss this rule in detail.

$$\frac{\mathcal{A}(\gamma) = \langle s, \mu \rangle \quad \tilde{\eta}, \sigma, \mu \xrightarrow{\phi} \tilde{\eta}', \sigma', \mu' \quad s \xrightarrow{\phi}_{\gamma} s'}{\tilde{\eta} \text{ thru } \gamma; C, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} \tilde{\eta}' \text{ thru } \gamma; C, \sigma', \mathcal{A}[\gamma \mapsto \langle s', \mu' \rangle]} [C|Com]$$

Rule $[C|Com]$ allows some of the communications in $\tilde{\eta}$ to reduce as long as the state of the automaton corresponding to connector γ allows it. The rule reads: under \mathcal{G} , triple $\tilde{\eta} \text{ thru } \gamma; C, \sigma, \mathcal{A}$ can reduce if automaton $\mathcal{G}(\gamma)$ can fire a transition out of its current state that is compatible with the interactions specified in $\tilde{\eta}$.

More formally, the first premise of this rule retrieves the current state s and memory snapshot μ of the connector γ controlling the communication. In the second premise, the labelled reduction $\tilde{\eta}, \sigma, \mu \xrightarrow{\phi} \tilde{\eta}', \sigma', \mu'$ (defined below) states that reducing $\tilde{\eta}$ to $\tilde{\eta}'$ transforms the state of processes σ into σ' and the memory snapshot of the connector μ into μ' . The label ϕ represents the actions executed in this reduction. The third premise checks that these actions are allowed by the automaton, by checking that ϕ labels an outgoing transition of s .

The rules defining labelled reductions $\tilde{\eta}, \sigma, \mu \xrightarrow{\phi} \tilde{\eta}', \sigma', \mu'$ are given in Fig. 4. They are obtained by considering the different possible cases for terms $\tilde{\eta}$ and matching them to appropriate constraints ϕ . Let $e \Downarrow_{\mathbf{p}}^{\sigma} v$ denote that expression e evaluates to value v under σ (i.e., after substituting every free variable x in e by $\sigma(\mathbf{p}.x)$). In rule $[C|SyncVal]$, an entire communication $\mathbf{p}\langle e \rangle \rightarrow \mathbf{q}.x$ is executed in one step. Accordingly, the label $\mathbf{p} \rightarrow \mathbf{q}$ denotes that the automaton should support a synchronous communication between \mathbf{p} and \mathbf{q} . The state of the receiver \mathbf{q} is updated with the value sent by \mathbf{p} . Rule $[C|SendVal]$ applies in the case where the message from \mathbf{p} should be stored in a memory cell m of the automaton (label $\mathbf{p} \rightarrow m$). This is used for asynchronous communications, where the message is received later on by the receiver. In the reductum, we keep a runtime term signalling that the receiver is still waiting to receive the message ($\mathbf{q}.x?v$). This kind of runtime terms is handled by $[C|RecvVal]$, whose label specifies that \mathbf{q} should receive the message stored in some memory cell m . Its premise checks that the value \mathbf{q} is expecting to receive (v , defined in the choreography term) is the one stored in m .³ Rules $[C|SyncSel]$, $[C|SendSel]$ and $[C|RecvSel]$

³In other words, the automaton delivers the messages as specified in the choreography.

$$\begin{array}{c}
\frac{e \downarrow_p^\sigma v}{\{\mathbf{p}\langle e \rangle \rightarrow \mathbf{q}.x\}, \sigma, \mu \xrightarrow{\mathbf{p} \rightarrow m} \mathbf{q}.x?v, \sigma, \mu[m \mapsto v]} \text{ [C|SendVal]} \quad \frac{\mu(m_1) = v}{\emptyset, \sigma, \mu \xrightarrow{m_1 \rightarrow m_2} \emptyset, \sigma, \mu[m_2 \mapsto v]} \text{ [C|Mem]} \\
\frac{\mu(m) = v}{\{\mathbf{q}.x?v\}, \sigma, \mu \xrightarrow{m \rightarrow \mathbf{q}} \emptyset, \sigma[\mathbf{q}.x \mapsto v], \mu} \text{ [C|RecvVal]} \quad \frac{\tilde{\eta}_1, \sigma, \mu \xrightarrow{\phi} \tilde{\eta}_2, \sigma', \mu'}{(\tilde{\eta}_1 \uplus \tilde{\eta}'), \sigma, \mu \xrightarrow{\phi} (\tilde{\eta}_2 \uplus \tilde{\eta}'), \sigma', \mu'} \text{ [C|Mon]} \\
\frac{e \downarrow_p^\sigma v}{\{\mathbf{p}\langle e \rangle \rightarrow \mathbf{q}.x\}, \sigma, \mu \xrightarrow{\mathbf{p} \rightarrow \mathbf{q}} \emptyset, \sigma[\mathbf{q}.x \mapsto v], \mu} \text{ [C|SyncVal]} \quad \frac{}{\{\mathbf{p} \rightarrow \mathbf{q}[\ell]\}, \sigma, \mu \xrightarrow{\mathbf{p} \rightarrow \mathbf{q}} \emptyset, \sigma, \mu} \text{ [C|SyncSel]} \\
\frac{}{\{\mathbf{p} \rightarrow \mathbf{q}[\ell]\}, \sigma, \mu \xrightarrow{\mathbf{p} \rightarrow m} \{\mathbf{q}[\ell]\}, \sigma, \mu[m \mapsto \ell]} \text{ [C|SendSel]} \quad \frac{\mu(m) = \ell}{\{\mathbf{q}[\ell]\}, \sigma, \mu \xrightarrow{m \rightarrow \mathbf{q}} \emptyset, \sigma, \mu} \text{ [C|RecvSel]} \\
\frac{\tilde{\eta}_1, \sigma, \mu \xrightarrow{\phi_1} \tilde{\eta}'_1, \sigma', \mu' \quad \tilde{\eta}_2, \sigma', \mu' \xrightarrow{\phi_2} \tilde{\eta}''_2, \sigma'', \mu'' \quad (\dagger)}{(\tilde{\eta}_1 \uplus \tilde{\eta}_2), \sigma, \mu \xrightarrow{\phi_1 \cup \phi_2} (\tilde{\eta}'_1 \uplus \tilde{\eta}''_2), \sigma'', \mu''} \text{ [C|Join]}
\end{array}$$

Figure 4: Semantics of individual communications. The side condition (\dagger) in rule [C|Join] reads: if memory cell m occurs in both ϕ_1 and ϕ_2 , then it is not both written to in ϕ_1 and read from in ϕ_2 .

deal with label selections similarly. Rule [C|Mem] covers internal transitions in the automaton that only modify memory. Finally, rules [C|Mon] and [C|Join] extend this notion to transitions labelled by non-singleton sets. Rule [C|Mon] states that some communications in $\tilde{\eta}$ may not be executed at all (i.e., they are postponed until a later reduction). Rule [C|Join] allows executing several communications at the same time; note that labels ϕ_1 and ϕ_2 may share constraints (e.g., with multi-cast), while $\tilde{\eta}_1$ and $\tilde{\eta}_2$ must be disjoint (\uplus is the disjoint union operator).

The syntactic assumptions on choreography terms (page 7), namely distinct receivers (all terms on the right-hand sides of interactions are distinct) and consistent sends (if a process sends to several processes, then it always sends the same value or label), ensure that the sequentialisation in [C|Join] is of no consequence (building the final set ϕ in any order always yields the same final states σ' and μ' in [C|Com]), except if the same memory cell is both written to and read from. In that case, the read must precede the write; this is guaranteed by side condition (\dagger) , which ensures that concurrent accesses to a memory cell are done in a consistent way.

Example 9 (Book sale). We formally derive the reductions in Ex. 8, using rule [C|Com] and Fig. 4. Let \mathcal{G} , σ_0 , σ'_0 , and \mathcal{A}_0 be defined as in Ex. 8.

For the first reduction, first, $\mathcal{A}_0(\mathbf{a}2\mathbf{c}) = \langle 1, \{m \mapsto \perp\} \rangle$ (rule [C|Com] , first premise). Next, automaton $\mathcal{G}(\mathbf{a}2\mathbf{c})$ has one transition out of state 1, namely $1 \xrightarrow{\mathbf{a} \rightarrow m} \mathbf{a}2\mathbf{c} \ 2$ (rule [C|Com] , third premise). Finally, using Fig. 4, we need to derive $\mathbf{a}(\text{title}) \rightarrow \mathbf{c}, \sigma_0, \{m \mapsto \perp\} \xrightarrow{\mathbf{a} \rightarrow m} \dots$ (rule [C|Com] , second premise); this follows from rule [C|SendVal] . Thus, we derive:

$$\begin{array}{c}
\frac{\text{title} \downarrow_a^{\sigma_0} \text{"foo"}}{\mathbf{a}(\text{title}) \rightarrow \mathbf{c}, \sigma_0, \{m \mapsto \perp\} \xrightarrow{\mathbf{a} \rightarrow m} \mathbf{c}. \text{title? "foo"}, \sigma_0, \{m \mapsto \text{"foo"}\}} \text{ [C|SendVal]} \\
\mathcal{A}_0(\mathbf{a}2\mathbf{c}) = \langle 1, \{m \mapsto \perp\} \rangle \quad \frac{}{1 \xrightarrow{\mathbf{a} \rightarrow m} \mathbf{a}2\mathbf{c} \ 2} \\
\frac{\mathbf{a}(\text{title}) \rightarrow \mathbf{c} \text{ thru } \mathbf{a}2\mathbf{c}; C', \sigma_0, \{\mathbf{a}2\mathbf{c} \mapsto \langle 1, \{m \mapsto \perp\} \rangle\} \cup \mathcal{A}_0^{\text{rest}}}{\rightsquigarrow_{\mathcal{G}} \mathbf{c}. \text{title? "foo"} \text{ thru } \mathbf{a}2\mathbf{c}; C', \sigma_0, \{\mathbf{a}2\mathbf{c} \mapsto \langle 2, \{m \mapsto \text{"foo"} \rangle\} \cup \mathcal{A}_0^{\text{rest}}}}
\end{array}$$

Henceforth, let $\mathcal{A}'_0 = \{\mathbf{a}2\mathbf{c} \mapsto \langle 2, \{m \mapsto \text{"foo"} \rangle\} \cup \mathcal{A}_0^{\text{rest}}\}$.

For the second reduction, similarly, we derive:

$$\begin{array}{c}
\frac{\{m \mapsto \text{"foo"}\}(m) = \text{"foo"}}{\mathbf{c}. \text{title? "foo"}, \sigma_0, \{m \mapsto \text{"foo"}\} \xrightarrow{m \rightarrow \mathbf{c}} \emptyset, \sigma'_0, \{m \mapsto \text{"foo"}\}} \text{ [C|RecvVal]} \\
\mathcal{A}'_0(\mathbf{a}2\mathbf{c}) = \langle 2, \{m \mapsto \text{"foo"}\} \rangle \quad \frac{}{2 \xrightarrow{m \rightarrow \mathbf{c}} \mathbf{a}2\mathbf{c} \ 1} \\
\frac{\mathbf{c}. \text{title? "foo"} \text{ thru } \mathbf{a}2\mathbf{c}; C', \sigma_0, \{\mathbf{a}2\mathbf{c} \mapsto \langle 2, \{m \mapsto \text{"foo"} \rangle\} \cup \mathcal{A}_0^{\text{rest}}\}}{\rightsquigarrow_{\mathcal{G}} \emptyset \text{ thru } \mathbf{a}2\mathbf{c}; C', \sigma'_0, \{\mathbf{a}2\mathbf{c} \mapsto \langle 1, \{m \mapsto \text{"foo"} \rangle\} \cup \mathcal{A}_0^{\text{rest}}\}}
\end{array}$$

$$\begin{array}{c}
\frac{\text{pn}(\tilde{\eta}) \cap \text{pn}(\tilde{\eta}') = \emptyset \quad \gamma \neq \gamma'}{(\tilde{\eta} \text{ thru } \gamma; \tilde{\eta}' \text{ thru } \gamma') \equiv (\tilde{\eta}' \text{ thru } \gamma'; \tilde{\eta} \text{ thru } \gamma)} \text{ [C|Eta-Eta]} \\
\frac{\text{pn}(\tilde{\eta}_1) \cap \text{pn}(\tilde{\eta}_2) = \emptyset}{(\tilde{\eta}_1 \text{ thru } \gamma; \tilde{\eta}_2 \text{ thru } \gamma) \equiv (\tilde{\eta}_1 \cup \tilde{\eta}_2) \text{ thru } \gamma} \text{ [C|Eta-Split]} \\
\frac{\text{p} \notin \text{pn}(\tilde{\eta})}{(\text{if p.e then } (\tilde{\eta} \text{ thru } \gamma; C_1) \text{ else } (\tilde{\eta} \text{ thru } \gamma; C_2)) \equiv (\tilde{\eta} \text{ thru } \gamma; \text{if p.e then } C_1 \text{ else } C_2)} \text{ [C|Eta-Cond]} \\
\frac{\text{pn}(C_i) \cap \text{pn}(\tilde{\eta}) = \emptyset}{(\text{def } X = C_2 \text{ in } (\tilde{\eta} \text{ thru } \gamma; C_1)) \equiv (\tilde{\eta} \text{ thru } \gamma; \text{def } X = C_2 \text{ in } C_1)} \text{ [C|Eta-Rec]} \\
\frac{\text{p} \neq \text{q}}{\text{if p.e then } (\text{if q.e' then } C_1 \text{ else } C_2) \text{ else } (\text{if q.e' then } C_1' \text{ else } C_2') \equiv \text{if q.e' then } (\text{if p.e then } C_1 \text{ else } C_1') \text{ else } (\text{if p.e then } C_2 \text{ else } C_2')} \text{ [C|Cond-Cond]} \\
\frac{}{(\text{def } X = C_2 \text{ in } C_1[X]) \preceq (\text{def } X = C_2 \text{ in } C_1[C_2])} \text{ [C|Unfold]} \\
\frac{}{\emptyset \text{ thru } \gamma; C \preceq C} \text{ [C|EtaEnd]} \quad \frac{}{(\text{def } X = C \text{ in } \mathbf{0}) \preceq \mathbf{0}} \text{ [C|ProcEnd]}
\end{array}$$

Figure 5: Cho-Reo-graphy, structural precongurence.

□

Example 10 (Book sale). *We formally derive the reduction of line 4 in Ex. 4. Let \mathcal{G} denote the connector mapping in Ex. 7, let σ denote a choreography state function such that $\sigma(\mathbf{a}.\text{money}) = \10 and $\sigma(\mathbf{c}.\text{book}) = \text{foo.pdf}$, let $\sigma' = \sigma[\mathbf{b}.\text{money} \mapsto \$10]$, let $\sigma'' = \sigma'[\mathbf{s}.\text{book} \mapsto \text{foo.pdf}]$, and let $\mathcal{A} = \mathcal{A}^{\text{rest}} \cup \{\text{ac2bs} \mapsto \{1, \emptyset\}\}$ denote an automaton state function, for some $\mathcal{A}^{\text{rest}}$. We derive:*

$$\begin{array}{c}
\frac{\frac{\text{money} \downarrow_{\mathbf{a}}^{\sigma} \$10}{\{\mathbf{a}\langle \text{money} \rangle \rightarrow \mathbf{b}\}, \sigma, \emptyset} \quad \frac{\text{book} \downarrow_{\mathbf{c}}^{\sigma} \text{foo.pdf}}{\{\mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}\}, \sigma', \emptyset}}{\frac{\text{a} \rightarrow \mathbf{b} \rightarrow \emptyset, \sigma', \emptyset \quad \text{c} \rightarrow \mathbf{s} \rightarrow \emptyset, \sigma'', \emptyset}{\{\mathbf{a}\langle \text{money} \rangle \rightarrow \mathbf{b}, \mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}\}, \sigma, \emptyset}} \text{ [C|Join]} \\
\frac{\mathcal{A}(\text{ac2bs}) = \langle 1, \emptyset \rangle \quad \frac{\text{a} \rightarrow \mathbf{b} \wedge \mathbf{c} \rightarrow \mathbf{s} \rightarrow \emptyset, \sigma'', \emptyset}{\{\mathbf{a}\langle \text{money} \rangle \rightarrow \mathbf{b}, \mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}\} \text{ thru ac2bs; } \mathbf{0}, \sigma, \mathcal{A}} \quad 1 \xrightarrow{\text{a} \rightarrow \mathbf{b} \wedge \mathbf{c} \rightarrow \mathbf{s}}_{\text{ac2bs}} 1}{\text{a} \rightarrow \mathbf{b} \wedge \mathbf{c} \rightarrow \mathbf{s} \rightarrow \emptyset, \sigma'', \emptyset \text{ thru ac2bs; } \mathbf{0}, \sigma'', \mathcal{A}} \text{ [C|EtaEnd]} \\
\rightsquigarrow_{\mathcal{G}}
\end{array}$$

□

Rule [C|Com] is the only rule in the semantics that can cause a choreography to get stuck; we discuss this in more detail in the next section.

The remaining rules defining $\rightsquigarrow_{\mathcal{G}}$ are standard from other choreography calculi (e.g., [22]), and they are given Fig. 6): they define reductions for conditionals, allow for reductions under procedure definitions, and close the reduction relation $\rightsquigarrow_{\mathcal{G}}$ under the structural precongurence \preceq . Rule [C|Struct] uses a structural precongurence that allows for actions to be swapped if they do not interfere. The most interesting rules deal with communications: [C|Eta-Split] and [C|EtaEnd], displayed in Fig. 5 (where $\text{pn}(\tilde{\eta})$ denotes the process names that occur in $\tilde{\eta}$). Rule [C|Eta-Split] allows interactions through the same connector to be joined in one $\tilde{\eta}$ or split among several ones, which is necessary for correct interaction with rule [C|Eta-Eta]. (See [21] for a similar discussion: this rule is needed whenever one $\tilde{\eta}$ can specify several communications. Ex. 11 below illustrates this interplay in our language.) Rule [C|EtaEnd] removes completed interactions from the head of a choreography. The remaining rules defining \preceq concern recursion unfolding and swapping of conditionals; these rules are straightforward adaptations of those in [22].

$$\begin{array}{c}
\frac{i = 1 \text{ if } e \downarrow_p^\sigma \text{ true, } i = 2 \text{ otherwise}}{\text{if } p.e \text{ then } C_1 \text{ else } C_2, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C_i, \sigma, \mathcal{A}} \quad [C|\text{Cond}] \\
\frac{C_1 \preceq C_2 \quad C_2, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C'_2, \sigma', \mathcal{A}' \quad C'_2 \preceq C'_1}{C_1, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C'_1, \sigma', \mathcal{A}'} \quad [C|\text{Struct}] \\
\frac{C_1, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C'_1, \sigma', \mathcal{A}'}{\text{def } X = C_2 \text{ in } C_1, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} \text{def } X = C_2 \text{ in } C'_1, \sigma', \mathcal{A}'} \quad [C|\text{Ctx}]
\end{array}$$

Figure 6: Cho-Reo-graphy, semantics.

Example 11. Consider the following choreography, where for simplicity we abstract from the actual values being communicated.

$$C \equiv \{p \rightarrow q, r \rightarrow s\} \text{ thru } \gamma; \{p \rightarrow q, t \rightarrow v\} \text{ thru } \gamma'$$

and assume that both $\mathcal{G}(\gamma)$ and $\mathcal{G}(\gamma')$ allow the interactions between the processes they connect to occur independently and in any order. Then it is actually possible that the communication between t and v is the first one to take place. In order for our choreography language to allow this behaviour, we need to use both $[C|\text{Eta-Split}]$ and $[C|\text{Eta-Eta}]$ to exchange actions, as follows.

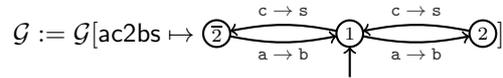
$$\begin{array}{l}
C \equiv \{p \rightarrow q, r \rightarrow s\} \text{ thru } \gamma; \{p \rightarrow q, t \rightarrow v\} \text{ thru } \gamma' \\
\equiv \{p \rightarrow q, r \rightarrow s\} \text{ thru } \gamma; t \rightarrow v \text{ thru } \gamma'; p \rightarrow q \text{ thru } \gamma' \quad \text{by } [C|\text{Eta-Split}] \\
\equiv t \rightarrow v \text{ thru } \gamma'; \{p \rightarrow q, r \rightarrow s\} \text{ thru } \gamma; p \rightarrow q \text{ thru } \gamma' \quad \text{by } [C|\text{Eta-Eta}]
\end{array}$$

□

4.2 Flexibility

An immediate advantage of CR is that different communication semantics can freely be mixed in the same choreography. A second advantage is that CR enables programmers to change the semantics of a choreography modularly, by altering the behaviour of the connectors through which the processes interact with each other, *without* the need to change the choreography itself.

Example 12 (Book sale). *The original book sale scenario (Ex. 1 and 2) requires Alice and Carol to send money and book to the bank and the shipper synchronously, as they initially do not trust each other. Now, suppose Alice and Carol establish mutual trust after successfully completing a number of book sales, such that their communications with the bank and the shipper no longer need to occur synchronously. Instead of redeveloping the choreography from scratch, we need to redefine only the connector mapping \mathcal{G} in Ex. 7, as follows:*



Thus, we updated the mapping for ac2bs ; for all other connectors, the mapping remains the same as in Ex. 7. The new automaton for ac2bs allows either a communication between Alice and the bank, asynchronously followed by a communication between Carol and the shipper (via state 2), or the same two communications in the reverse order (via state $\bar{2}$).⁴

Redefining the connector mapping for ac2bs is the only change we need to make: the choreography itself is exactly the same as in Ex. 4. This means that also the first reductions remain exactly the same as in Ex. 8 and 9. By contrast, the reduction in Ex. 10 is no longer valid, as it

⁴The communications between Alice and the bank, and between Carol and the shipper, are synchronous in this automaton. We can easily make those communications asynchronous as well, but we skip this here to save space (the automaton gets larger).

relies on the semantics of ac2bs . To show the difference formally, let σ , σ' , σ'' , and \mathcal{A} be defined as in Ex. 10. Let also $\mathcal{A}' = \mathcal{A}[\text{ac2bs} \mapsto \langle 2, \emptyset \rangle]$ and $\mathcal{A}'' = \mathcal{A}$. We derive:

$$\mathcal{A}(\text{ac2bs}) = \langle 1, \emptyset \rangle \xrightarrow{\sim_{\mathcal{G}}} \frac{\frac{\frac{\text{money} \downarrow_a^\sigma \$10}{\{\mathbf{a}\langle \text{money} \rangle \rightarrow \mathbf{b}\}, \sigma, \emptyset \xrightarrow{\mathbf{a} \rightarrow \mathbf{b}} \emptyset, \sigma', \emptyset} \llbracket C|\text{SyncVal} \rrbracket}{\{\mathbf{a}\langle \text{money} \rangle \rightarrow \mathbf{b}, \mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}\}, \sigma, \emptyset} \llbracket C|\text{Mon} \rrbracket}{\{\mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}\}, \sigma', \emptyset} \xrightarrow{\mathbf{a} \rightarrow \mathbf{b}} \frac{1 \xrightarrow{\mathbf{a} \rightarrow \mathbf{b}}_{\text{ac2bs}} 2}{\{\mathbf{a}\langle \text{money} \rangle \rightarrow \mathbf{b}, \mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}\} \text{ thru ac2bs}; \mathbf{0}, \sigma, \mathcal{A}} \llbracket C|\text{Mon} \rrbracket$$

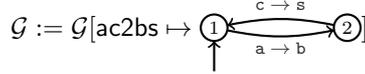
Next, we derive:

$$\mathcal{A}(\text{ac2bs}) = \langle 2, \emptyset \rangle \xrightarrow{\sim_{\mathcal{G}}} \frac{\frac{\frac{\text{book} \downarrow_c^\sigma \text{foo.pdf}}{\{\mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}\}, \sigma', \emptyset \xrightarrow{\mathbf{c} \rightarrow \mathbf{s}} \emptyset, \sigma'', \emptyset} \llbracket C|\text{SyncVal} \rrbracket}{\{\mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}\} \text{ thru ac2bs}; \mathbf{0}, \sigma', \mathcal{A}'} \llbracket C|\text{Mon} \rrbracket}{\emptyset \text{ thru ac2bs}; \mathbf{0}, \sigma'', \mathcal{A}''} \xrightarrow{\mathbf{c} \rightarrow \mathbf{s}}_{\text{ac2bs}} 1$$

Thus, as intended, our reduction rules let us derive two separate reductions with one communication each (first Alice and the bank, then Carol and the shipper) instead of one reduction with two communications (Ex. 10). Similarly, we can derive two separate reductions whereby Carol and the shipper communicate first, followed by Alice and the bank. \square

Example 13. The previous example works also “in the opposite direction”, from a trusting Alice and Carol (using connector mapping \mathcal{G} in Ex. 12) to cautious ones (using connector mapping \mathcal{G} in Ex. 7).

Our choreography is also compatible with the case where we have a trusting Alice and a cautious Carol, who only sends the book after receiving payment. A connector mapping that implements this behaviour is the following.



This connector mapping is still compatible with the choreography in Ex. 4. The symmetric case where Carol is trusting and Alice is cautious is similar. \square

Example 14. Note that the changes to connector mappings in Ex. 12 and 13 would still be possible if the programmer had written, e.g.,

$$\dots; \{\mathbf{a}\langle \text{money} \rangle \rightarrow \mathbf{b}\} \text{ thru ac2bs}; \{\mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}\} \text{ thru ac2bs}; \mathbf{0}$$

instead of the choreography in Ex. 4. Indeed, these two choreographies are equivalent due to the congruence rule $\llbracket C|\text{Eta-Split} \rrbracket$, and thus the sets of connectors that are compatible with each of them are the same.

This might be surprising at first, but it fits with the view of choreographies as global specifications of independent processes. Specifically in this case, no choreography can impose a causal dependency between the two communications $\mathbf{a}\langle \text{money} \rangle \rightarrow \mathbf{b}$ and $\mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}$ unless it includes an additional communication in the middle involving a process that can observe both. The lack of causal dependencies in this example thus leaves the connector for ac2bs free to decide the order in which the interactions are performed. \square

4.3 Deadlock-freedom

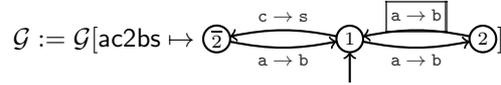
Rule $\llbracket C|\text{Com} \rrbracket$ is the only rule in the semantics that can cause a choreography to get stuck: in choreography $\tilde{\eta} \text{ thru } \gamma; C$, there can be incompatibilities between the communications allowed by connector γ and the intended communications in $\tilde{\eta}$, causing none of the communications in $\tilde{\eta}$ to be permitted by γ . In this case, we say that γ does not *respect* the choreography.⁵ Concretely, this

⁵A choreography expresses the intentions of the programmer. Although she may instantiate connectors however she likes, we assume they do not violate her intentions.

can happen because the transitions available at the current state s either require communications between processes not involved in $\tilde{\eta}$ or because there is an incompatibility with messages in transit (the premises of rules $\lfloor C|\text{RecvVal} \rfloor$ and $\lfloor C|\text{RecvSel} \rfloor$).

In more detail, the first premise in rule $\lfloor C|\text{Com} \rfloor$ always holds (assuming \mathcal{A} is defined for all connector names in $\tilde{\eta} \text{ thru } \gamma; C$). This gives us unique bindings for s and μ . The third premise in $\lfloor C|\text{Com} \rfloor$ is also always true (assuming every state of a connector has at least one outgoing transition; this can trivially be checked). For every outgoing transition of s , this gives us bindings for ϕ and s' . Now, the choreography gets stuck if for each of those bindings, the second premise in $\lfloor C|\text{Com} \rfloor$ is false. This can happen in two cases: either $\tilde{\eta}, \sigma, \mu \xrightarrow{\varphi} \tilde{\eta}', \sigma', \mu'$ can be derived (using the rules in Fig. 4) and $\phi \neq \varphi$ for every derivation, or $\tilde{\eta}, \sigma, \mu \xrightarrow{\varphi} \tilde{\eta}', \sigma', \mu'$ cannot be derived at all. The former happens if every φ contains different processes than ϕ (see Ex. 15), or the same processes but in different send/receive pairs (see Ex. 16); the latter happens if $\tilde{\eta}$ contains only asynchronous receives for which rules $\lfloor C|\text{RecvVal} \rfloor$ and $\lfloor C|\text{RecvSel} \rfloor$ in Fig. 4 are inapplicable (see Ex. 17).

Example 15 (Book sale). *Suppose we mistakenly redefine the connector mapping \mathcal{G} in Ex. 7 as follows (cf. Ex. 12; i.e., the boxed label is wrong):*



Thus, connector ac2bs initially allows a communication either between Alice and the bank, or between Carol and the shipper. In the latter case, ac2bs subsequently allows a communication between Alice and the bank, as in Ex. 12. But in the former case, ac2bs subsequently allows a second communication between Alice and the bank (instead of between Carol and the shipper).

The first derivation in Ex. 12 is still valid, but the second derivation is not: rule $\lfloor C|\text{SyncVal} \rfloor$ is still applied to derive $\{c\langle \text{book} \rangle \rightarrow s\}, \sigma', \emptyset \xrightarrow{c \rightarrow s} \emptyset, \sigma'', \emptyset$ to fulfill the second premise of rule $\lfloor C|\text{Com} \rfloor$, but ac2bs has no transition in state 2 labelled with $c \rightarrow s$. As there are no other derivations to fulfill the second premise of rule $\lfloor C|\text{Com} \rfloor$, the choreography gets stuck. \square

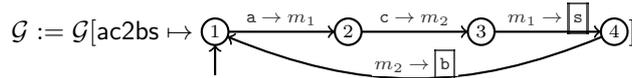
Example 16 (Book sale). *Suppose we mistakenly redefine connector mapping \mathcal{G} in Ex. 7 as follows (i.e., the boxed process names are wrong/swapped):*

$$\mathcal{G} := \mathcal{G}[\text{ac2bs} \mapsto \text{Barrier}[\mathbf{a}/p_1, \boxed{\mathbf{s}}/p_2, c/p_3, \boxed{\mathbf{b}}/p_4]]$$

Formally, automaton $\mathcal{G}(\text{ac2bs})$ has the following transition: $1 \xrightarrow{a \rightarrow s \wedge c \rightarrow b} \text{ac2bs} \ 1$. Thus, connector ac2bs allows communications between Alice and the shipper (instead of the bank), and between Carol and the bank (instead of the shipper).

The derivation in Ex. 10 is no longer valid: rules $\lfloor C|\text{SyncVal} \rfloor$ and $\lfloor C|\text{Join} \rfloor$ are still applied to derive $\{a\langle \text{money} \rangle \rightarrow b, c\langle \text{book} \rangle \rightarrow s\}, \sigma, \emptyset \xrightarrow{a \rightarrow b \wedge c \rightarrow s} \emptyset, \sigma'', \emptyset$ to fulfill the second premise of rule $\lfloor C|\text{Com} \rfloor$, but ac2bs has no transition labelled with $a \rightarrow b \wedge c \rightarrow s$. As there are no other derivations to fulfill the second premise of rule $\lfloor C|\text{Com} \rfloor$, the choreography gets stuck. \square

Example 17 (Book sale). *Suppose we mistakenly redefine the connector mapping \mathcal{G} in Ex. 7 as follows (i.e., the boxed process names are wrong/swapped):*



Thus, connector ac2bs allows an asynchronous send by Alice, followed by an asynchronous send by Carol, followed by an asynchronous receive by the shipper, followed by an asynchronous receive by the bank. However, the shipper receives the value sent by Alice (instead of Carol), while the bank receives the value sent by Carol (instead of Alice)

Let $\sigma, \sigma', \sigma''$, and \mathcal{A} be defined as in Ex. 10. Furthermore, let $\mu = \{m_1 \mapsto \perp, m_2 \mapsto \perp\}$, let $\mu' = \{m_1 \mapsto \$10, m_2 \mapsto \perp\}$, and let $\mu'' = \{m_1 \mapsto \$10, m_2 \mapsto \text{foo.pdf}\}$. The following reductions

can be derived using rule $\lfloor C \rfloor \text{Com} \rfloor$:

$$\begin{aligned} & \{ \mathbf{a} \langle \text{money} \rangle \rightarrow \mathbf{b}, \mathbf{c} \langle \text{book} \rangle \rightarrow \mathbf{s} \} \text{thru } \text{ac2bs}; \mathbf{0}, \sigma, \mathcal{A}[\text{ac2bs} \mapsto \langle 1, \mu \rangle] \\ \rightsquigarrow_{\mathcal{G}} & \{ \mathbf{b}.\text{money}?\$10, \mathbf{c} \langle \text{book} \rangle \rightarrow \mathbf{s} \} \text{thru } \text{ac2bs}; \mathbf{0}, \sigma, \mathcal{A}[\text{ac2bs} \mapsto \langle 2, \mu' \rangle] \\ \rightsquigarrow_{\mathcal{G}} & \{ \mathbf{b}.\text{money}?\$10, \mathbf{s}.\text{book}?\text{foo.pdf} \} \text{thru } \text{ac2bs}; \mathbf{0}, \sigma, \mathcal{A}[\text{ac2bs} \mapsto \langle 3, \mu'' \rangle] \end{aligned}$$

At this point, the choreography gets stuck: there are no derivations to fulfill the second premise of $\lfloor C \rfloor \text{Com} \rfloor$. To see this, note that only rule $\lfloor C \rfloor \text{RecvVal} \rfloor$ may be applicable (together with rule $\lfloor C \rfloor \text{Join} \rfloor$), but $\mu''(m_2) = \text{foo.pdf}$, whereas the choreography states $\mathbf{b}.\text{money}?\10 . In other words, the choreography expects the bank to receive $\$10$, but connector ac2bs allows the bank only to receive foo.pdf , out of memory cell m_2 . \square

None of these situations can arise in existing choreography models, where all (a)synchronous channels are guaranteed to respect their choreographies, because the choreography syntax is carefully tuned to the *fixed* communication semantics of these channels. In CR, we have no fixed communication semantics: the fact that connectors in CR may not respect their choreography is, thus, a consequence of the added expressiveness and flexibility CR provides.

We proceed with a more formal account.

Definition 2. Connector mapping \mathcal{G} in automaton state function \mathcal{A} respects choreography C if: for every $\sigma, \tilde{\eta}, \gamma, \sigma'$ and \mathcal{A}' , if $C, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}}^* \tilde{\eta} \text{thru } \gamma; C', \sigma', \mathcal{A}'$, then there exist σ'' and \mathcal{A}'' such that $\tilde{\eta} \text{thru } \gamma; C', \sigma', \mathcal{A}' \rightsquigarrow_{\mathcal{G}}^* \sigma'', \mathcal{A}''$. Connector mapping \mathcal{G} respects choreography C if \mathcal{G} respects C in initial automaton state function \mathcal{A}_0 (which assigns each automaton to its initial state and memory snapshot, as specified in \mathcal{G}).

Definition 3. C, σ, \mathcal{A}_0 is deadlock-free for every σ iff \mathcal{G} respects C .

We can show respectfulness/deadlock-freedom to be undecidable using a classical recursion-theoretic argument.

Theorem 1 (Undecidability of Deadlock-Freedom). *In general, it is undecidable whether a connector mapping \mathcal{G} respects a choreography C .*

Proof. Let η be a communication action that does not respect \mathcal{A} , and assume that connector γ has synchronous links $\mathbf{p}2\mathbf{q}$ and $\mathbf{q}2\mathbf{p}$, from \mathbf{p} to \mathbf{q} and conversely (e.g., *Sync* in Fig. 1). Synchronous links are always enabled and do not change \mathcal{A} .

Let f be a total function implemented at \mathbf{p} and consider the choreography

$$\begin{aligned} C \equiv \text{def } X &= \mathbf{q} \langle y \rangle \rightarrow \mathbf{p}.x \text{ thru } \gamma; \\ & \text{if } (\mathbf{p}.f(x) = 0) \text{ then } \mathbf{p} \langle x+1 \rangle \rightarrow \mathbf{q}.y \text{ thru } \gamma; X \\ & \text{else } \eta \text{ thru } \gamma \\ & \text{in } \mathbf{p} \langle 0 \rangle \rightarrow \mathbf{q}.y \text{ thru } \gamma; X \end{aligned}$$

In this choreography, \mathbf{q} sequentially sends the natural numbers to \mathbf{p} , which applies f to its input and proceeds if the result is 0. If \mathbf{q} sends a value where f is not 0, the choreography attempts to perform η and deadlocks. Then C respects \mathcal{A} iff f is constantly equal to 0, which by Rice's theorem is not decidable. \square

Remark 2. *Undecidability of deadlock-freedom arises because of the new ways in which a choreography and connectors can affect each other, which did not exist in previous work. Specifically, deadlock occurs if a connector's current state has no transitions for the interactions in the choreography's current $\tilde{\eta}$. In previous models, this can never happen, since the choreography syntax matches the hard-wired communication semantics by definition. Violation of respectfulness is, thus, a unique byproduct of allowing custom communication semantics, through connectors. Concretely, the proof of Thm. 1 relies on the existence of a communication action that does not respect \mathcal{G} , which does not exist in previous models.* \square

$$\begin{array}{c}
\mathcal{A}(\gamma) = \langle s, \mu \rangle \quad \left\{ \begin{array}{l} \Gamma \vdash_{\mathcal{A}[\gamma \rightarrow \langle s', \mu' \rangle]} \tilde{\eta}' \text{ thru } \gamma; C' \\ C \equiv \tilde{\eta} \text{ thru } \gamma; C' \\ \text{and } \tilde{\eta}, \mu \xrightarrow{\phi} \tilde{\eta}', \mu' \\ \text{and } s \xrightarrow{\phi} s' \end{array} \right\} \quad (\dagger) \\
\hline
\Gamma \vdash_{\mathcal{A}} C \quad [\text{CC|Com}] \\
\\
\frac{\Gamma \vdash_{\mathcal{A}} C}{\Gamma \vdash_{\mathcal{A}} \emptyset \text{ thru } \gamma; C} [\text{CC|Done}] \quad \frac{}{\Gamma \vdash_{\mathcal{A}} \mathbf{0}} [\text{CC|Nil}] \quad \frac{\Gamma \vdash_{\mathcal{A}} C_1 \quad \Gamma \vdash_{\mathcal{A}} C_2}{\Gamma \vdash_{\mathcal{A}} \text{if } p.e \text{ then } C_1 \text{ else } C_2} [\text{CC|Cond}] \\
\frac{\Gamma, (X : \mathcal{A}_X) \vdash_{\mathcal{A}} C_1 \quad \Gamma, (X : \mathcal{A}_X) \vdash_{\mathcal{A}} C_2}{\Gamma \vdash_{\mathcal{A}} \text{def } X = C_2 \text{ in } C_1} [\text{CC|Def}] \quad \frac{(X : \mathcal{A}) \in \Gamma}{\Gamma \vdash_{\mathcal{A}} X} [\text{CC|Call}]
\end{array}$$

Figure 7: Cho-Reo-graphy, compatibility relation. The side condition (\dagger) reads: the set of judgments on the left is nonempty. We abuse notation in rule $[\text{CC|Com}]$ to indicate that all judgments in this set must be true.

We can approximate respectfulness by a decidable relation, called *compatibility*, essentially by abstracting away from data. The key point is that a conditional satisfies compatibility only if both its branches satisfy compatibility.

Definition 4. Let C be a choreography, \mathcal{G} be a connector mapping, and \mathcal{A} be an automaton state function. We say that C and \mathcal{G} are compatible by automaton state function \mathcal{A} if $\vdash_{\mathcal{A}}^{\mathcal{G}} C$, where the relation \vdash is defined by the rules in Fig. 7. We say C and \mathcal{G} are compatible, written $\vdash^{\mathcal{G}} C$, if $\vdash_{\mathcal{A}_0}^{\mathcal{G}} C$ with \mathcal{A}_0 as in Definition 2.

Relation \vdash uses a context Γ , defined inductively as $\Gamma ::= (X : \mathcal{A}), \Gamma \mid \cdot$, and an abstraction of the labelled reductions for communications from Fig. 4, $\tilde{\eta}, \mu \xrightarrow{\phi} \tilde{\eta}', \mu'$. The latter models a symbolic execution of communications; it is defined as in Fig. 4, with two differences: (i) σ is removed from the domain of the reduction and (ii) in rule $[\text{C|SendVal}]$, v is a fresh token.

We comment on this relation. Intuitively, $X : \mathcal{A} \in \Gamma$ indicates that procedure X can be called only whenever the automata have current states \mathcal{A} ; this is encoded in rules $[\text{CC|Def}]$ and $[\text{CC|Call}]$ (in the former rule, a unique automaton state function \mathcal{A}_X is stipulated; in the latter rule, it is checked against the current automaton state function \mathcal{A}). Together with the fact that we allow actions to be swapped in rule $[\text{CC|Com}]$, but not recursive calls to be unfolded, this means that the recursive structures of the choreography and the automata in the connector mapping must be similar (i.e., the loops in the automata must match the recursions in the choreography). Furthermore, in order for these rules to ensure respectfulness, the transition relation in the automaton also needs to be confluent (cf. Thm. 2).

Remark 3. *Compatibility can become more robust/modular by disregarding connectors not occurring in procedure bodies in $[\text{CC|Def}]$. We chose the current formulation for simplicity.* \square

We now revisit our previous examples to demonstrate cases where the compatibility relation constitutes a precise approximation of respectfulness.

Example 18 (Book sale). We illustrate how compatibility works in the context of our running example by revisiting choreography $\{\mathbf{a}\langle \text{money} \rangle \rightarrow \mathbf{b}, \mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}\} \text{ thru } \text{ac2bs}; \mathbf{0}$ with five different connector mappings from previous examples.

✓ Let \mathcal{G} and \mathcal{A} be defined as in Ex. 10. Using Fig. 7, we derive:

$$\frac{\frac{\frac{}{\cdot \vdash_{\mathcal{A}[\text{ac2bs} \rightarrow \langle 1, \emptyset \rangle]}^{\mathcal{G}} \mathbf{0}}{\cdot \vdash_{\mathcal{A}[\text{ac2bs} \rightarrow \langle 1, \emptyset \rangle]}^{\mathcal{G}} \emptyset \text{ thru } \text{ac2bs}; \mathbf{0}} [\text{CC|Done}]}{\cdot \vdash_{\mathcal{A}}^{\mathcal{G}} \{\mathbf{a}\langle \text{money} \rangle \rightarrow \mathbf{b}, \mathbf{c}\langle \text{book} \rangle \rightarrow \mathbf{s}\} \text{ thru } \text{ac2bs}; \mathbf{0}} [\text{CC|Com}]}$$

$\mathcal{A}' = \mathcal{A}[\text{ac2bs} \mapsto \langle 2, \mu' \rangle]$, and let $\mathcal{A}'' = \mathcal{A}[\text{ac2bs} \mapsto \langle 2, \mu'' \rangle]$. Using Fig. 7, we attempt:

$$\frac{\frac{\frac{\mathcal{A}(\text{ac2bs})}{= \langle 1, \mu \rangle} \cdot \vdash_{\mathcal{A}}^{\mathcal{G}} \{ \mathbf{a} \langle \text{money} \rangle \rightarrow \mathbf{b}, \mathbf{c} \langle \text{book} \rangle \rightarrow \mathbf{s} \} \text{ thru ac2bs; } \mathbf{0}}{[CC|Com]} \quad \frac{\frac{\frac{\mathcal{A}'(\text{ac2bs})}{= \langle 2, \mu' \rangle} \cdot \vdash_{\mathcal{A}'}^{\mathcal{G}} \{ \mathbf{b} \langle \text{money} \rangle ? \diamond, \mathbf{c} \langle \text{book} \rangle \rightarrow \mathbf{s} \} \text{ thru ac2bs; } \mathbf{0}}{[CC|Com]} \quad \frac{\frac{\mathcal{A}''(\text{ac2bs})}{= \langle 3, \mu'' \rangle} \cdot \vdash_{\mathcal{A}''}^{\mathcal{G}} \{ \mathbf{b} \langle \text{money} \rangle ? \diamond, \mathbf{s} \langle \text{book} \rangle ? \heartsuit \} \text{ thru ac2bs; } \mathbf{0}}{[CC|Com]} \quad \boxed{\phantom{\text{derivation}}}}{[CC|Com]}$$

This attempted derivation fails, because the intended subderivation inside box (the receive of \diamond , followed by the receive of \heartsuit) is invalid: under our current \mathcal{G} , connector ac2bs in state 3 has no transition labelled with $m_1 \rightarrow \mathbf{b}$.

Thus, the choreography and the connector mapping are incompatible. In fact, in this case, the choreography deadlocks under \mathcal{G} . \square

However, the restriction that a choreography and the automata in its connector mapping must have similar recursive structures (for them to be judged compatible) implies there exist connector mappings that respect their choreographies, but that cannot be shown to do so by means of the compatibility relation – which is unavoidable in view of our undecidability result. We illustrate this by some examples.

Example 19. Let C be the simple choreography:

$$\text{def } X = \mathbf{p} \rightarrow \mathbf{q} \text{ thru } \gamma; \mathbf{p} \rightarrow \mathbf{q} \text{ thru } \gamma; \mathbf{r} \rightarrow \mathbf{s} \text{ thru } \gamma; X \text{ in } X$$

and $\mathcal{G}(\gamma)$ a connector that allows communications from \mathbf{p} to \mathbf{q} to occur simultaneously with communications from \mathbf{r} to \mathbf{s} (e.g., Barrier in Fig. 1). Then C is deadlock-free, since structural pre-congruence allows the second communication from \mathbf{p} to \mathbf{q} to be “delayed” and the communication from \mathbf{r} to \mathbf{s} to be “pushed forward”:

$$\begin{aligned} & \mathbf{p} \rightarrow \mathbf{q} \text{ thru } \gamma; \mathbf{p} \rightarrow \mathbf{q} \text{ thru } \gamma; \mathbf{r} \rightarrow \mathbf{s} \text{ thru } \gamma \\ \equiv & \mathbf{p} \rightarrow \mathbf{q} \text{ thru } \gamma; \{ \mathbf{p} \rightarrow \mathbf{q}, \mathbf{r} \rightarrow \mathbf{s} \} \text{ thru } \gamma && \text{by } [C|Eta\text{-Split}] \\ \equiv & \mathbf{p} \rightarrow \mathbf{q} \text{ thru } \gamma; \mathbf{r} \rightarrow \mathbf{s} \text{ thru } \gamma; \mathbf{p} \rightarrow \mathbf{q} \text{ thru } \gamma && \text{by } [C|Eta\text{-Split}] \end{aligned}$$

However, $\not\vdash^{\mathcal{G}} C$, since the second communication from \mathbf{p} to \mathbf{q} in the body of X cannot be consumed without unfolding the definition of X .

In this example, the recursive structure of X (two communications from \mathbf{p} to \mathbf{q} and one from \mathbf{r} to \mathbf{s}) differs from the recursive structure of $\mathcal{G}(\gamma)$ (one communication between each pair of processes). \square

Example 20. Consider now the choreography C defined as

$$\text{def } X = \mathbf{p} \rightarrow \mathbf{q} \text{ thru } \gamma; \mathbf{r} \rightarrow \mathbf{s} \text{ thru } \gamma; X \text{ in } X$$

where $\mathcal{G}(\gamma)$ only allows communications from \mathbf{p} to \mathbf{q} . Again C is deadlock-free, since structural pre-congruence allows the communications from \mathbf{r} to \mathbf{s} to be indefinitely postponed. However, $\not\vdash^{\mathcal{G}} C$. In general, compatibility ensures that the choreography is not only deadlock-free, but also that there is a correspondence between the recursive structure of the choreography and the recursive structure of the connectors: the connector must allow all interactions in the body of a definition to be executed before calling other procedures. \square

Theorem 2 (Compatibility Preservation). Let C and C' be choreographies, \mathcal{G} be a connector mapping, σ and σ' be choreography states, and \mathcal{A} and \mathcal{A}' be automaton state functions. If the transition relation in each $\mathcal{G}(\gamma)$ is confluent, $\vdash_{\mathcal{A}}^{\mathcal{G}} C$ and $C, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C', \sigma', \mathcal{A}'$, then $\vdash_{\mathcal{A}'}^{\mathcal{G}} C'$.

Proof. Straightforward by case analysis on the reduction from C, σ, \mathcal{A} to $C', \sigma', \mathcal{A}'$, using the fact that the automata are confluent (to make sure unfolding cannot add unwanted reductions), and therefore compatibility is preserved by structural precongruence. \square

The hypothesis that the transition relations of automata are confluent is required to make sure that unfolding cannot add unwanted reductions.

Corollary 1 (Soundness of Compatibility). *Under the assumptions of Theorem 2, if $\vdash_{\mathcal{A}}^{\mathcal{G}} C$, then \mathcal{G} in \mathcal{A} respects C .*

Proof. If $C, \sigma \mathcal{A} \rightsquigarrow_{\mathcal{G}}^* \eta; C', \sigma', \mathcal{A}'$, then, by induction on the length of this sequence of reductions, we use Theorem 2 to show that $\eta; C', \sigma', \mathcal{A}' \rightsquigarrow_{\mathcal{G}} C', \sigma'', \mathcal{A}''$ for some \mathcal{A}'' and σ'' . \square

Furthermore, compatibility is decidable.

Theorem 3 (Decidability of Compatibility). *There is an algorithm that, given C, \mathcal{G} and \mathcal{A} , returns YES if $\vdash_{\mathcal{A}}^{\mathcal{G}} C$ and NO if $\not\vdash_{\mathcal{A}}^{\mathcal{G}} C$ in time $O(P \times \max(p)_{\mathcal{A}} \times k \times (\sum d_{\mathcal{A}})^{2k})$, where $p_{\mathcal{A}}$ is the maximum number of ports in any automaton, $d_{\mathcal{A}}$ is the maximum number of transitions from a state in each automaton, k is the maximum number of communication actions in any procedure definition (or main choreography) and P is the total number of procedure definitions (including the main choreography).*

A simple finiteness argument suffices for establishing decidability of compatibility, since the number of automaton states is finite, the number of applicable rules at each step is finite, and all rules have a finite number of premises, and the size of the choreographies in the premises is always smaller than the size of the choreographies in the conclusions. Therefore, by non-deterministically guessing the types of all procedures, we can decide whether $\vdash_{\mathcal{A}}^{\mathcal{G}} C$ or not. However, we provide a more intelligent proof that constructs the types for recursive definitions.

Proof of Theorem 3. We assume that every procedure defined in C is called at least once outside of its body.

The idea behind our algorithm is to construct a derivation for $\vdash_{\mathcal{A}}^{\mathcal{G}} C$ by applying the rules in Fig. 7 bottom-up. When we meet a term of the form $\text{def } X = C_2 \text{ in } C_1$, we focus on C_1 first, and leave \mathcal{A}_X (see rule [CC|Def]) unspecified. We instantiate \mathcal{A}_X later, when we meet X for the first time inside of C_1 . More precisely:

1. Initialize a list $\mathcal{L} = [\cdot \vdash_{\mathcal{A}}^{\mathcal{G}} C]$.
2. While \mathcal{L} is not empty:
 - (a) Remove the first pending judgement $\Gamma \vdash_{\mathcal{A}}^{\mathcal{G}} C$ from \mathcal{L} .
 - (b) If C is $\mathbf{0}$, proceed to the next iteration.
 - (c) If C is of the form $\text{if } p.e \text{ then } C_1 \text{ else } C_2$, then add $\Gamma \vdash_{\mathcal{A}}^{\mathcal{G}} C_1$ and $\Gamma \vdash_{\mathcal{A}}^{\mathcal{G}} C_2$ at the beginning of \mathcal{L} .
 - (d) If C is of the form $\text{def } X = C_2 \text{ in } C_1$, then add $\Gamma, (X : \mathcal{A}_X) \vdash_{\mathcal{A}}^{\mathcal{G}} C_1$ and $\Gamma, (X : \mathcal{A}_X) \vdash_{\mathcal{A}}^{\mathcal{G}} C_2$, in this order, at the beginning of \mathcal{L} . Here, \mathcal{A}_X is a unique variable representing an unknown state function.
 - (e) If C is of the form X , there are two cases. If Γ contains $(X : \mathcal{A}_X)$ with \mathcal{A}_X instantiated, check whether $\mathcal{A}_X = \mathcal{A}$; if so, proceed to the next iteration, otherwise return NO. If Γ contains $(X : \mathcal{A}_X)$ with \mathcal{A}_X uninstantiated, replace all occurrences of \mathcal{A}_X in \mathcal{L} by \mathcal{A} and proceed to the next iteration.⁶
 - (f) Otherwise, C is of the form $\tilde{\eta} \text{ thru } \gamma; C'$. Consider all possible ways of rewriting C as $\tilde{\eta}' \text{ thru } \gamma; C'$ by swapping independent actions, without unfolding recursive definitions. Let $\mathcal{A}(\gamma) = \langle s, \mu \rangle$. For each such $\tilde{\eta}'$, check whether $\tilde{\eta}', \mu \xrightarrow{\phi} \tilde{\eta}'', \mu'$ for some ϕ , and in the affirmative case compute s' such that $s \xrightarrow{\phi}_{\gamma} s'$ and add $\Gamma \vdash_{\mathcal{A}[\gamma \rightarrow \langle s', \mu' \rangle]}^{\mathcal{G}} \tilde{\eta}'' \text{ thru } \gamma; C'$ at the beginning of \mathcal{L} . If no such transitions exist, return NO.

⁶Note that Γ must contain $(X : \mathcal{A}_X)$, otherwise the initial choreography is not well-formed.

3. Return YES.

Termination of this algorithm is straight-forward: the sum of the sizes of all the choreographies in \mathcal{L} strictly decreases at each iteration, and each step terminates in finite time. (The size of a choreography is the number of nodes in its abstract syntax tree, except that $p\langle e \rangle \rightarrow q.x$ and $p \rightarrow q[\ell]$ count as 2, while $q.x?v$ and $q[\ell]$ count as 1.) Soundness is immediate by observing that the judgements stored in \mathcal{L} are exactly those that are necessary to construct a derivation of $\vdash_{\mathcal{A}}^{\mathcal{G}} C$, since at each stage there is only one rule that can be applied to build such a derivation, and this rule is determined by the structure of C . If the algorithm returns YES, then a valid derivation for $\vdash_{\mathcal{A}}^{\mathcal{G}} C$ can be built. If the algorithm returns NO because of a mismatch between the state of the automata and a communication action (Step 2.f), then clearly $\not\vdash_{\mathcal{A}}^{\mathcal{G}} C$. If the algorithm returns NO because of an incompatibility between the state assigned to a procedure name X in Γ and the state in the current judgement (Step 2.e), then this failure means that we constructed two judgements involving X with different automaton state functions, which also implies that $\not\vdash_{\mathcal{A}}^{\mathcal{G}} C$.

To obtain the complexity bound, perform step 2f as follows: consider all possible transitions ($\sum d_A$) and check which ones are enabled (naively: go through the current choreography and check each transition, which yields $\max(p)_A \times k$). Each transition consumes at least half an interaction, so this can be repeated $\leq 2k$ times for each procedure definition. \square

Since automata are deterministic, $d_A < 2^{p_A}$ (each subset of ports determines at most one transition), and p_A is at most the number of processes in the choreography. Thus, this upper bound can be stated independently of the automata considered.

Although this complexity is high, we believe that compatibility checking is feasible in practice. The bound is an over-approximation, since choreographies typically contain many causal dependencies among communications that reduce non-determinism (as in our examples). Previous works on choreographies proposed algorithms with even worse worst-case complexity, but feasible in practice for the same reason [33].

Theorem 4 (Progress). *Let C be a choreography, \mathcal{G} be a connector mapping, σ be a choreography state and \mathcal{A} be an automaton state function such that $\vdash_{\mathcal{A}}^{\mathcal{G}} C$. Then, either $C \preceq \mathbf{0}$ (C has terminated) or there exist C' , σ' and \mathcal{A}' such that $C, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C', \sigma', \mathcal{A}'$.*

Proof. If $C \not\preceq \mathbf{0}$, then C is of the form $\tilde{\eta} \text{ thru } \gamma; C'$ or **if** $p.e$ **then** C_1 **else** C_2 , eventually inside some recursive definitions. In the latter case, C can always reduce; in the former case, compatibility guarantees that C can reduce. \square

Theorem 5 (Deadlock-Freedom by Design). *Let C be a choreography, σ be a choreography state function, and \mathcal{A} be an automaton state function. If $\vdash_{\mathcal{A}}^{\mathcal{G}} C$ and $C, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}}^* C', \sigma', \mathcal{A}'$, then either $C' \preceq \mathbf{0}$ or there exist C'' , σ'' and \mathcal{A}'' such that $C', \sigma', \mathcal{A}' \rightsquigarrow_{\mathcal{G}} C'', \sigma'', \mathcal{A}''$.*

Proof (sketch). From Theorem 4, if $C \not\preceq \mathbf{0}$, then by Theorem 2 we also have that $\vdash_{\mathcal{A}'}^{\mathcal{G}} C'$ whenever $C, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C', \sigma', \mathcal{A}'$. The thesis then follows by induction. \square

5 Connected Processes

CR shows how choreographies can be combined with connectors, but it does not indicate how we can obtain executable implementations. The missing link is determining how a choreography can be compiled to terms representing executable processes that communicate through connectors. We address this aspect by presenting a process calculus based on standard I/O actions and a translation (compilation procedure) from CR to this calculus.

5.1 Syntax and semantics

We define Connected Processes (CP), the process calculus to represent concrete implementations of choreographies. The syntax of CP is given in Fig. 8. A network N is a parallel composition of

$$\begin{aligned}
B ::= & \text{ol}\langle e \rangle; B \mid \text{i}?x; B \mid \text{o} \oplus \ell; B \mid \text{i}\&\{\ell_i : B_i\}_{i \in I} & N, M ::= & \text{p} \triangleright_\rho B \mid (N \mid M) \mid \mathbf{0} \\
& \mid \text{if } e \text{ then } B_1 \text{ else } B_2 \mid \text{def } X = B_2 \text{ in } B_1 \mid X \mid \mathbf{0}
\end{aligned}$$

Figure 8: Connected Processes, Syntax.

$$\begin{aligned}
& \frac{e \downarrow^\rho v}{\text{p} \triangleright_{\rho_p} \text{ol}\langle e \rangle; B_p \mid \text{q} \triangleright_{\rho_q} \text{i}?x; B_q, \mu \xrightarrow{\text{p.o} \rightarrow \text{q.i}} \text{p} \triangleright_{\rho_p} B_p \mid \text{q} \triangleright_{\rho_q[x \mapsto v]} B_q, \mu} \text{[CP|SyncVal]} \\
& \frac{e \downarrow^\rho v}{\text{p} \triangleright_\rho \text{ol}\langle e \rangle; B, \mu \xrightarrow{\text{p.o} \rightarrow m} \text{p} \triangleright_\rho B, \mu[m \mapsto v]} \text{[CP|SendVal]} \\
& \frac{\mu(m) = v}{\text{q} \triangleright_\rho \text{i}?x; B, \mu \xrightarrow{m \rightarrow \text{q.i}} \text{q} \triangleright_{\rho[x \mapsto v]} B, \mu} \text{[CP|RecvVal]} \\
& \frac{j \in I}{\text{p} \triangleright_{\rho_p} \text{o} \oplus \ell_j; B \mid \text{q} \triangleright_{\rho_q} \text{i}\&\{\ell_i : B_i\}_{i \in I}, \mu \xrightarrow{\text{p.o} \rightarrow \text{q.i}} \text{p} \triangleright_{\rho_p} B \mid \text{q} \triangleright_{\rho_q} B_j, \mu} \text{[CP|SyncSel]} \\
& \frac{}{\text{p} \triangleright_\rho \text{o} \oplus \ell; B, \mu \xrightarrow{\text{p.o} \rightarrow m} \text{p} \triangleright_\rho B, \mu} \text{[CP|SendSel]} \\
& \frac{\mu(m) = \ell_j \quad j \in I}{\text{q} \triangleright_\rho \text{i}\&\{\ell_i : B_i\}_{i \in I}, \mu \xrightarrow{m \rightarrow \text{q.i}} \text{q} \triangleright_\rho B_j, \mu} \text{[CP|RecvSel]} \\
& \frac{N_1, \mu \xrightarrow{\phi_1} N'_1, \mu' \quad N_2, \mu' \xrightarrow{\phi_2} N'_2, \mu''}{(N_1 \mid N_2), \mu \xrightarrow{\phi_1 \sqcup \phi_2} (N'_1 \mid N'_2), \sigma'', \mu''} \text{[CP|Join]}
\end{aligned}$$

Figure 9: Semantics of communications (process level). Side condition (\dagger) in [CP|Join] is the same as in [C|Join] .

processes. A process is written $\text{p} \triangleright_\rho B$, where p is its identifier, ρ its state (mapping variable names to values), and B its behaviour. Behaviours correspond to local views of choreography interactions. Procedure definitions and calls, conditionals, and termination ($\mathbf{0}$) follow the same ideas as in CR. Communication actions implement the local behaviour of each process in a choreography interaction: sending a value through an output port ($\text{ol}\langle e \rangle$); receiving a value through an input port ($\text{i}?x$); selecting a label through an output port ($\text{o} \oplus \ell$); and offering a choice on some labels through an input port ($\text{i}\&\{\ell_i : B_i\}_{i \in I}$).

The key difference from choreographies is that communications now refer to actual ports, instead of to connectors (we have no “ $\text{thru } \gamma$ ” for communications in the process calculus). This reflects the principle that processes should not know how they are connected [5, 29, 30].

The semantics of CP is parameterised on connectors represented as a set of automata \mathcal{C} that do not share any ports. Differently from the automata used in CR, the ones in \mathcal{C} use the names of the actual ports to which they are connected (and which are also used by the processes). Reductions in CP have the form $N, A \rightsquigarrow_{\mathcal{C}} N', A'$, where A maps each automaton in \mathcal{C} to a pair $\langle s, \mu \rangle$ of its state s and memory snapshot μ . The key reduction rule of CP is the one for communications.

$$\frac{a \in \mathcal{C} \quad A(a) = \langle s, \mu \rangle \quad N, \mu \xrightarrow{\phi} N', \mu' \quad s \xrightarrow{\phi}_a s'}{N, A \rightsquigarrow_{\mathcal{C}} N', A[a \mapsto \langle s', \mu' \rangle]} \text{[CP|Com]}$$

This rule is reminiscent of rule [C|Com] for choreographies. In particular, it uses a similar auxiliary reduction relation on pairs of networks and memory snapshots (stated in premise $N, \mu \xrightarrow{\phi} N', \mu'$), whose rules are given in Fig. 9.

The remaining rules defining the semantics of CP are standard, and given in Fig. 10. Rule

$$\begin{array}{c}
\frac{i = 1 \text{ if } e \downarrow^{\rho} \text{ true, } i = 2 \text{ otherwise}}{\mathbf{p} \triangleright_{\rho} \text{ if } e \text{ then } B_1 \text{ else } B_2, A \rightsquigarrow_{\mathcal{C}} \mathbf{p} \triangleright_{\rho} B_i, A} \quad [\text{CP|Cond}] \\
\frac{\mathbf{p} \triangleright_v B_1 \mid N, A \rightsquigarrow_{\mathcal{C}} \mathbf{p} \triangleright_v B'_1 \mid N', A'}{\mathbf{p} \triangleright_v \text{ def } X = B_2 \text{ in } B_1 \mid N, A \rightsquigarrow_{\mathcal{C}} \mathbf{p} \triangleright_v \text{ def } X = B_2 \text{ in } B'_1 \mid N', A'} \quad [\text{CP|Ctx}] \\
\frac{N, A \rightsquigarrow_{\mathcal{C}} N', A'}{N \mid M, A \rightsquigarrow_{\mathcal{C}} N' \mid M, A'} \quad [\text{CP|Par}] \quad \frac{N \preceq M \quad M, A \rightsquigarrow_{\mathcal{C}} M', A' \quad M' \preceq N'}{N, A \rightsquigarrow_{\mathcal{C}} N', A'} \quad [\text{CP|Struct}]
\end{array}$$

Figure 10: Connected Processes, Semantics.

$$\begin{aligned}
\llbracket \tilde{\eta} \text{ thru } \gamma; C \rrbracket_{\mathbf{r}} &= \begin{cases} \mathfrak{o}_{\gamma}!(e); [C]_{\mathbf{r}} & \text{if } \mathbf{r} = \mathbf{p} \text{ and } \mathbf{p}(e) \rightarrow \mathbf{q}.x \in \tilde{\eta} \\ \mathfrak{i}_{\gamma}?x; [C]_{\mathbf{r}} & \text{if } \mathbf{r} = \mathbf{q} \text{ and } (\mathbf{p}(e) \rightarrow \mathbf{q}.x \in \tilde{\eta} \text{ or } \mathbf{q}.x?v \in \tilde{\eta}) \\ \mathfrak{o}_{\gamma} \oplus \ell; [C]_{\mathbf{r}} & \text{if } \mathbf{r} = \mathbf{p} \text{ and } \mathbf{p} \rightarrow \mathbf{q}[\ell] \in \tilde{\eta} \\ \mathfrak{i}_{\gamma} \& \{\ell : [C]_{\mathbf{r}}\} & \text{if } \mathbf{r} = \mathbf{q} \text{ and } (\mathbf{p} \rightarrow \mathbf{q}[\ell] \in \tilde{\eta} \text{ or } \mathbf{q}[\ell] \in \tilde{\eta}) \end{cases} \\
\llbracket \text{if } \mathbf{p}.e \text{ then } C_1 \text{ else } C_2 \rrbracket_{\mathbf{r}} &= \begin{cases} \text{if } e \text{ then } [C_1]_{\mathbf{r}} \text{ else } [C_2]_{\mathbf{r}} & \text{if } \mathbf{r} = \mathbf{p} \\ [C_1]_{\mathbf{r}} \sqcup [C_2]_{\mathbf{r}} & \mathbf{r} \neq \mathbf{p} \end{cases} \\
\llbracket \text{def } X = C_2 \text{ in } C_1 \rrbracket_{\mathbf{r}} &= \text{def } X = [C_2]_{\mathbf{r}} \text{ in } [C_1]_{\mathbf{r}} \quad \llbracket \mathbf{0} \rrbracket_{\mathbf{r}} = \mathbf{0} \quad \llbracket X \rrbracket_{\mathbf{r}} = X
\end{aligned}$$

Figure 11: Cho-Reo-graphies, Behaviour Projection.

$[\text{CP|Struct}]$ uses the structural precongruence relation \preceq (including associativity and commutativity of \sqcup), defined in the standard way [22].

5.2 EndPoint Projection (EPP)

The EPP of a choreography C from CR into CP follows the usual construction, but with an additional ingredient: we need to add port names associated with communication actions. This is visible in the rules for projecting the individual behaviour of each process (Fig. 11), notably in the rule for projecting communications.

Remark 4. In Fig. 11, \mathfrak{o} and \mathfrak{i} denote variables that range over concrete ports. Thus, a process \mathbf{p} has output ports $\mathfrak{o}_{\gamma_1}, \mathfrak{o}_{\gamma_2}, \dots$, and input ports $\mathfrak{i}_{\gamma_1}, \mathfrak{i}_{\gamma_2}, \dots$, where \mathfrak{o} and \mathfrak{i} actually stand for $\mathfrak{o}^{\mathbf{p}}$ (“output port at \mathbf{p} ”) and $\mathfrak{i}^{\mathbf{p}}$ (“input port at \mathbf{p} ”), while connector γ_1 knows output ports $\mathfrak{o}_{\gamma_1}^{\mathbf{p}_1}, \mathfrak{o}_{\gamma_1}^{\mathbf{p}_2}, \dots$, and similarly for input ports. \square

The rule for projecting conditionals uses the standard partial merging operator \sqcup , where $B \sqcup B'$ is isomorphic to B and B' up to branching with different labels (see [22] for details).

We now define the projection of C given a state σ . As usual, this is the parallel composition of the projections of all processes in C .

$$\llbracket C, \sigma \rrbracket = \prod_{\mathbf{p} \in \text{pn}(C)} \mathbf{p} \triangleright_{\rho_{\mathbf{p}}} \llbracket C \rrbracket_{\mathbf{p}} \quad \text{where } \rho_{\mathbf{p}}(x) = \sigma(\mathbf{p}, x) \text{ for each variable } x \text{ at } \mathbf{p}$$

C is *projectable* when $\llbracket C, \sigma \rrbracket$ is defined for some σ . This is equivalent to saying that $\llbracket C, \sigma \rrbracket$ is defined for all σ . We illustrate endpoint projection in Ex. 21 and 22.

Example 21 (Book sale). *Continuing with our running example, the choreography presented in*

Ex. 4 is projectable, and yields the following network of connected processes each state σ .

$$\begin{aligned}
& \mathbf{a} \triangleright_{\rho_a} \mathbf{o}_{a2c}!(\langle title \rangle); \mathbf{i}_{c2a}?price; \mathbf{if}(\mathbf{happy}) \mathbf{then}(\mathbf{o}_{a2cbs} \oplus \mathbf{ok}; \mathbf{o}_{ac2bs}!(\langle money \rangle); \mathbf{0}) \\
& \qquad \qquad \qquad \mathbf{else}(\mathbf{o}_{a2cbs} \oplus \mathbf{ko}; \mathbf{0}) \\
& \mathbf{b} \triangleright_{\rho_b} \mathbf{i}_{a2cbs} \&\{ \mathbf{ok} : \mathbf{i}_{ac2bs}?money; \mathbf{0}; \mathbf{ko} : \mathbf{0} \} \\
& \mathbf{c} \triangleright_{\rho_c} \mathbf{i}_{a2c}?title; \mathbf{o}_{c2a}!(\langle price \rangle); \mathbf{i}_{a2cbs} \&\{ \mathbf{ok} : \mathbf{o}_{ac2bs}!(\langle book \rangle); \mathbf{0}; \mathbf{ko} : \mathbf{0} \} \\
& \mathbf{s} \triangleright_{\rho_s} \mathbf{i}_{a2cbs} \&\{ \mathbf{ok} : \mathbf{i}_{ac2bs}?book; \mathbf{0}; \mathbf{ko} : \mathbf{0} \}
\end{aligned}$$

□

Example 22. It is also worthwhile to note that the following choreographies are not congruent, and that they have different EPPs:

$$C_1 = \mathbf{p} \rightarrow \{ \mathbf{q}, \mathbf{r} \}[\ell] \mathbf{thru} \gamma; \mathbf{0} \qquad C_2 = \mathbf{p} \rightarrow \mathbf{q}[\ell] \mathbf{thru} \gamma; \mathbf{p} \rightarrow \mathbf{r}[\ell] \mathbf{thru} \gamma; \mathbf{0}$$

Choreography C_1 is syntactic sugar for $\{ \mathbf{p} \rightarrow \mathbf{q}[\ell], \mathbf{p} \rightarrow \mathbf{r}[\ell] \} \mathbf{thru} \gamma; \mathbf{0}$. The EPP to \mathbf{p} , thus, consists of one send; connector γ must subsequently ensure that label ℓ is replicated to, and received by, both \mathbf{q} and \mathbf{r} (i.e., formally, γ must be compatible with C_1).⁷

Choreography C_2 , in contrast, is not congruent to $\{ \mathbf{p} \rightarrow \mathbf{q}[\ell], \mathbf{p} \rightarrow \mathbf{r}[\ell] \} \mathbf{thru} \gamma; \mathbf{0}$. Specifically, we cannot use rule $[C]E\eta\text{-Split}$ to merge the two interactions in C_2 , because its disjointness premise does not hold (\mathbf{p} occurs in both interactions). Accordingly, the EPP of choreography C_2 on \mathbf{p} consists of two sends. □

To state the operational correspondence between a choreography and its projection, we need to map the process names used as ports in a connector mapping \mathcal{G} to the actual port names used in networks. We define $\llbracket \mathcal{G} \rrbracket$ to be the set of all automata in the codomain of \mathcal{G} , where each output port \mathbf{p} in automaton $\mathcal{G}(\gamma)$ becomes $\mathbf{p.o}_\gamma$ (and likewise for input ports). We define a similar function for automaton state function \mathcal{A} .

Theorem 6 (Operational Correspondence). *Let C be a projectable choreography. Then, for all σ , \mathcal{G} , and \mathcal{A} :*

Completeness: *If $C, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C', \sigma', \mathcal{A}'$, then $\llbracket C, \sigma \rrbracket, \llbracket \mathcal{A} \rrbracket \rightsquigarrow_{\llbracket \mathcal{G} \rrbracket} \llbracket C', \sigma' \rrbracket, \llbracket \mathcal{A}' \rrbracket$;*

Soundness: *If $\llbracket C, \sigma \rrbracket, \llbracket \mathcal{A} \rrbracket \rightsquigarrow_{\llbracket \mathcal{G} \rrbracket} N, \mathcal{A}'$, then $C, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C', \sigma', \mathcal{A}'$ for some σ' and \mathcal{A}' with $\llbracket C', \sigma' \rrbracket \prec N$ and $\llbracket \mathcal{A}' \rrbracket = \mathcal{A}'$.*

In the soundness result, the *pruning relation* \prec [15, 17] states that the processes in N' may offer more branches than those present in $\llbracket C', \sigma' \rrbracket$, but these are never selected [15, 32, 25].

In particular, if $\vdash^{\mathcal{G}} C$, then $\llbracket C, \sigma \rrbracket$ is guaranteed to be deadlock-free when executed with all automata in $\llbracket \mathcal{G} \rrbracket$ in their initial states.

Example 23 (Book sale). *For any connector mapping \mathcal{G} , the process network in Ex. 21 operates under $\llbracket \mathcal{G} \rrbracket$ exactly as the choreography in Ex. 4 under \mathcal{G} . In particular, if \mathcal{G} respects the original choreography, then this implementation never deadlocks under $\llbracket \mathcal{G} \rrbracket$.* □

6 Conclusions

Choreographic approaches to concurrent programming have been heavily investigated [1, 28], but they typically adopt some fixed (and restrictive) communication semantics (like point-to-point synchronous). CR is the first model that modularly integrates choreographies with what runs “under the hood” of communications, allowing for user-defined communication semantics given as connectors. Thanks to compatibility (Definition 4), CR inherits the good properties of both Choreographic Programming and Exogenous Coordination. Thus, we have significantly extended the

⁷This behaviour was the motivation for introducing multicasts as abbreviations: the notation in C_1 better conveys how communications really happen; EPP follows this intuition.

applicability of choreographies: not only can we capture new kinds of behaviours in choreographies (like barriers, cf. Ex. 7, and alternators, cf. Ex. 13), but we can even define systems that integrate different parts with different communication semantics *and* check whether such integration will lead to deadlocks. This is essential in many real-world scenarios, where different components with different communication semantics are usually combined (e.g., some microservices in a distributed system may asynchronously exchange data to be used later in a synchronous multiparty transition, similarly to our example).

This work lays the foundations for applying the combined power of choreographies and connectors to the challenge of concurrent programming, in that CR contains all the necessary foundations to obtain a concrete implementation. The results in § 5 specify how to use CR to obtain code in a process model supported by connectors. Thus, a natural next step will be to implement CR by combining implementations of processes generated from choreographies [19, 25] with distributed implementations of Reo connectors [31, 41, 42]. The main pieces exist; the main challenge lies in their effective composition, and CR is the first essential step towards this objective.

CR also provides a very explicit direction for future developments of this new combined research line: allowing for more kinds of connectors would make the model immediately more expressive. By relaxing the requirements we imposed on the automata in CR (see page 4), we can introduce non-deterministic communication semantics to choreographies, to cater to applications that require lossy channels and safe communication races. Likewise, a more fine-grained semantics that splits communications into two independent send and receive actions (similar to [24]) would enrich the class of behaviours that are captured.

We have followed the traditional approach of viewing choreographies as precise specifications of the intended interactions. However, it would be reasonable to allow choreographies to underspecify communications, such that the underlying connectors were allowed to exchange messages also to participants not defined in the choreography. For example, the semantics for the choreography term $p\langle e \rangle \rightarrow q.x \text{ thru } \gamma$ can allow γ to send the message from p to q via an intermediate process r that may perform additional actions (like logging the message, or sharing it through another connector). This generalisation can, in particular, provide a novel way for studying how choreographies can be applied to open-ended systems, where the processes projected from *multiple* choreographies execute in parallel and share connectors.

References

- [1] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniérou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. T. Vasconcelos, and N. Yoshida. Behavioral types in programming languages. *Found. Trends Progr. Lang.*, 3(2-3):95–230, 2016.
- [2] F. Arbab. The IWIM model for coordination of concurrent activities. In *COORDINATION*, pages 34–56. Springer, 1996.
- [3] F. Arbab. What do you mean, coordination. *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI*, 1122:1–18, 1998.
- [4] F. Arbab. Reo: a channel-based coordination model for component composition. *MSCS*, 14(3):329–366, 2004.
- [5] F. Arbab. Puff, the magic protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *LNCS*, pages 169–206. Springer, 2011.
- [6] F. Arbab, C. Baier, F. S. de Boer, J. J. M. M. Rutten, and M. Sirjani. Synthesis of reo circuits for implementation of component-connector automata specifications. In *COORDINATION*, volume 3454 of *LNCS*, pages 236–251. Springer, 2005.
- [7] C. Baier, J. Klein, and S. Klüppelholz. Synthesis of reo connectors for strategies and controllers. *Fundam. Inform.*, 130(1):1–20, 2014.

- [8] C. Baier, M. Sirjani, F. Arbab, and J. Rutten. Modeling component connectors in Reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, 2006.
- [9] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.
- [10] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12. IEEE Computer Society, 2006.
- [11] S. Bliudze and J. Sifakis. The algebra of connectors—structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008.
- [12] S. Bliudze and J. Sifakis. Causal semantics for the algebra of connectors. *Formal methods in system design*, 36(2):167–194, 2010.
- [13] R. Bruni, I. Lanese, and U. Montanari. A basic algebra of stateless connectors. *Theoretical Computer Science*, 366(1-2):98–120, 2006.
- [14] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2), 1994.
- [15] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
- [16] M. Carbone, S. Lindley, F. Montesi, C. Schürmann, and P. Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*, volume 59 of *LIPICs*, pages 33:1–33:15. Schloss Dagstuhl, 2016.
- [17] M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013.
- [18] M. Carbone, F. Montesi, C. Schürmann, and N. Yoshida. Multiparty session types as coherence proofs. *Acta Informatica*, pages 1–27, 2016.
- [19] Chor. Programming Language. <http://www.chor-lang.org/>.
- [20] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. Comp. Sci.*, 26(2):238–302, 2016.
- [21] L. Cruz-Filipe, K. S. Larsen, and F. Montesi. The paths to choreography extraction. In *FoSSaCS*, volume 10203 of *LNCS*, pages 424–440. Springer, 2017.
- [22] L. Cruz-Filipe and F. Montesi. A core model for choreographic programming. In *FACS 2016*, volume 10231 of *LNCS*, pages 17–35. Springer, 2017.
- [23] L. Cruz-Filipe and F. Montesi. Procedural choreographic programming. In *FORTE 2017*, volume 10321 of *LNCS*, pages 92–107. Springer, 2017.
- [24] L. Cruz-Filipe and F. Montesi. On asynchrony and choreographies. In *ICE 2017*, EPTCS. EPTCS, accepted for publication.
- [25] M. Dalla Preda, M. Gabbriellini, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies: Theory and implementation. *LMCS*, 13(2), 2017.
- [26] P.-M. Deniérou and N. Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP (II)*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013.
- [27] S. Giallorenzo. *Real-World Choreographies*. PhD thesis, University of Bologna, Italy, 2016.

- [28] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniérou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.
- [29] S.-S. Jongmans. *Automata-Theoretic Protocol Programming*. PhD thesis, Leiden University, 2016.
- [30] S.-S. Jongmans and F. Arbab. Prdk: Protocol programming with automata. In *TACAS*, pages 547–552. Springer, 2016.
- [31] S.-S. T. Jongmans, F. Santini, and F. Arbab. Partially distributed coordination with Reo and constraint automata. *Service Oriented Computing and Applications*, 9(3-4):311–339, 2015.
- [32] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *SEFM*, pages 323–332. IEEE, 2008.
- [33] J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 221–232, 2015.
- [34] K.-K. Lau, P. V. Elizondo, and Z. Wang. Exogenous connectors for software components. In *CBSE*, pages 90–106. Springer, 2005.
- [35] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, pages 517–530. ACM, 2016.
- [36] H. A. López, E. R. B. Marques, F. Martins, N. Ng, C. Santos, V. T. Vasconcelos, and N. Yoshida. Protocol-based verification of message-passing parallel programs. In *OOPSLA*, pages 280–298. ACM, 2015.
- [37] H. A. López, F. Nielson, and H. R. Nielson. Enforcing availability in failure-aware communicating systems. In *FORTE*, volume 9688 of *Lecture Notes in Computer Science*, pages 195–211. Springer, 2016.
- [38] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339. ACM, 2008.
- [39] F. Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013.
- [40] F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.
- [41] J. Proença. *Synchronous coordination of distributed components*. PhD thesis, Leiden University, 2011.
- [42] J. Proença, D. Clarke, E. De Vink, and F. Arbab. Dreams: a framework for distributed synchronous coordination. In *SAC*, pages 1510–1515. ACM, 2012.
- [43] C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.