Connectors meet Choreographies

Anonymous Author(s)

Abstract

Choreographic Programming and Exogenous Coordination are two powerful programming paradigms for developing correct concurrent software. Choreographic Programming enables a correct-by-construction synthesis procedure of concurrent process implementations that guarantees deadlock-freedom. Exogenous Coordination enables an elegant compositional approach to the development of inter-process connectors, where complex connectors can be built by assembling simpler ones (like synchronous barriers and asynchronous multi-casts).

We present Cho-Reo-graphies (CR), a new choreography calculus where these two research lines are combined for the first time, merging their benefits. In CR, choreographies are parametric in the connectors through which processes communicate. CR is the first choreography calculus where different communication semantics (determined by the connectors) can be freely mixed in the same program. Since connectors are user-defined, CR also supports many communication semantics that could not be achieved before for choreographies. We develop a decidable static analysis that guarantees that a choreography in CR and its user-defined connectors are compatible, and prove that this compatibility guarantees deadlock-freedom in process implementations synthesised from choreographies.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Background

Programming concurrent software is hard, because it is difficult to reason about how independent processes interact at runtime (e.g., by exchanging messages). Empirical studies reveal that two major reasons for this difficulty are: (i) while programmers have clear intentions about the order in which interactions should take place, existing tools do not adequately support the translation of such intentions to code [36]; (ii) it is easy to cause unintended side-effects when composing multiple interaction protocols together [34].

The challenge of concurrent programming has driven decades of research on new programming models. A particularly fruitful idea was that of providing 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* [37]. Central to Choreographic Programming is the *choreography*, a programming artefact that specifies a concurrent system in terms of the admissible 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), implementations in process languages can subsequently be synthesised from a choreography, which faithfully realise the interactions given in the choreography and are guaranteed to be deadlock-free by construction, even in the presence of arbitrary compositions [17, 38].

Previous work studied models for choreographies in settings with different interaction semantics (e.g., synchronous, asynchronous, multi-cast). Regardless of the specifics, 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. None of the existing models supports, for instance, 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, 33], where interaction protocols are developed separately from the code of the processes that will enact them. Process programs can then be modularly composed with protocols, given as *connectors*, which dictate which interactions take place 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.

Exogenous Coordination is strong where Choreographic Programming is weak: connectors offer an elegant model for defining different communication semantics, given in terms of arbitrary protocols. However, the opposite also holds: Choreographic Programming is strong where Exogenous Coordination is weak. When we compose processes with connectors, we do not have a global view of the system as in choreographies. Therefore, the programmer has no way to define the global flow of information in the system when multiple connectors are used separately by different parts of the system. Since we cannot know if the entire system is consistent, by considering both processes and their connectors, we may have deadlocks (which are prevented in Choreographic Programming).

Contribution

Choreographic Programming and Exogenous Coordination have complementary good properties. Can we then hope to combine them and obtain the best of the two worlds? In this paper, we answer this question in the affirmative, developing the first investigation of how connectors and choreographies can be integrated. We believe that our results represent the beginning of an interesting research line on concurrent programming. In the rest of this section, we summarise the achievements reported in this article; we state what we believe this work may lead to in the future in § 6.

We combine the best aspects of Choreographic Programming and Exogenous Coordination by presenting a new calculus of choreographies, called *Cho-Reo-graphies* (CR), in which the interactions among processes specified in a choreography are animated by arbitrary, user-defined, Reo connectors. CR allows for mixing different Reo 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 Reo 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 Reo 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 Reo connectors.

By allowing, for the first time, different communication semantics (i.e., Reo connectors) to be freely mixed in the same choreography, new technical challenges of formalisation and decidability arise. On the formalisation front, the main challenge is to balance expressiveness and comprehensibility (the formal semantics of the calculus should be easy enough to explain and understand). On the decidability front, the main challenges are proving that (1) deadlock-freedom is generally undecidable in our calculus, but (2) we can establish deadlock-freedom for a large subset of the language.

By leveraging existing work, distributed implementations of Reo connectors (e.g., in Scala [39, 40] or in Java [31]) can be automatically generated and deployed on different machines (e.g., for distributed objects and components); our approach is compatible with this existing work. As such, a practical tool based on the work in this paper can in principle be built on top of existing code generators for distributed implementations of Reo connectors.

Structure

In § 2, we motivate our work by means of an example. In § 3 we give the background on Reo connectors necessary for our development. In § 4 we develop our language integrating choreographies and Reo connectors, and prove the usual results on deadlock-freedom. In § 5 we introduce the target language for EndPoint Projection, show how to compile our choreographies, and prove the operational correspondence between a choreography and its projection. We conclude in § 6 with a discussion on the directions in which this work can be extended. Proofs are given in the Appendix.

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.

In Choreographic Programming, developers write high-level programs that describe the intended communications from a global viewpoint, assuming a fixed communication semantics. (e.g., asynchronous point to point). This enables a correct-by-construction synthesis procedure of concurrent process implementations that guarantees deadlock-freedom. Choreographies have been studied in settings with different communication semantics, including synchronous [15, 22, 32], asynchronous [17, 24, 27, 38], one-to-many [18, 20], and many-to-one [16, 35]. In our model, these communication semantics are simply instances of what we can do. But since we tap into the generality of Reo connectors, we can also do more (for example, we illustrate how to use barriers). Our development also extends the line of work on out-of-order execution for choreographies, originally from [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). Having out-of-order execution also simplifies our syntax: as in many other works, 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, our model captures asynchronous behaviour without requiring the programmer to reason about it in choreographies: communications are still specified as atomic interactions, which may be asynchronously reduced at runtime in a safe way (cf. [17, 26, 38]).

In Exogenous Coordination, developers program interaction protocols for communicating processes, called connectors, separately from the internal code of each process. This enables an elegant compositional approach to the development of protocols, where complex protocols can be built by assembling simpler ones (like synchronous barriers and asynchronous multi-casts). Exogenous Coordination has been studied extensively over the last two decades [2, 3, 33]. 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, 41], and Reo [4, 5].

2 Motivating Example & Approach

We now 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 to illustrate the different concepts we introduce at each stage.

▶ **Example 1** (Book sale). Consider a scenario where buyer 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. $\texttt{a}\langle \mathit{title} \rangle$ `->` `c`; $\texttt{c}\langle \mathit{price} \rangle$ `->` `a`;
2. `if` $\texttt{a}.\mathit{happy}$ `then` (`a` `->` $\texttt{c}[\mathit{ok}]$; `a` `->` $\texttt{b}[\mathit{ok}]$; `a` `->` $\texttt{s}[\mathit{ok}]$;
   $\texttt{a}\langle \mathit{money} \rangle$ `->` `b`; $\texttt{c}\langle \mathit{book} \rangle$ `->` `s`)
   `else` (`a` `->` $\texttt{c}[\mathit{ko}]$; `a` `->` $\texttt{b}[\mathit{ko}]$; `a` `->` $\texttt{s}[\mathit{ko}]$)

◀

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 her money before Carol has sent her book, and vice versa. To resolve this impasse, Alice and Carol should synchronously (barrier-like) send money and book, so they are sure each of them holds her end of the bargain. ◀

As we show in the next sections, our choreography model equipped with connectors is powerful enough to express all these interactions (and more). The idea is to tag every interaction between a sender `p` and a receiver `q` with the name of the particular connector through which the interaction transpires. For instance, a value communication from `p` to `q` through a synchronous channel sync is expressed as `p -> q thru sync`, while the same communication through an asynchronous channel async is expressed as `p -> q 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 Constraint Automata and Reo

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.

We model connectors using a general existing model of Exogenous Coordination, namely *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. 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 $p_1 \to p_2$ (port p_1 passes a message to port p_2), $m_1 \to m_2$ (cell m_1 passes a message to cell m_2), $p \to m$ (port p passes a message to cell m), and $m \to p$ (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., $p_1 \to m \wedge p_2 \to m$ is forbidden, but $p \to m_1 \wedge p \to m_2$ is allowed).

Formally, a constraint automaton is a tuple $(S, P, M, \longrightarrow, s_0, \mu_0)$, where S is a finite set of states, P is a finite set of ports, M is a finite set of memory cells, $\longrightarrow \subseteq S \times \Phi_{P,M} \times S$ is a transition relation, $s_0 \in S$ is an initial state, and μ_0 is an initial memory snapshot. The initial memory snapshot is a function that maps the memory cells in M to their initial content. We denote $(s, \phi, s') \in \longrightarrow$ as $s \xrightarrow{\phi} s'$, for short. A transition $s \xrightarrow{\phi} s'$ means that, from state s, a subset of the ports in P can interact according to ϕ. 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.

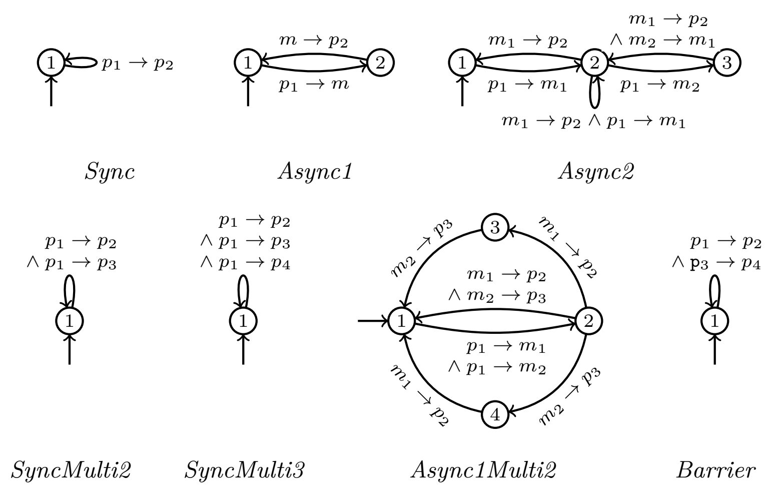
In this work, we restrict ourselves to a subset of constraint automata that satisfy two additional assumptions on their transitions. First, the occurrences of each port in transition labels are either all as sender (it occurs only on the left-hand side of constraints) or all as receiver (it occurs only 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 $p_1 \to p_2$ and the other by $p_2 \to p_3$. 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 § 4; the second assumption ensures that this semantics is deterministic, in line with previous work on choreographies.

▶ **Example 3.** Figure 1 shows example constraint automata for useful connectors.

Sync models a synchronous channel. Indefinitely, this connector 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. *Async1Multi2* models an asynchronous multi-cast connector for two receivers. *Barrier* models a barrier send/receive connector. ◀

Choreography programmers can model connectors by explicitly defining constraint automata. Alternatively, programmers can model connectors by *composing* constraint automata

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



$p_1 \to p_2$
Sync
$m \to p_2$
$p_1 \to m$
Async1
$m_1 \to p_2$
$m_1 \to p_2 \wedge m_2 \to m_1$
$p_1 \to m_1$
$p_1 \to m_2$
$m_1 \to p_2 \wedge p_1 \to m_1$
Async2
$p_1 \to p_2 \wedge p_1 \to p_3$
SyncMulti2
$p_1 \to p_2 \wedge p_1 \to p_3 \wedge p_1 \to p_4$
SyncMulti3
$m_2 \to p_3$
$m_1 \to p_2$
$m_1 \to p_2 \wedge m_2 \to p_3$
$p_1 \to m_1 \wedge p_1 \to m_2$
$m_1 \to p_2$
$m_2 \to p_3$
Async1Multi2
$p_1 \to p_2 \wedge p_3 \to p_4$
Barrier

Figure 1 Example constraint automata.

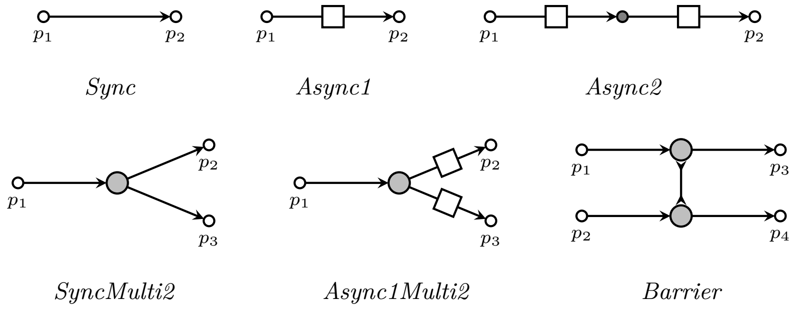


Figure 2 Example Reo connectors for the constraint automata in Figure 1.

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://reo.project.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.

4 Cho-Reo-graphies

4.1 Overview

We now present our choreography calculus that combines choreographies with Reo connectors, called *Cho-Reo-graphies* (CR). A choreography describes the behaviour of a set of processes.

$$C ::= \tilde{\eta}\,\texttt{thru}\,\gamma; C \mid \texttt{if}\,\texttt{p}.e\,\texttt{then}\,C_1\,\texttt{else}\,C_2 \mid \texttt{def}\,X = C_2\,\texttt{in}\,C_1 \mid X \mid \mathbf{0}$$

$$\eta ::= \texttt{p}\langle e\rangle\ \texttt{->}\ \texttt{q}.x \mid \texttt{p}\ \texttt{->}\ \texttt{q}[\ell] \mid \boxed{\texttt{q}.x?v} \mid \boxed{\texttt{q}[\ell]}$$

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.

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 Figure 3. We use $C, C', C_1, \ldots$ to range over choreographies; $\texttt{p}, \texttt{q}, \ldots$ to range over processes; $\eta, \eta', \eta_1, \ldots$ to range over interactions (and $\tilde{\eta}, \tilde{\eta}', \tilde{\eta}_1, \ldots$ to range over sets of them); $\gamma, \gamma', \gamma_1, \ldots$ to range over connector names; X to range over procedure names; $e, e', \ldots$ to range over (side-effect free) expressions; $v, v', \ldots$ to range over values; and $\ell, \ell', \ldots$ to range over selection labels. Each process $\texttt{p}$ has associated with it a finite set of local variables $\texttt{var}_\texttt{p} = \{x_1, \ldots, x_n\}$, and expressions are assumed to be an inductively defined set including $\texttt{var}_\texttt{p}$ 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 Example 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 $\texttt{p}\langle e\rangle\ \texttt{->}\ \{\texttt{q}_1.x_1, \ldots, \texttt{q}_n.x_n\}$ (value multicast) and $\texttt{p}\ \texttt{->}\ \{\texttt{q}_1, \ldots, \texttt{q}_n\}[\ell]$ (label multicast) to abbreviate $\{\texttt{p}\langle e\rangle\ \texttt{->}\ \texttt{q}_1.x_1, \ldots, \texttt{p}\langle e\rangle\ \texttt{->}\ \texttt{q}_n.x_n\}$ and $\{\texttt{p}\ \texttt{->}\ \texttt{q}_1[\ell], \ldots, \texttt{p}\ \texttt{->}\ \texttt{q}_n[\ell]\}$, respectively. Also, we write $\texttt{p}\langle x\rangle\ \texttt{->}\ \texttt{q}$ for $\texttt{p}\langle x\rangle\ \texttt{->}\ \texttt{q}.x$ (i.e., if sender $\texttt{p}$ sends the value in local variable x, and if receiver $\texttt{q}$ 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 (cf. Example 1) can be written as follows in CR.

```
1.  a⟨title⟩ -> c thru a2c;
2.  c⟨price⟩ -> a thru c2a;
3.  if a.happy then ( a -> {c, b, s}[ok] thru a2cbs;
4.                     {a⟨money⟩ -> b, c⟨book⟩ -> s} thru ac2bs; 0)
5.              else ( a -> {c, b, s}[ko] thru a2cbs; 0)
```

◀

The semantics of most terms is standard. In $\texttt{if}\,\texttt{p}.e\,\texttt{then}\,C_1\,\texttt{else}\,C_2$, process $\texttt{p}$ evaluates expression e; if this results in *true*, the choreography proceeds as C_1, and otherwise, as C_2. In $\texttt{def}\,X = C_2\,\texttt{in}\,C_1$, procedure X is defined as C_2; it can then be invoked as X from both C_1 and C_2. $\mathbf{0}$ indicates successful termination.

The interesting new bit is the semantics of a term $\tilde{\eta}\,\texttt{thru}\,\gamma$. Informally, $\tilde{\eta}\,\texttt{thru}\,\gamma$ specifies that all communications in $\tilde{\eta}$ occur through connector γ. More precisely, in $\texttt{p}\langle e\rangle\ \texttt{->}\ \texttt{q}.x\,\texttt{thru}\,\gamma$, process $\texttt{p}$ (the *sender*) evaluates expression e and offers the resulting value to connector γ; the connector eventually accepts and offers it to process $\texttt{q}$ (the *receiver*), who stores it in its local variable x. The behaviour of $\texttt{p}\ \texttt{->}\ \texttt{q}[\ell]\,\texttt{thru}\,\gamma$ is similar, except that $\texttt{p}$ offers a selected 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 Figure 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 (i.e., they do not necessarily synchronise sends with receives). In particular, $\mathtt{q}.x?v$ is obtained when a communication $\mathtt{p}\langle e\rangle$ `->` $\mathtt{q}.x$ is partially executed, and $\mathtt{p}$ has already sent its value, but $\mathtt{q}$ has not yet received it; $\mathtt{q}[\ell]$ arises similarly – see Example 7 below.

For the semantics of $\tilde{\eta}\,\mathtt{thru}\,\gamma$ to be well-defined, $\tilde{\eta}$ must satisfy two conditions. First, all interactions in $\tilde{\eta}$ must have *distinct receivers*: if $\mathtt{p}_1\langle e_1\rangle$ `->` $\mathtt{q}_1.x_1 \in \tilde{\eta}$ and $\mathtt{p}_2\langle e_2\rangle$ `->` $\mathtt{q}_2.x_2 \in \tilde{\eta}$, then $\mathtt{q}_1 \neq \mathtt{q}_2$. This ensures that the value received by a receiver is uniquely defined. Second, all sends must be *consistent*: if a process $\mathtt{p}$ 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.

- ✓ $\{\mathtt{a}\langle \mathit{money}\rangle \texttt{->} \mathtt{b}, \mathtt{c}\langle \mathit{book}\rangle \texttt{->} \mathtt{s}\}$ – Alice sends money to the bank, while Carol sends a book to the shipper (distinct receivers; consistent sends).
- ✗ $\{\mathtt{a}\langle \mathit{money}\rangle \texttt{->} \mathtt{b}, \mathtt{c}\langle \mathit{money}\rangle \texttt{->} \mathtt{b}\}$ – 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).
- ✗ $\{\mathtt{c}\langle \mathit{price}\rangle \texttt{->} \mathtt{a}, \mathtt{c}\langle \mathit{book}\rangle \texttt{->} \mathtt{s}\}$ – 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).
- ✓ $\{\mathtt{a} \texttt{->} \mathtt{b}[\mathit{ok}], \mathtt{a} \texttt{->} \mathtt{c}[\mathit{ok}], \mathtt{a} \texttt{->} \mathtt{s}[\mathit{ok}]\}$ – Alice sends label *ok* to Carol, the bank, and the shipper (distinct receivers; consistent sends). ◀

4.2 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 6** (Book sale). The connector names that occur in the choreography in Example 4 are a2c, c2a, a2cbs, and ac2bs. The requirements in Example 2 subsequently give rise to the following connector mapping:

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

where $\mathit{Async1}[\mathtt{a}/p_1, \mathtt{c}/p_2]$ denotes automaton *Async1* in Figure 1, with $\mathtt{a}$ substituted for p_1, and $\mathtt{c}$ for p_2 (and likewise in the other mappings). Under this connector mapping, thus, Alice and Carol communicate via asynchronous channels (a2c and c2a) to exchange title and price;

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

Alice, Carol, the bank, and the shipper communicate via synchronous multi-cast ($\mathsf{a2cbs}$) to exchange *ok* or *ko*, and via barrier sends/receives ($\mathsf{ac2bs}$) to exchange money and book. ◀

▶ Remark. 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 § 4.4. ◀

The reduction relation for CR under a given $\mathcal{G}$ is denoted as $\rightsquigarrow_{\mathcal{G}}$; it ranges over triples $C, \sigma, \mathcal{A}$, where C is a choreography, σ is a choreography state function (mapping each process to a mapping of its variables to values, i.e., $\sigma(\mathtt{p}.x)$ is the value stored at variable x in process $\mathtt{p}$), 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 7** (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 Example 6, let C' denote lines 2–3 in Example 4, let σ_0 denote the initial choreography state function such that $\sigma_0(\mathtt{a}.\mathit{title}) = \texttt{"foo"}$, let $\sigma_0' = \sigma_0[\mathtt{c}.\mathit{title} \mapsto \texttt{"foo"}]$, and let $\mathcal{A}_0 = \{\mathsf{a2c} \mapsto \langle 1, \{m \mapsto \bot\}\rangle\} \cup \mathcal{A}_0^{\text{rest}}$ denote the initial automaton state function, where:

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

Initially, thus, all connectors are in their initial state (state 1). Furthermore, connectors $\mathsf{a2c}$ and $\mathsf{c2a}$ have an empty memory cell ($m \mapsto \bot$); connectors $\mathsf{a2cbs}$ and $\mathsf{ac2bs}$ have no memory cells (memory snapshot $\emptyset$). By rule $\lfloor \text{C}|\text{Com} \rceil$ (presented after this example), the choreography in Example 4 reduces under $\mathcal{G}$ as follows:

$$\mathtt{a}\langle \mathit{title} \rangle \; \texttt{->} \; \mathtt{c} \; \mathtt{thru} \; \mathsf{a2c}; C', \quad \sigma_0, \quad \{\mathsf{a2c} \mapsto \langle 1, \{m \mapsto \bot\}\rangle\} \cup \mathcal{A}_0^{\text{rest}}$$

$\rightsquigarrow_{\mathcal{G}}$ In state 1, according to $\mathcal{G}$, connector $\mathsf{a2c}$ only has a transition that allows Alice to send (asynchronously) to Carol. By performing such a send, Alice enables the choreography to make a reduction, in which the first half of the communication completes; the (asynchronous) receive remains. In the same step, $\mathsf{a2c}$ moves to state 2, and the value sent by Alice is stored in $\mathsf{a2c}$'s internal memory cell (*title* evaluates to $\texttt{"foo"}$, based on σ_0).

$$\mathtt{c}.\mathit{title}?\texttt{"foo"} \; \mathtt{thru} \; \mathsf{a2c}; C', \quad \sigma_0, \quad \{\mathsf{a2c} \mapsto \langle 2, \{m \mapsto \texttt{"foo"}\}\rangle\} \cup \mathcal{A}_0^{\text{rest}}$$

$\rightsquigarrow_{\mathcal{G}}$ In state 2, according to $\mathcal{G}$, connector $\mathsf{a2c}$ only has a transition that allows Carol to receive (asynchronously) from Alice. By performing such a receive, Carol enables the choreography to make a reduction in which the whole communication completes. In the same step, $\mathsf{a2c}$ moves to state 1 (the internal memory cell is not cleared).

$$\emptyset \; \mathtt{thru} \; \mathsf{a2c}; C', \quad \sigma_0', \quad \{\mathsf{a2c} \mapsto \langle 1, \{m \mapsto \texttt{"foo"}\}\rangle\} \cup \mathcal{A}_0^{\text{rest}}$$

◀

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

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

$$\frac{e \downarrow^{\sigma}_{\mathtt{p}} v}{\{\mathtt{p}\langle e\rangle \text{ -> } \mathtt{q}.x\}, \sigma, \mu \xrightarrow{\mathtt{p}\to\mathtt{q}} \emptyset, \sigma[\mathtt{q}.x \mapsto v], \mu} \;\lfloor\mathrm{C|SyncVal}\rceil$$

$$\frac{e \downarrow^{\sigma}_{\mathtt{p}} v}{\{\mathtt{p}\langle e\rangle \text{ -> } \mathtt{q}.x\}, \sigma, \mu \xrightarrow{\mathtt{p}\to m} \mathtt{q}.x?v, \sigma, \mu[m \mapsto v]} \;\lfloor\mathrm{C|SendVal}\rceil$$

$$\frac{\mu(m) = v}{\{\mathtt{q}.x?v\}, \sigma, \mu \xrightarrow{m\to\mathtt{q}} \emptyset, \sigma[\mathtt{q}.x \mapsto v], \mu} \;\lfloor\mathrm{C|RecvVal}\rceil \qquad \frac{\mu(m) = \ell}{\{\mathtt{q}[\ell]\}, \sigma, \mu \xrightarrow{m\to\mathtt{q}} \emptyset, \sigma, \mu} \;\lfloor\mathrm{C|RecvSel}\rceil$$

$$\frac{}{\{\mathtt{p} \text{ -> } \mathtt{q}[\ell]\}, \sigma, \mu \xrightarrow{\mathtt{p}\to\mathtt{q}} \emptyset, \sigma, \mu} \;\lfloor\mathrm{C|SyncSel}\rceil \qquad \frac{}{\{\mathtt{p} \text{ -> } \mathtt{q}[\ell]\}, \sigma, \mu \xrightarrow{\mathtt{p}\to m} \{\mathtt{q}[\ell]\}, \sigma, \mu[m \mapsto \ell]} \;\lfloor\mathrm{C|SendSel}\rceil$$

$$\frac{\mu(m_1) = v}{\emptyset, \sigma, \mu \xrightarrow{m_1\to m_2} \emptyset, \sigma, \mu[m_2 \mapsto v]} \;\lfloor\mathrm{C|Mem}\rceil \qquad \frac{\tilde{\eta}_1, \sigma, \mu \xrightarrow{\phi} \tilde{\eta}_2, \sigma', \mu'}{(\tilde{\eta}_1 \uplus \tilde{\eta}'), \sigma, \mu \xrightarrow{\phi} (\tilde{\eta}_2 \uplus \tilde{\eta}'), \sigma', \mu'} \;\lfloor\mathrm{C|Mon}\rceil$$

$$\frac{\tilde{\eta}_1, \sigma, \mu \xrightarrow{\phi_1} \tilde{\eta}'_1, \sigma', \mu' \qquad \tilde{\eta}_2, \sigma', \mu' \xrightarrow{\phi_2} \tilde{\eta}'_2, \sigma'', \mu'' \qquad (\dagger)}{(\tilde{\eta}_1 \uplus \tilde{\eta}_2), \sigma, \mu \xrightarrow{\phi_1 \cup \phi_2} (\tilde{\eta}'_1 \uplus \tilde{\eta}'_2), \sigma'', \mu''} \;\lfloor\mathrm{C|Join}\rceil$$

Figure 4 Semantics of individual communications. The side condition (†) in rule $\lfloor$C|Join$\rceil$ reads: if a 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.

Rule $\lfloor$C|Com$\rceil$ allows some of the communications in $\tilde{\eta}$ to reduce only if the state of the automaton corresponding to connector γ allows it. The rule reads: under $\mathcal{G}$, triple $\tilde{\eta}\,\mathtt{thru}\,\gamma; C, \sigma, \mathcal{A}$ 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 Figure 4. They are obtained by considering the different possible cases for terms $\tilde{\eta}$ and matching them to appropriate constraints ϕ. Let $e \downarrow^{\sigma}_{\mathtt{p}} v$ denote that expression e evaluates to value v under σ (i.e., after substituting every free variable x in e by $\sigma(\mathtt{p}.x)$). In rule $\lfloor$C|SyncVal$\rceil$, an entire communication $\mathtt{p}\langle e\rangle$ `->` $\mathtt{q}.x$ is executed in one step. Accordingly, the label $\mathtt{p} \to \mathtt{q}$ denotes that the automaton should support a synchronous communication between $\mathtt{p}$ and $\mathtt{q}$. The state of the receiver $\mathtt{q}$ is updated with the value sent by $\mathtt{p}$. Rule $\lfloor$C|SendVal$\rceil$ applies in the case where the message from $\mathtt{p}$ should be stored in a memory cell m of the automaton (label $\mathtt{p} \to m$). This is used for asynchronous communications, where the message will be 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 ($\mathtt{q}.x?v$). This kind of runtime terms is handled by rule $\lfloor$C|RecvVal$\rceil$, whose label specifies that $\mathtt{q}$ should receive the message stored in some memory cell m. The premise of this rule checks that the value $\mathtt{q}$ is expecting to receive (v, defined in the choreography term) is the one stored in m.[3] Rules $\lfloor$C|SyncSel$\rceil$, $\lfloor$C|SendSel$\rceil$

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

and $\lfloor$C|RecvSel$\rceil$ deal with the case of label selection in a similar way. Rule $\lfloor$C|Mem$\rceil$ covers internal transitions in the automaton that only modify memory. Finally, rules $\lfloor$C|Mon$\rceil$ and $\lfloor$C|Join$\rceil$ extend this notion to transitions labelled by non-singleton sets. Rule $\lfloor$C|Mon$\rceil$ states that some communications in $\tilde{\eta}$ may not be executed at all (i.e., they are postponed until a later reduction). Rule $\lfloor$C|Join$\rceil$ allows executing several communications at the same time; note that the 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 8), 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 rule $\lfloor$C|Join$\rceil$ is of no consequence (building the final set ϕ in any order always yields the same final states σ' and μ' in rule $\lfloor$C|Com$\rceil$), 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 (†), which ensures that concurrent accesses to a memory cell are done in a consistent way.

▶ **Example 8** (Book sale). We formally derive the reductions in Example 7, using rule $\lfloor$C|Com$\rceil$ and Figure 4. Let $\mathcal{G}$, σ_0, σ_0', and $\mathcal{A}_0$ be defined as in Example 7.

For the first reduction, first, $\mathcal{A}_0(\mathsf{a2c}) = \langle 1, \{m \mapsto \bot\}\rangle$ (rule $\lfloor$C|Com$\rceil$, first premise). Next, automaton $\mathcal{G}(\mathsf{a2c})$ has one transition out of state 1, namely $1 \xrightarrow{\mathsf{a}\to m}_{\mathsf{a2c}} 2$ (rule $\lfloor$C|Com$\rceil$, third premise). Finally, using Figure 4, we need to derive $\mathsf{a}\langle \mathit{title}\rangle \texttt{ -> } \mathsf{c}, \sigma_0, \{m \mapsto \bot\} \xrightarrow{\mathsf{a}\to m} \dots$ (rule $\lfloor$C|Com$\rceil$, second premise); this follows from rule $\lfloor$C|SendVal$\rceil$. Thus, we derive:

$$\dfrac{\mathcal{A}_0(\mathsf{a2c}) = \langle 1, \{m \mapsto \bot\}\rangle \qquad \dfrac{\mathit{title} \downarrow^{\sigma_0}_{\mathsf{a}} \texttt{"foo"}}{\mathsf{a}\langle \mathit{title}\rangle \texttt{ -> } \mathsf{c}, \sigma_0, \{m \mapsto \bot\} \xrightarrow{\mathsf{a}\to m} \mathsf{c}.\mathit{title}?\texttt{"foo"}, \sigma_0, \{m \mapsto \texttt{"foo"}\}}\lfloor\text{C|SendVal}\rceil \qquad 1 \xrightarrow{\mathsf{a}\to m}_{\mathsf{a2c}} 2}{\begin{array}{rl} & \mathsf{a}\langle \mathit{title}\rangle \texttt{ -> } \mathsf{c}\ \texttt{thru}\ \mathsf{a2c}; C',\ \sigma_0,\ \{\mathsf{a2c} \mapsto \langle 1, \{m \mapsto \bot\}\rangle\} \cup \mathcal{A}_0^{\text{rest}} \\ \rightsquigarrow_{\mathcal{G}} & \mathsf{c}.\mathit{title}?\texttt{"foo"}\ \texttt{thru}\ \mathsf{a2c}; C',\ \sigma_0,\ \{\mathsf{a2c} \mapsto \langle 2, \{m \mapsto \texttt{"foo"}\}\rangle\} \cup \mathcal{A}_0^{\text{rest}} \end{array}}$$

Henceforth, let $\mathcal{A}_0' = \{\mathsf{a2c} \mapsto \langle 2, \{m \mapsto \texttt{"foo"}\}\rangle\} \cup \mathcal{A}_0^{\text{rest}}$.

For the second reduction, similarly, we derive:

$$\dfrac{\mathcal{A}_0'(\mathsf{a2c}) = \langle 2, \{m \mapsto \texttt{"foo"}\}\rangle \qquad \dfrac{\{m \mapsto \texttt{"foo"}\}(m) = \texttt{"foo"}}{\mathsf{c}.\mathit{title}?\texttt{"foo"}, \sigma_0, \{m \mapsto \texttt{"foo"}\} \xrightarrow{m\to\mathsf{c}} \emptyset, \sigma_0', \{m \mapsto \texttt{"foo"}\}}\lfloor\text{C|RecvVal}\rceil \qquad 2 \xrightarrow{m\to\mathsf{c}}_{\mathsf{a2c}} 1}{\begin{array}{rl} & \mathsf{c}.\mathit{title}?\texttt{"foo"}\ \texttt{thru}\ \mathsf{a2c}; C',\ \sigma_0,\ \{\mathsf{a2c} \mapsto \langle 2, \{m \mapsto \texttt{"foo"}\}\rangle\} \cup \mathcal{A}_0^{\text{rest}} \\ \rightsquigarrow_{\mathcal{G}} & \emptyset\ \texttt{thru}\ \mathsf{a2c}; C',\ \sigma_0',\ \{\mathsf{a2c} \mapsto \langle 1, \{m \mapsto \texttt{"foo"}\}\rangle\} \cup \mathcal{A}_0^{\text{rest}} \end{array}}$$

◀

▶ **Example 9** (Book sale). We formally derive the reduction of line 4 in Example 4. Let $\mathcal{G}$ denote the connector mapping in Example 6, let σ denote a choreography state function such that $\sigma(\mathsf{a}.\mathit{money}) = \texttt{\$10}$ and $\sigma(\mathsf{c}.\mathit{book}) = \texttt{foo.pdf}$, let $\sigma' = \sigma[\mathsf{b}.\mathit{money} \mapsto \texttt{\$10}]$, let $\sigma'' = \sigma'[\mathsf{s}.\mathit{book} \mapsto \texttt{foo.pdf}]$, and let $\mathcal{A} = \mathcal{A}^{\text{rest}} \cup \{\mathsf{ac2bs} \mapsto \{1, \emptyset\}\}$ denote an automaton state function, for some $\mathcal{A}^{\text{rest}}$. We derive:

$$\dfrac{\mathcal{A}(\mathsf{ac2bs}) = \langle 1, \emptyset\rangle \qquad \dfrac{\dfrac{\mathit{money} \downarrow^{\sigma}_{\mathsf{a}} \texttt{\$10}}{\{\mathsf{a}\langle \mathit{money}\rangle \texttt{ -> } \mathsf{b}\}, \sigma, \emptyset \xrightarrow{\mathsf{a}\to\mathsf{b}} \emptyset, \sigma', \emptyset} \qquad \dfrac{\mathit{book} \downarrow^{\sigma}_{\mathsf{c}} \texttt{foo.pdf}}{\{\mathsf{c}\langle \mathit{book}\rangle \texttt{ -> } \mathsf{s}\}, \sigma', \emptyset \xrightarrow{\mathsf{c}\to\mathsf{s}} \emptyset, \sigma'', \emptyset}\lfloor\text{C|SyncVal}\rceil}{\{\mathsf{a}\langle \mathit{money}\rangle \texttt{ -> } \mathsf{b}, \mathsf{c}\langle \mathit{book}\rangle \texttt{ -> } \mathsf{s}\}, \sigma, \emptyset \xrightarrow{\mathsf{a}\to\mathsf{b}\wedge\mathsf{c}\to\mathsf{s}} \emptyset, \sigma'', \emptyset}\lfloor\text{C|Join}\rceil \qquad 1 \xrightarrow{\mathsf{a}\to\mathsf{b}\wedge\mathsf{c}\to\mathsf{s}}_{\mathsf{ac2bs}} 1}{\begin{array}{rl} & \{\mathsf{a}\langle \mathit{money}\rangle \texttt{ -> } \mathsf{b}, \mathsf{c}\langle \mathit{book}\rangle \texttt{ -> } \mathsf{s}\}\ \texttt{thru}\ \mathsf{ac2bs}; \mathbf{0},\ \sigma,\ \mathcal{A} \\ \rightsquigarrow_{\mathcal{G}} & \emptyset\ \texttt{thru}\ \mathsf{ac2bs}; \mathbf{0},\ \sigma'',\ \mathcal{A} \end{array}}$$

$$\frac{i = 1 \text{ if } e \downarrow^{\sigma}_{\mathsf{p}} \mathit{true},\ \ i = 2 \text{ otherwise}}{\texttt{if}\ \mathsf{p}.e\ \texttt{then}\ C_1\ \texttt{else}\ C_2, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C_i, \sigma, \mathcal{A}}\ \lfloor\text{C}|\text{Cond}\rceil$$

$$\frac{C_1 \preceq C_2 \qquad C_2, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C_2', \sigma', \mathcal{A}' \qquad C_2' \preceq C_1'}{C_1, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C_1', \sigma', \mathcal{A}'}\ \lfloor\text{C}|\text{Struct}\rceil$$

$$\frac{C_1, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C_1', \sigma', \mathcal{A}}{\texttt{def}\ X = C_2\ \texttt{in}\ C_1, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} \texttt{def}\ X = C_2\ \texttt{in}\ C_1', \sigma', \mathcal{A}'}\ \lfloor\text{C}|\text{Ctx}\rceil$$

Figure 5 Cho-Reo-graphy, semantics.

$$\frac{\mathsf{pn}(\tilde{\eta}) \cap \mathsf{pn}(\widetilde{\eta'}) = \emptyset \qquad \gamma \neq \gamma'}{\left(\tilde{\eta}\ \texttt{thru}\ \gamma; \widetilde{\eta'}\ \texttt{thru}\ \gamma'\right) \equiv \left(\widetilde{\eta'}\ \texttt{thru}\ \gamma'; \tilde{\eta}\ \texttt{thru}\ \gamma\right)}\ \lfloor\text{C}|\text{Eta-Eta}\rceil$$

$$\frac{\mathsf{pn}(\widetilde{\eta_1}) \cap \mathsf{pn}(\widetilde{\eta_2}) = \emptyset}{(\widetilde{\eta_1}\ \texttt{thru}\ \gamma; \widetilde{\eta_2}\ \texttt{thru}\ \gamma) \equiv (\widetilde{\eta_1} \cup \widetilde{\eta_2})\ \texttt{thru}\ \gamma}\ \lfloor\text{C}|\text{Eta-Split}\rceil$$

$$\frac{\mathsf{p} \notin \mathsf{pn}(\tilde{\eta})}{(\texttt{if}\ \mathsf{p}.e\ \texttt{then}\ (\tilde{\eta}\ \texttt{thru}\ \gamma; C_1)\ \texttt{else}\ (\tilde{\eta}\ \texttt{thru}\ \gamma; C_2)) \equiv (\tilde{\eta}\ \texttt{thru}\ \gamma; \texttt{if}\ \mathsf{p}.e\ \texttt{then}\ C_1\ \texttt{else}\ C_2)}\ \lfloor\text{C}|\text{Eta-Cond}\rceil$$

$$\frac{\mathsf{pn}(C_i) \cap \mathsf{pn}(\tilde{\eta}) = \emptyset}{(\texttt{def}\ X = C_2\ \texttt{in}\ (\tilde{\eta}\ \texttt{thru}\ \gamma; C_1)) \equiv (\tilde{\eta}\ \texttt{thru}\ \gamma; \texttt{def}\ X = C_2\ \texttt{in}\ C_1)}\ \lfloor\text{C}|\text{Eta-Rec}\rceil$$

$$\frac{\mathsf{p} \neq \mathsf{q}}{\begin{array}{c}\texttt{if}\ \mathsf{p}.e\ \texttt{then}\ (\texttt{if}\ \mathsf{q}.e'\ \texttt{then}\ C_1\ \texttt{else}\ C_2)\ \texttt{else}\ (\texttt{if}\ \mathsf{q}.e'\ \texttt{then}\ C_1'\ \texttt{else}\ C_2')\\ \equiv \\ \texttt{if}\ \mathsf{q}.e'\ \texttt{then}\ (\texttt{if}\ \mathsf{p}.e\ \texttt{then}\ C_1\ \texttt{else}\ C_1')\ \texttt{else}\ (\texttt{if}\ \mathsf{p}.e\ \texttt{then}\ C_2\ \texttt{else}\ C_2')\end{array}}\ \lfloor\text{C}|\text{Cond-Cond}\rceil$$

$$\frac{}{(\texttt{def}\ X = C_2\ \texttt{in}\ C_1[X]) \preceq (\texttt{def}\ X = C_2\ \texttt{in}\ C_1[C_2])}\ \lfloor\text{C}|\text{Unfold}\rceil$$

$$\frac{}{\emptyset\ \texttt{thru}\ \gamma; C \preceq C}\ \lfloor\text{C}|\text{EtaEnd}\rceil \qquad \frac{}{(\texttt{def}\ X = C\ \texttt{in}\ \mathbf{0}) \preceq \mathbf{0}}\ \lfloor\text{C}|\text{ProcEnd}\rceil$$

Figure 6 Cho-Reo-graphy, structural precongruence.

◀

Rule ⌊C|Com⌉ is the only rule in the semantics that can cause a choreography to get stuck; we discuss this in more detail shortly.

The remaining rules defining $\rightsquigarrow_{\mathcal{G}}$ are standard from other choreography calculi (e.g., [22]), and they are given in Figure 5. Rule ⌊C|Struct⌉ uses a structural precongruence that allows for actions to be swapped if they do not interfere. Its definition, given in Figure 6 (where $\mathsf{pn}(\tilde{\eta})$ denotes the process names that occur in $\tilde{\eta}$), contains two interesting new rules: ⌊C|Eta-Split⌉ and ⌊C|EtaEnd⌉. 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⌉. (The example below illustrates this interaction; see also [21] for a similar discussion: this rule is needed whenever one $\tilde{\eta}$ can specify several communications.) Rule ⌊C|EtaEnd⌉ removes completed interactions from the head of a choreography.

▶ **Example 10.** Consider the following choreography, where for simplicity we abstract from the actual values being communicated.

$$C \equiv \{\mathtt{p}\ \texttt{->}\ \mathtt{q}, \mathtt{r}\ \texttt{->}\ \mathtt{s}\}\ \texttt{thru}\ \gamma; \{\mathtt{p}\ \texttt{->}\ \mathtt{q}, \mathtt{t}\ \texttt{->}\ \mathtt{v}\}\ \texttt{thru}\ \gamma'$$

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 $\lfloor$C|Eta-Split$\rceil$ and $\lfloor$C|Eta-Eta$\rceil$ to exchange actions, as follows.

$$\begin{aligned} C &\equiv \{\texttt{p -> q}, \texttt{r -> s}\}\,\texttt{thru}\,\gamma; \{\texttt{p -> q}, \texttt{t -> v}\}\,\texttt{thru}\,\gamma' \\ &\equiv \{\texttt{p -> q}, \texttt{r -> s}\}\,\texttt{thru}\,\gamma; \texttt{t -> v}\,\texttt{thru}\,\gamma'; \texttt{p -> q}\,\texttt{thru}\,\gamma' && \text{by } \lfloor\text{C|Eta-Split}\rceil \\ &\equiv \texttt{t -> v}\,\texttt{thru}\,\gamma'; \{\texttt{p -> q}, \texttt{r -> s}\}\,\texttt{thru}\,\gamma; \texttt{p -> q}\,\texttt{thru}\,\gamma' && \text{by } \lfloor\text{C|Eta-Eta}\rceil \end{aligned}$$

◀

4.3 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. The following example further motivates this feature and illustrates its use.

▶ **Example 11** (Book sale). The original book sale scenario (Examples 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 Example 6, as follows:

$$\mathcal{G} := \mathcal{G}[\mathsf{ac2bs} \mapsto \begin{tikzcd} \overline{2} \arrow[r, bend right, "\mathtt{a} \to \mathtt{b}"'] & 1 \arrow[l, bend right, "\mathtt{c} \to \mathtt{s}"'] \arrow[r, bend right, "\mathtt{a} \to \mathtt{b}"'] & 2 \arrow[l, bend right, "\mathtt{c} \to \mathtt{s}"'] \\ & {} \arrow[u] & \end{tikzcd}]$$

Thus, we updated the mapping for ac2bs; for all other connectors, the mapping remains the same as in Example 6. 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 $\overline{2}$).[4]

Redefining the connector mapping for ac2bs is the only change we need to make: the choreography itself is *exactly* the same as in Example 4. This means that also the first reductions remain *exactly* the same as in Examples 7 and 8. By contrast, the reduction in Example 9 is no longer valid, as it relies on the semantics of ac2bs. To show the difference formally, let σ, σ', σ'', and $\mathcal{A}$ be defined as in Example 9. Let also $\mathcal{A}' = \mathcal{A}[\mathsf{ac2bs} \mapsto \langle 2, \emptyset\rangle]$ and $\mathcal{A}'' = \mathcal{A}$. We derive:

$$\dfrac{\mathcal{A}(\mathsf{ac2bs}) = \langle 1, \emptyset\rangle \qquad \dfrac{\dfrac{\mathit{money} \downarrow^{\sigma}_{\mathtt{a}} \mathsf{\$10}}{\{\mathtt{a}\langle \mathit{money}\rangle\ \texttt{->}\ \mathtt{b}\}, \sigma, \emptyset \xrightarrow{\mathtt{a}\to\mathtt{b}} \emptyset, \sigma', \emptyset}\lfloor\text{C|SyncVal}\rceil}{\{\mathtt{a}\langle \mathit{money}\rangle\ \texttt{->}\ \mathtt{b}, \mathtt{c}\langle \mathit{book}\rangle\ \texttt{->}\ \mathtt{s}\}, \sigma, \emptyset \xrightarrow{\mathtt{a}\to\mathtt{b}} \{\mathtt{c}\langle \mathit{book}\rangle\ \texttt{->}\ \mathtt{s}\}, \sigma', \emptyset}\lfloor\text{C|Mon}\rceil \qquad 1 \xrightarrow{\mathtt{a}\to\mathtt{b}}_{\mathsf{ac2bs}} 2}{\begin{aligned} &\{\mathtt{a}\langle \mathit{money}\rangle\ \texttt{->}\ \mathtt{b}, \mathtt{c}\langle \mathit{book}\rangle\ \texttt{->}\ \mathtt{s}\}\,\texttt{thru}\,\mathsf{ac2bs}; \mathbf{0},\ \sigma,\ \mathcal{A} \\ \rightsquigarrow_{\mathcal{G}} \quad &\{\mathtt{c}\langle \mathit{book}\rangle\ \texttt{->}\ \mathtt{s}\}\,\texttt{thru}\,\mathsf{ac2bs}; \mathbf{0},\ \sigma',\ \mathcal{A}' \end{aligned}}$$

[4] The 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).

Next, we derive:

$$\frac{\mathcal{A}(\mathsf{ac2bs}) = \langle 2, \emptyset \rangle \qquad \dfrac{book \downarrow^{\sigma}_{\mathsf{c}} \texttt{foo.pdf}}{\{\mathtt{c}\langle book \rangle \texttt{ -> s}\}, \sigma', \emptyset \xrightarrow{\mathsf{c} \to \mathsf{s}} \emptyset, \sigma'', \emptyset}\lfloor \text{C}|\text{SyncVal} \rceil \qquad 2 \xrightarrow{\mathsf{c} \to \mathsf{s}}_{\mathsf{ac2bs}} 1}{\begin{array}{rc} & \{\mathtt{c}\langle book \rangle \texttt{ -> s}\}\,\texttt{thru}\,\mathsf{ac2bs}; \mathbf{0},\ \sigma',\ \mathcal{A}' \\ \rightsquigarrow_{\mathcal{G}} & \emptyset\,\texttt{thru}\,\mathsf{ac2bs}; \mathbf{0},\ \sigma'',\ \mathcal{A}'' \end{array}}$$

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 (Example 9). Similarly, we can derive two separate reductions whereby Carol and the shipper communicate first, followed by Alice and the bank. ◀

▶ **Example 12.** The previous example works also "in the opposite direction", from a trusting Alice and Carol (using connector mapping $\mathcal{G}$ in Example 11) to cautious ones (using connector mapping $\mathcal{G}$ in Example 6).

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.

$$\begin{tikzcd} \mathcal{G} := \mathcal{G}[\mathsf{ac2bs} \mapsto & \textcircled{1} \arrow[r, bend right, "\mathtt{a} \to \mathtt{b}"'] & \textcircled{2} \arrow[l, bend right, "\mathtt{c} \to \mathtt{s}"']] \\ & \arrow[u] & \end{tikzcd}$$

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

▶ **Example 13.** Note that the changes to connector mappings in Examples 11 and 12 would still be possible if the programmer had written, e.g.,

$$\ldots; \{\mathtt{a}\langle money \rangle \texttt{ -> b}\}\,\texttt{thru}\,\mathsf{ac2bs}; \{\mathtt{c}\langle book \rangle \texttt{ -> s}\}\,\texttt{thru}\,\mathsf{ac2bs}; \mathbf{0}$$

instead of the choreography in Example 4. Indeed, these two choreographies are equivalent due to the congruence rule $\lfloor$C|Eta-Split$\rceil$, 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 $\mathtt{a}\langle money \rangle$ `->` b and $\mathtt{c}\langle book \rangle$ `->` 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. ◀

4.4 Deadlock-freedom

Rule $\lfloor$C|Com$\rceil$ is the only reduction rule in the semantics that can cause a choreography to get stuck: in choreography $\tilde{\eta}\,\texttt{thru}\,\gamma; C$, 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]

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

In more detail, the first premise in rule $\lfloor\mathrm{C|Com}\rceil$ is always true (assuming the connector mapping is defined for all connector names in $\tilde{\eta}\,\mathtt{thru}\,\gamma; C$). This gives us unique bindings for s and μ. The third premise in $\lfloor\mathrm{C|Com}\rceil$ 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 $\lfloor\mathrm{C|Com}\rceil$ 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 Figure 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 (Example 14 below), or the same processes but in different send/receive pairs (Example 15 below); the latter happens if $\tilde{\eta}$ contains only asynchronous receives for which rules $\lfloor\mathrm{C|RecvVal}\rceil$ and $\lfloor\mathrm{C|RecvSel}\rceil$ in Figure 4 are inapplicable (Example 16 below). We now exemplify these cases.

▶ **Example 14** (Book sale)**.** Suppose we mistakenly redefine the connector mapping $\mathcal{G}$ in Example 6 as follows (cf. Example 11; i.e., the boxed label is wrong):

$$\begin{tikzcd}
\mathcal{G} := \mathcal{G}[\mathsf{ac2bs} \mapsto & \bar{2} \arrow[r, bend right, "\mathtt{a} \to \mathtt{b}"'] & 1 \arrow[l, bend right, "\mathtt{c} \to \mathtt{s}"'] \arrow[r, bend right, "\mathtt{a} \to \mathtt{b}"'] & 2 \arrow[l, bend right, "\boxed{\mathtt{a} \to \mathtt{b}}"'] & {]} \\
& & {} \arrow[u] & &
\end{tikzcd}$$

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 Example 11. 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 Example 11 is still valid, but the second derivation is not: rule $\lfloor\mathrm{C|SyncVal}\rceil$ is still applied to derive $\{\mathtt{c}\langle \mathit{book}\rangle \mathtt{\ ->\ s}\}, \sigma', \emptyset \xrightarrow{\mathtt{c} \to \mathtt{s}} \emptyset, \sigma'', \emptyset$ to fulfill the second premise of rule $\lfloor\mathrm{C|Com}\rceil$, but ac2bs has no transition in state 2 labelled with $\mathtt{c} \to \mathtt{s}$. As there are no other derivations to fulfill the second premise of rule $\lfloor\mathrm{C|Com}\rceil$, the choreography gets stuck. ◀

▶ **Example 15** (Book sale)**.** Suppose we mistakenly redefine connector mapping $\mathcal{G}$ in Example 6 as follows (i.e., the boxed process names are wrong/swapped):

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

Formally, automaton $\mathcal{G}(\mathsf{ac2bs})$ has the following transition: $1 \xrightarrow{\mathtt{a} \to \mathtt{s} \wedge \mathtt{c} \to \mathtt{b}}_{\mathsf{ac2bs}} 1$. 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 Example 9 is no longer valid: rules $\lfloor\mathrm{C|SyncVal}\rceil$ and $\lfloor\mathrm{C|Join}\rceil$ are still applied to derive $\{\mathtt{a}\langle \mathit{money}\rangle \mathtt{\ ->\ b}, \mathtt{c}\langle \mathit{book}\rangle \mathtt{\ ->\ s}\}, \sigma, \emptyset \xrightarrow{\mathtt{a} \to \mathtt{b} \wedge \mathtt{c} \to \mathtt{s}} \emptyset, \sigma'', \emptyset$ to fulfill the second premise of rule $\lfloor\mathrm{C|Com}\rceil$, but ac2bs has no transition labelled with $\mathtt{a} \to \mathtt{b} \wedge \mathtt{c} \to \mathtt{s}$. As there are no other derivations to fulfill the second premise of rule $\lfloor\mathrm{C|Com}\rceil$, the choreography gets stuck. ◀

▶ **Example 16** (Book sale)**.** Suppose we mistakenly redefine the connector mapping $\mathcal{G}$ in Example 6 as follows (i.e., the boxed process names are wrong/swapped):

$$\begin{tikzcd}
\mathcal{G} := \mathcal{G}[\mathsf{ac2bs} \mapsto & 1 \arrow[r, "\mathtt{a} \to m_1"] & 2 \arrow[r, "\mathtt{c} \to m_2"] & 3 \arrow[r, "m_1 \to \boxed{\mathtt{s}}"] & 4 \arrow[lll, bend left, "m_2 \to \boxed{\mathtt{b}}"] & {]} \\
& {} \arrow[u] & & & &
\end{tikzcd}$$

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 σ, σ', σ'', and $\mathcal{A}$ be defined as in Example 9. Furthermore, let $\mu = \{m_1 \mapsto \bot, m_2 \mapsto \bot\}$, let $\mu' = \{m_1 \mapsto \texttt{\$10}, m_2 \mapsto \bot\}$, and let $\mu'' = \{m_1 \mapsto \texttt{\$10}, m_2 \mapsto \texttt{foo.pdf}\}$. The following reductions can be derived using rule $\lfloor$C$|$Com$\rceil$:

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

At this point, the choreography gets stuck: there are no derivations to fulfill the second premise of $\lfloor$C$|$Com$\rceil$. To see this, note that only rule $\lfloor$C$|$RecvVal$\rceil$ may be applicable (together with rule $\lfloor$C$|$Join$\rceil$), but $\mu''(m_2) = \texttt{foo.pdf}$, whereas the choreography states $\mathsf{b}.money?\texttt{\$10}$. 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. ◀

None of these examples can be constructed in existing choreography models: in existing models, 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 17.** Connector mapping $\mathcal{G}$ in automaton state function $\mathcal{A}$ *respects* choreography C if: for every σ, $\tilde{\eta}$, γ, σ' and $\mathcal{A}'$, if $C, \sigma, \mathcal{A} \rightsquigarrow^*_{\mathcal{G}} \tilde{\eta}\,\texttt{thru}\,\gamma; C', \sigma', \mathcal{A}'$, then there exist σ'' and $\mathcal{A}''$ such that $\tilde{\eta}\,\texttt{thru}\,\gamma; C', \sigma', \mathcal{A}' \rightsquigarrow^*_{\mathcal{G}} C', \sigma'', \mathcal{A}''$.

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 $\mathcal{G}$).

By definition, respectfulness is equivalent to deadlock-freedom.

▶ **Definition 18.** $C, \sigma, \mathcal{A}_0$ 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 19** (Undecidability of Deadlock-Freedom). *In general, it is undecidable whether a connector mapping $\mathcal{G}$ respects a choreography C.*

▶ Remark. Undecidability of deadlock-freedom arises exactly 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 choreography 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 Theorem 19 relies on the existence of a communication action that does not respect $\mathcal{G}$. Such an action does not exist in previous models. ◀

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

$$\dfrac{\mathcal{A}(\gamma) = \langle s, \mu \rangle \qquad \left\{ \Gamma \vdash^{\mathcal{G}}_{\mathcal{A}[\gamma \mapsto \langle s', \mu' \rangle]} \tilde{\eta}' \, \mathtt{thru}\, \gamma; C' \;\middle|\; \begin{array}{c} C \equiv \tilde{\eta}\, \mathtt{thru}\, \gamma; C' \\ \text{and } \tilde{\eta}, \mu \xrightarrow{\phi} \tilde{\eta}', \mu' \\ \text{and } s \xrightarrow{\phi}_{\gamma} s' \end{array} \right\} \quad (\dagger)}{\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} C} \lfloor \mathrm{CC|Com} \rceil$$

$$\dfrac{\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} C}{\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} \emptyset\, \mathtt{thru}\, \gamma; C} \lfloor \mathrm{CC|Done} \rceil \qquad \dfrac{}{\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} \mathbf{0}} \lfloor \mathrm{CC|Nil} \rceil \qquad \dfrac{\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} C_1 \quad \Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} C_2}{\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} \mathtt{if}\, \mathsf{p}.e\, \mathtt{then}\, C_1\, \mathtt{else}\, C_2} \lfloor \mathrm{CC|Cond} \rceil$$

$$\dfrac{\Gamma, (X : \mathcal{A}_X) \vdash^{\mathcal{G}}_{\mathcal{A}} C_1 \quad \Gamma, (X : \mathcal{A}_X) \vdash^{\mathcal{G}}_{\mathcal{A}} C_2}{\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} \mathtt{def}\, X = C_2\, \mathtt{in}\, C_1} \lfloor \mathrm{CC|Def} \rceil \qquad \dfrac{(X : \mathcal{A}) \in \Gamma}{\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} X} \lfloor \mathrm{CC|Call} \rceil$$

Figure 7 Cho-Reo-graphy, compatibility relation. The side condition (†) reads: the set of judgments on the left is nonempty. By abuse of notation, we use set builder notation in rule $\lfloor \mathrm{CC|Com} \rceil$ to indicate that all judgments in the set must be true.

▶ **Definition 20.** 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 $\vdash^{\mathcal{G}}_{\mathcal{A}} C$, where the relation $\vdash$ is defined by the rules in Figure 7. We say C and $\mathcal{G}$ are compatible, written $\vdash^{\mathcal{G}} C$, if $\vdash^{\mathcal{G}}_{\mathcal{A}_0} C$ with $\mathcal{A}_0$ as in Definition 17.

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 Figure 4, $\tilde{\eta}, \mu \xrightarrow{\phi} \tilde{\eta}', \mu'$. The latter models a symbolic execution of communications; it is defined as in Figure 4, with two differences: (i) σ is removed from the domain of the reduction and (ii) in rule $\lfloor C|\mathit{SendVal} \rceil$, v is a fresh token value.

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 $\lfloor \mathrm{CC|Def} \rceil$ and $\lfloor \mathrm{CC|Call} \rceil$ (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 $\lfloor \mathrm{CC|Com} \rceil$, 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. Theorem 24).

▶ Remark. Compatibility can become more robust/modular by disregarding connectors not occurring in procedure bodies in $\lfloor \mathrm{CC|Def} \rceil$. We chose our current formulation for simplicity. ◀

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

▶ **Example 21** (Book sale). We illustrate how compatibility works in the context of our running example by revisiting choreography $\{\mathsf{a}\langle \mathit{money} \rangle \texttt{ -> } \mathsf{b}, \mathsf{c}\langle \mathit{book} \rangle \texttt{ -> } \mathsf{s}\}\, \mathtt{thru}\, \mathsf{ac2bs}; \mathbf{0}$ with five different connector mappings from previous examples.

✓ Let $\mathcal{G}$ and $\mathcal{A}$ be defined as in Example 9. Using Figure 7, we derive:

$$\dfrac{\mathcal{A}(\mathsf{ac2bs}) = \langle 1, \emptyset \rangle \qquad \dfrac{\dfrac{}{\cdot \vdash^{\mathcal{G}}_{\mathcal{A}[\mathsf{ac2bs} \mapsto \langle 1, \emptyset \rangle]} \mathbf{0}} \lfloor \mathrm{CC|Nil} \rceil}{\cdot \vdash^{\mathcal{G}}_{\mathcal{A}[\mathsf{ac2bs} \mapsto \langle 1, \emptyset \rangle]} \emptyset\, \mathtt{thru}\, \mathsf{ac2bs}; \mathbf{0}} \lfloor \mathrm{CC|Done} \rceil}{\cdot \vdash^{\mathcal{G}}_{\mathcal{A}} \{\mathsf{a}\langle \mathit{money} \rangle \texttt{ -> } \mathsf{b}, \mathsf{c}\langle \mathit{book} \rangle \texttt{ -> } \mathsf{s}\}\, \mathtt{thru}\, \mathsf{ac2bs}; \mathbf{0}} \lfloor \mathrm{CC|Com} \rceil$$

Thus, the choreography and the connector mapping are compatible. Corollary 25 below implies that the connector mapping respects the choreography.

✓ Let $\mathcal{G}$ and $\mathcal{A}$ be defined as in Example 11. Furthermore, let $\mathcal{A}'_2 = \mathcal{A}[\mathsf{ac2bs} \mapsto \langle 2, \emptyset \rangle]$, let $\mathcal{A}'_{\overline{2}} = \mathcal{A}[\mathsf{ac2bs} \mapsto \langle \overline{2}, \emptyset \rangle]$, and let $\mathcal{A}'' = \mathcal{A}'_2[\mathsf{ac2bs} \mapsto \langle 1, \emptyset \rangle] = \mathcal{A}'_{\overline{2}}[\mathsf{ac2bs} \mapsto \langle 1, \emptyset \rangle] = \mathcal{A}$. Using Figure 7, we derive:

$$\dfrac{\begin{matrix}\mathcal{A}(\mathsf{ac2bs}) \\ = \langle 1, \emptyset \rangle\end{matrix} \quad \dfrac{\begin{matrix}\mathcal{A}'_{\overline{2}}(\mathsf{ac2bs}) \\ = \langle \overline{2}, \emptyset \rangle\end{matrix} \quad \dfrac{\dfrac{}{\cdot \vdash^{\mathcal{G}}_{\mathcal{A}''} \mathbf{0}}}{\begin{matrix}\cdot \vdash^{\mathcal{G}}_{\mathcal{A}''} \emptyset\, \mathtt{thru} \\ \mathsf{ac2bs}; \mathbf{0}\end{matrix}}}{\begin{matrix}\cdot \vdash^{\mathcal{G}}_{\mathcal{A}'_{\overline{2}}} \{\mathtt{a}\langle \mathit{money} \rangle \texttt{ -> } \mathtt{b}\} \\ \mathtt{thru}\, \mathsf{ac2bs}; \mathbf{0}\end{matrix}} \quad \dfrac{\begin{matrix}\mathcal{A}'_2(\mathsf{ac2bs}) \\ = \langle 2, \emptyset \rangle\end{matrix} \quad \dfrac{\dfrac{}{\cdot \vdash^{\mathcal{G}}_{\mathcal{A}''} \mathbf{0}}\lfloor \text{CC}|\text{Nil} \rceil}{\begin{matrix}\cdot \vdash^{\mathcal{G}}_{\mathcal{A}''} \emptyset\, \mathtt{thru} \\ \mathsf{ac2bs}; \mathbf{0}\end{matrix}}\lfloor \text{CC}|\text{Done} \rceil}{\begin{matrix}\cdot \vdash^{\mathcal{G}}_{\mathcal{A}'_2} \{\mathtt{c}\langle \mathit{book} \rangle \texttt{ -> } \mathtt{s}\} \\ \mathtt{thru}\, \mathsf{ac2bs}; \mathbf{0}\end{matrix}}\lfloor \text{CC}|\text{Com} \rceil}{\cdot \vdash^{\mathcal{G}}_{\mathcal{A}} \{\mathtt{a}\langle \mathit{money} \rangle \texttt{ -> } \mathtt{b}, \mathtt{c}\langle \mathit{book} \rangle \texttt{ -> } \mathtt{s}\}\, \mathtt{thru}\, \mathsf{ac2bs}; \mathbf{0}}\lfloor \text{CC}|\text{Com} \rceil$$

The bottom application of rule $\lfloor$CC|Com$\rceil$ requires two subderivations: one to cover the case where connector $\mathsf{ac2bs}$ makes a transition to state $\overline{2}$ (left subderivation), and another to cover the case where $\mathsf{ac2bs}$ makes a transition to state 2 (right subderivation). In both cases, we have compatibility.

Thus, the choreography and the connector mapping are compatible. Corollary 25 below implies that the connector mapping respects the choreography.

✗ Let $\mathcal{G}$ and $\mathcal{A}$ be defined as in Example 14. Furthermore, let $\mathcal{A}'_2 = \mathcal{A}[\mathsf{ac2bs} \mapsto \langle 2, \emptyset \rangle]$, and let $\mathcal{A}'_{\overline{2}} = \mathcal{A}[\mathsf{ac2bs} \mapsto \langle \overline{2}, \emptyset \rangle]$. Using Figure 7, we attempt:

$$\dfrac{\begin{matrix}\mathcal{A}(\mathsf{ac2bs}) \\ = \langle 1, \emptyset \rangle\end{matrix} \quad \dfrac{\begin{matrix}\mathcal{A}'_{\overline{2}}(\mathsf{ac2bs}) \\ = \langle \overline{2}, \emptyset \rangle\end{matrix} \quad \dfrac{\dfrac{}{\cdot \vdash^{\mathcal{G}}_{\mathcal{A}''} \mathbf{0}}}{\begin{matrix}\cdot \vdash^{\mathcal{G}}_{\mathcal{A}''} \emptyset\, \mathtt{thru} \\ \mathsf{ac2bs}; \mathbf{0}\end{matrix}}}{\begin{matrix}\cdot \vdash^{\mathcal{G}}_{\mathcal{A}'_{\overline{2}}} \{\mathtt{a}\langle \mathit{money} \rangle \texttt{ -> } \mathtt{b}\} \\ \mathtt{thru}\, \mathsf{ac2bs}; \mathbf{0}\end{matrix}} \quad \dfrac{\begin{matrix}\mathcal{A}'_2(\mathsf{ac2bs}) \\ = \langle 2, \emptyset \rangle\end{matrix} \quad \boxed{}}{\begin{matrix}\cdot \vdash^{\mathcal{G}}_{\mathcal{A}'_2} \{\mathtt{c}\langle \mathit{book} \rangle \texttt{ -> } \mathtt{s}\} \\ \mathtt{thru}\, \mathsf{ac2bs}; \mathbf{0}\end{matrix}}\lfloor \text{CC}|\text{Com} \rceil}{\cdot \vdash^{\mathcal{G}}_{\mathcal{A}} \{\mathtt{a}\langle \mathit{money} \rangle \texttt{ -> } \mathtt{b}, \mathtt{c}\langle \mathit{book} \rangle \texttt{ -> } \mathtt{s}\}\, \mathtt{thru}\, \mathsf{ac2bs}; \mathbf{0}}\lfloor \text{CC}|\text{Com} \rceil$$

This attempted derivation is the same as in the second ✓-item, except the subderivation inside the box has become invalid: under our current $\mathcal{G}$, connector $\mathsf{ac2bs}$ in state 2 has no transition labelled with $\mathtt{c} \to \mathtt{s}$.

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

✗ Let $\mathcal{G}$ and $\mathcal{A}$ be defined as in Example 15. Using Figure 7, we attempt:

$$\dfrac{\mathcal{A}(\mathsf{ac2bs}) = \langle 1, \emptyset \rangle \quad \boxed{}}{\cdot \vdash^{\mathcal{G}}_{\mathcal{A}} \{\mathtt{a}\langle \mathit{money} \rangle \texttt{ -> } \mathtt{b}, \mathtt{c}\langle \mathit{book} \rangle \texttt{ -> } \mathtt{s}\}\, \mathtt{thru}\, \mathsf{ac2bs}; \mathbf{0}}\lfloor \text{CC}|\text{Com} \rceil$$

This attempted derivation is the same as in the first ✓-item, except the subderivation inside the box has become invalid: under our current $\mathcal{G}$, connector $\mathsf{ac2bs}$ in state 1 has no transition labelled with $\mathtt{a} \to \mathtt{b} \wedge \mathtt{c} \to \mathtt{s}$.

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

✗ Let $\mathcal{G}$ and $\mathcal{A}$ be defined as in Example 16. Furthermore, let $\diamondsuit$ and $\heartsuit$ denote two fresh token values, let $\mu = \{m_1 \mapsto \bot, m_2 \mapsto \bot\}$, let $\mu' = \{m_1 \mapsto \diamondsuit, m_2 \mapsto \bot\}$, let

$\mu'' = \{m_1 \mapsto \Diamond, m_2 \mapsto \heartsuit\}$, let $\mathcal{A}' = \mathcal{A}[\mathsf{ac2bs} \mapsto \langle 2, \mu' \rangle]$, and let $\mathcal{A}'' = \mathcal{A}[\mathsf{ac2bs} \mapsto \langle 2, \mu'' \rangle]$. Using Figure 7, we attempt:

$$\dfrac{\begin{array}{c}\mathcal{A}(\mathsf{ac2bs})\\ = \langle 1,\mu\rangle\end{array} \quad \dfrac{\begin{array}{c}\mathcal{A}'(\mathsf{ac2bs})\\ = \langle 2,\mu'\rangle\end{array} \quad \dfrac{\begin{array}{c}\mathcal{A}''(\mathsf{ac2bs})\\ = \langle 3,\mu''\rangle\end{array} \quad \boxed{}}{\begin{array}{c}\cdot \vdash^{\mathcal{G}}_{\mathcal{A}''} \{\mathtt{b}.\mathit{money}?\Diamond, \mathtt{s}.\mathit{book}?\heartsuit\}\\ \mathtt{thru}\ \mathsf{ac2bs}; \mathbf{0}\end{array}}\lfloor\mathrm{CC}|\mathrm{Com}\rceil}{\begin{array}{c}\cdot \vdash^{\mathcal{G}}_{\mathcal{A}'} \{\mathtt{b}.\mathit{money}?\Diamond, \mathtt{c}\langle \mathit{book}\rangle \texttt{ -> } \mathtt{s}\}\\ \mathtt{thru}\ \mathsf{ac2bs}; \mathbf{0}\end{array}}\lfloor\mathrm{CC}|\mathrm{Com}\rceil}{\cdot \vdash^{\mathcal{G}}_{\mathcal{A}} \{\mathtt{a}\langle \mathit{money}\rangle \texttt{ -> } \mathtt{b}, \mathtt{c}\langle \mathit{book}\rangle \texttt{ -> } \mathtt{s}\}\ \mathtt{thru}\ \mathsf{ac2bs}; \mathbf{0}}\lfloor\mathrm{CC}|\mathrm{Com}\rceil$$

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 $\mathsf{ac2bs}$ in state 3 has no transition labelled with $m_1 \to \mathsf{b}$.
Thus, the choreography and the connector mapping are incompatible. In fact, in this case, the choreography deadlocks under $\mathcal{G}$. ◀

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 22.** Let C be the simple choreography:

$$\mathtt{def}\ X = \mathtt{p} \texttt{ -> } \mathtt{q}\ \mathtt{thru}\ \gamma; \mathtt{p} \texttt{ -> } \mathtt{q}\ \mathtt{thru}\ \gamma; \mathtt{r} \texttt{ -> } \mathtt{s}\ \mathtt{thru}\ \gamma; X\ \mathtt{in}\ X$$

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

$$\begin{array}{lll} & \mathtt{p} \texttt{ -> } \mathtt{q}\ \mathtt{thru}\ \gamma; \mathtt{p} \texttt{ -> } \mathtt{q}\ \mathtt{thru}\ \gamma; \mathtt{r} \texttt{ -> } \mathtt{s}\ \mathtt{thru}\ \gamma & \\ \equiv & \mathtt{p} \texttt{ -> } \mathtt{q}\ \mathtt{thru}\ \gamma; \{\mathtt{p} \texttt{ -> } \mathtt{q}, \mathtt{r} \texttt{ -> } \mathtt{s}\}\ \mathtt{thru}\ \gamma & \text{by } \lfloor\mathrm{C}|\text{Eta-Split}\rceil \\ \equiv & \mathtt{p} \texttt{ -> } \mathtt{q}\ \mathtt{thru}\ \gamma; \mathtt{r} \texttt{ -> } \mathtt{s}\ \mathtt{thru}\ \gamma; \mathtt{p} \texttt{ -> } \mathtt{q}\ \mathtt{thru}\ \gamma & \text{by } \lfloor\mathrm{C}|\text{Eta-Split}\rceil \end{array}$$

However, $\not\vdash^{\mathcal{G}} C$, since the second communication from $\mathtt{p}$ to $\mathtt{q}$ 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 $\mathtt{p}$ to $\mathtt{q}$ and one from $\mathtt{r}$ to $\mathtt{s}$) differs from the recursive structure of $\mathcal{G}(\gamma)$ (one communication between each pair of processes). ◀

▶ **Example 23.** Consider now the choreography C defined as

$$\mathtt{def}\ X = \mathtt{p} \texttt{ -> } \mathtt{q}\ \mathtt{thru}\ \gamma; \mathtt{r} \texttt{ -> } \mathtt{s}\ \mathtt{thru}\ \gamma; X\ \mathtt{in}\ X$$

where $\mathcal{G}(\gamma)$ only allows communications from $\mathtt{p}$ to $\mathtt{q}$. Again C is deadlock-free, since structural precongruence allows the communications from $\mathtt{r}$ to $\mathtt{s}$ 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. ◀

▶ **Theorem 24** (Preservation of Compatibility). *Let C and C' be choreographies, $\mathcal{G}$ be a connector mapping, such that the transition relation in each automaton $\mathcal{G}(\gamma)$ is confluent, σ and σ' be choreography states, and $\mathcal{A}$ and $\mathcal{A}'$ be automata state functions. If $\vdash^{\mathcal{G}}_{\mathcal{A}} C$ and $C, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C', \sigma', \mathcal{A}'$, then $\vdash^{\mathcal{G}}_{\mathcal{A}'} C'$.*

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

▶ **Corollary 25** (Soundness of Compatibility). *Under the assumptions of Theorem 24, if $\vdash^{\mathcal{G}}_{\mathcal{A}} C$, then $\mathcal{G}$ in $\mathcal{A}$ respects C.*

Furthermore, compatibility is decidable.

▶ **Theorem 26** (Decidability of Compatibility). *There is an algorithm that, given C, $\mathcal{G}$ and $\mathcal{A}$, returns* YES *if $\vdash^{\mathcal{G}}_{\mathcal{A}} C$ and* NO *if $\nvdash^{\mathcal{G}}_{\mathcal{A}} C$.*

Proof. 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 $\vdash^{\mathcal{G}}_{\mathcal{A}} C$ or not. ◀

In the Appendix, we give a more intelligent proof that constructs the types for the recursive definitions.

▶ **Theorem 27** (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 $\vdash^{\mathcal{G}}_{\mathcal{A}} C$. 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 $\tilde{\eta}\,\texttt{thru}\,\gamma; C'$ or $\texttt{if}\,\mathsf{p}.e\,\texttt{then}\,C_1\,\texttt{else}\,C_2$, eventually inside some recursive definitions. In the latter case, C can always reduce; in the former case, compatibility guarantees that C can reduce. ◀

Given the previous theorem, if $C \not\preceq \mathbf{0}$, then by Theorem 24 we also have that $\vdash^{\mathcal{G}}_{\mathcal{A}'} C'$ whenever $C, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C', \sigma', \mathcal{A}'$. By induction, we therefore can state a stronger result: every choreography starting from a compatible automaton state function always keeps reducing unless it reaches $\mathbf{0}$. Thus, choreographies starting from an automaton state function they are compatible with can never deadlock.

▶ **Theorem 28** (Deadlock-Freedom by Design). *Let C be a choreography, σ be a choreography state function, and $\mathcal{A}$ be an automaton state function. If $\vdash^{\mathcal{G}}_{\mathcal{A}} C$ 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 $C', \sigma', \mathcal{A}' \rightsquigarrow_{\mathcal{G}} C'', \sigma'', \mathcal{A}''$.*

5 Connected Processes

The results presented so far show that choreographies can be combined with connectors, but do not indicate yet how we can use CR to obtain executable implementations of concurrent systems. The missing link is determining how, given a choreography, we can synthesise terms that represents executable concurrent processes that communicate through connectors. We address this aspect in this section, first by defining a process calculus based on standard I/O actions and then by presenting a translation (compilation procedure) from CR to this calculus.

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

Figure 8 Connected Processes, Syntax.

5.1 Syntax and semantics

We define Connected Processes (CP), the process calculus that we use to represent concrete implementations of choreographies. The syntax of CP is given in Figure 8. A network N is a parallel composition of processes. A process is written $\mathtt{p} \triangleright_\rho B$, where $\mathtt{p}$ is its identifier, ρ its current 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 choreographies. Communication actions implement the local behaviour of each process in a choreography interaction: sending a value through an output port ($\mathtt{o}!\langle e\rangle$); receiving a value through an input port ($\mathtt{i}?x$); selecting a label through an output port ($\mathtt{o} \oplus \ell$); and offering a choice on some labels through an input port ($\mathtt{i}\&\{\ell_i : B_i\}_{i\in I}$).

The key difference with respect to choreographies is that communications now refer to actual ports, instead of to connectors (i.e., we have no "$\mathtt{thru}\,\gamma$" 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 choreographies, the automata 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 (the remaining ones are standard; see the Appendix):

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

This rule is reminiscent of rule $\lfloor \mathrm{C}|\mathrm{Com}\rceil$ 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 main rules are given in Figure 9 (see the Appendix for the others).

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 (Figure 10), notably in the rule for projecting communications.

▶ Remark. In Figure 10, $\mathtt{o}$ and $\mathtt{i}$ denote variables that range over concrete ports. Thus, a process $\mathtt{p}$ has output ports $\mathtt{o}_{\gamma_1}, \mathtt{o}_{\gamma_2}, \ldots$, and input ports $\mathtt{i}_{\gamma_1}, \mathtt{i}_{\gamma_2}, \ldots$, where $\mathtt{o}$ and $\mathtt{i}$ actually stand for $\mathtt{o}^{\mathtt{p}}$ ("output port at $\mathtt{p}$") and $\mathtt{i}^{\mathtt{p}}$ ("input port at $\mathtt{p}$"), while connector γ_1 knows output ports $\mathtt{o}^{\mathtt{p}_1}_{\gamma_1}, \mathtt{o}^{\mathtt{p}_2}_{\gamma_1}, \ldots$, and similarly for input ports. ◀

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).

$$\frac{e \downarrow^{\rho_\mathtt{p}} v}{\mathtt{p} \triangleright_{\rho_\mathtt{p}} \mathtt{o}!\langle e\rangle; B_\mathtt{p} \mid \mathtt{q} \triangleright_{\rho_\mathtt{q}} \mathtt{i}?x; B_\mathtt{q}, \mu \xrightarrow{\mathtt{p.o}\to\mathtt{q.i}} \mathtt{p} \triangleright_{\rho_\mathtt{p}} B_\mathtt{p} \mid \mathtt{q} \triangleright_{\rho_\mathtt{q}[x\mapsto v]} B_\mathtt{q}, \mu} \lfloor\text{CP}|\text{SyncVal}\rceil$$

$$\frac{e \downarrow^{\rho} v}{\mathtt{p} \triangleright_{\rho} \mathtt{o}!\langle e\rangle; B, \mu \xrightarrow{\mathtt{p.o}\to m} \mathtt{p} \triangleright_{\rho} B, \mu[m \mapsto v]} \lfloor\text{CP}|\text{SendVal}\rceil$$

$$\frac{\mu(m) = v}{\mathtt{q} \triangleright_{\rho} \mathtt{i}?x; B, \mu \xrightarrow{m\to\mathtt{q.i}} \mathtt{q} \triangleright_{\rho[x\mapsto v]} B, \mu} \lfloor\text{CP}|\text{RecvVal}\rceil \qquad \frac{\mu(m_1) = v}{\mathbf{0}, \mu \xrightarrow{m_1\to m_2} \mathbf{0}, \mu[m_2 \mapsto v]} \lfloor\text{CP}|\text{Mem}\rceil$$

$$\frac{N_1, \mu \xrightarrow{\phi_1} N_1', \mu' \quad N_2, \mu' \xrightarrow{\phi_2} N_2', \mu'' \quad (\dagger)}{(N_1 \mid N_2), \mu \xrightarrow{\phi_1\cup\phi_2} (N_1' \mid N_2'), \sigma'', \mu''} \lfloor\text{CP}|\text{Join}\rceil$$

Figure 9 Semantics of communications (process level, main rules). Side condition (†) in $\lfloor\text{CP}|\text{Join}\rceil$ is the same as in $\lfloor\text{C}|\text{Join}\rceil$.

$$[\![\tilde\eta\ \mathtt{thru}\ \gamma; C]\!]_\mathtt{r} = \begin{cases} \mathtt{o}_\gamma!\langle e\rangle; [\![C]\!]_\mathtt{r} & \text{if } \mathtt{r} = \mathtt{p} \text{ and } \mathtt{p}\langle e\rangle \texttt{ -> } \mathtt{q}.x \in \tilde\eta \\ \mathtt{i}_\gamma?x; [\![C]\!]_\mathtt{r} & \text{if } \mathtt{r} = \mathtt{q} \text{ and } (\mathtt{p}\langle e\rangle \texttt{ -> } \mathtt{q}.x \in \tilde\eta \text{ or } \mathtt{q}.x?v \in \tilde\eta) \\ \mathtt{o}_\gamma \oplus \ell; [\![C]\!]_\mathtt{r} & \text{if } \mathtt{r} = \mathtt{p} \text{ and } \mathtt{p} \texttt{ -> } \mathtt{q}[\ell] \in \tilde\eta \\ \mathtt{i}_\gamma \& \{\ell : [\![C]\!]_\mathtt{r}\} & \text{if } \mathtt{r} = \mathtt{q} \text{ and } (\mathtt{p} \texttt{ -> } \mathtt{q}[\ell] \in \tilde\eta \text{ or } \mathtt{q}[\ell] \in \tilde\eta) \end{cases}$$

$$[\![\mathtt{if}\ \mathtt{p}.e\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2]\!]_\mathtt{r} = \begin{cases} \mathtt{if}\ e\ \mathtt{then}\ [\![C_1]\!]_\mathtt{r}\ \mathtt{else}\ [\![C_2]\!]_\mathtt{r} & \text{if } \mathtt{r} = \mathtt{p} \\ [\![C_1]\!]_\mathtt{r} \sqcup [\![C_2]\!]_\mathtt{r} & \mathtt{r} \neq \mathtt{p} \end{cases}$$

$$[\![\mathtt{def}\ X = C_2\ \mathtt{in}\ C_1]\!]_\mathtt{r} = \mathtt{def}\ X = [\![C_2]\!]_\mathtt{r}\ \mathtt{in}\ [\![C_1]\!]_\mathtt{r} \qquad [\![\mathbf{0}]\!]_\mathtt{r} = \mathbf{0} \qquad [\![X]\!]_\mathtt{r} = X$$

Figure 10 Cho-Reo-graphies, Behaviour Projection.

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.

$$[\![C, \sigma]\!] = \prod_{\mathtt{p}\in\mathsf{pn}(C)} \mathtt{p} \triangleright_{\rho_\mathtt{p}} [\![C]\!]_\mathtt{p} \qquad \text{where } \rho_\mathtt{p}(x) = \sigma(\mathtt{p}, x) \text{ for each variable } x \text{ at } \mathtt{p}$$

We say that C is projectable when $[\![C, \sigma]\!]$ is defined for some σ. This is equivalent to saying that $[\![C, \sigma]\!]$ is defined for all σ.

▶ **Example 29** (Book sale)**.** Continuing with our running example, the choreography presented in Example 4 is projectable, and yields the following network of connected processes each state σ.

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

◀

▶ **Example 30.** It is also worthwhile to note that the following choreographies are *not* congruent, and that they have *different* EPPs:

$$C_1 = \mathtt{p} \ \texttt{->}\ \{\mathtt{q},\mathtt{r}\}[\ell]\ \mathtt{thru}\ \gamma;\ \mathbf{0}$$
$$C_2 = \mathtt{p} \ \texttt{->}\ \mathtt{q}[\ell]\ \mathtt{thru}\ \gamma;\ \mathtt{p}\ \texttt{->}\ \mathtt{r}[\ell]\ \mathtt{thru}\ \gamma;\ \mathbf{0}$$

Choreography C_1 is syntactic sugar for $\{\mathtt{p}\ \texttt{->}\ \mathtt{q}[\ell], \mathtt{p}\ \texttt{->}\ \mathtt{r}[\ell]\}\ \mathtt{thru}\ \gamma; \mathbf{0}$. The EPP to p, thus, consists of *one* send; connector γ must subsequently ensure that label ℓ is replicated to, and received by, both q and r (i.e., formally, γ must be compatible with C_1).[6]

Choreography C_2, in contrast, is not congruent to $\{\mathtt{p}\ \texttt{->}\ \mathtt{q}[\ell], \mathtt{p}\ \texttt{->}\ \mathtt{r}[\ell]\}\ \mathtt{thru}\ \gamma; \mathbf{0}$. Specifically, we cannot use rule ⌊C|Eta-Split⌉ to merge the two interactions in C_2, because its disjointness premise does not hold (p occurs in both interactions). Accordingly, the EPP of choreography C_2 on 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 $[\![\mathcal{G}]\!]$ to be the set of all automata in the codomain of $\mathcal{G}$, where each output port p in automaton $\mathcal{G}(\gamma)$ becomes $\mathtt{p.o}_\gamma$ (and likewise for input ports). We define a similar function for automaton state function $\mathcal{A}$.

▶ **Theorem 31** (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 $[\![C, \sigma]\!], [\![\mathcal{A}]\!] \rightsquigarrow_{[\![\mathcal{G}]\!]} [\![C', \sigma']\!], [\![\mathcal{A}']\!]$;*
Soundness: *If $[\![C, \sigma]\!], [\![\mathcal{A}]\!] \rightsquigarrow_{[\![\mathcal{G}]\!]} N, A'$, then $C, \sigma, \mathcal{A} \rightsquigarrow_{\mathcal{G}} C', \sigma', \mathcal{A}'$ for some σ' and $\mathcal{A}'$ with $[\![C', \sigma']\!] \prec N$ and $[\![\mathcal{A}']\!] = A'$.*

In the soundness result, the *pruning relation* $\prec$ [15, 17] states that the processes in the network N' may offer more branches than those present in $[\![C', \sigma']\!]$. Such branches can never be selected [15, 32, 25].

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

▶ **Example 32** (Book sale). For any connector mapping $\mathcal{G}$, the process network in Example 29 operates under $[\![\mathcal{G}]\!]$ exactly as the choreography in Example 4 under $\mathcal{G}$. In particular, if $\mathcal{G}$ respects the original choreography, then this implementation never deadlocks under $[\![\mathcal{G}]\!]$. ◀

6 Conclusions

Choreographic approaches to concurrent programming have been heavily investigated [1, 28], but these approaches typically adopt some fixed (and restrictive) definition (like point-to-point synchronous) for the semantics of communications. CR is the first model that allows choreographies to be modularly integrated with what runs "under the hood" of communications, allowing for user-defined communication semantics specified as connectors. Thanks to the notion of compatibility (Definition 20), CR inherits the good properties of both Choreographic Programming and Exogenous Coordination. Thus, we have significantly extended the applicability of choreographies. Not only can we capture new kinds of behaviours in choreographies (like barriers, cf. Example 6, and alternators, cf. Example 12), but

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

we can even use choreographies to describe 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, 39, 40]. 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 5), 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 $\mathtt{p}\langle e\rangle \texttt{ -> } \mathtt{q}.x\,\mathtt{thru}\,\gamma$ can allow γ to send the message from $\mathtt{p}$ to $\mathtt{q}$ via an intermediate process $\mathtt{r}$ 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. Deniélou, 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. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies: Theory and implementation. *LMCS*, 13(2), 2017.

26 P.-M. Deniélou 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. Deniélou, 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 K.-K. Lau, P. V. Elizondo, and Z. Wang. Exogenous connectors for software components. In *CBSE*, pages 90–106. Springer, 2005.

34 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.

35 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.

36 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.

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

38 F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.

39 J. Proença. *Synchronous coordination of distributed components.* PhD thesis, Leiden University, 2011.

40 J. Proença, D. Clarke, E. De Vink, and F. Arbab. Dreams: a framework for distributed synchronous coordination. In *SAC*, pages 1510–1515. ACM, 2012.

41 C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014.

Additional Proofs

Theorem 19. Let η be a communication action that does not respect $\mathcal{A}$, and assume that connector γ has synchronous links `p2q` and `q2p`, from `p` to `q` and conversely (e.g., *Sync* in Figure 1). Synchronous links are always enabled and do not change $\mathcal{A}$.

Let f be a total function implemented at `p` and consider the choreography

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

In this choreography, `q` sequentially sends the natural numbers to `p`, which applies f to its input and proceeds if the result is 0. If `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. ◀

Theorem 24. Straightforward by case analysis on the reduction from $C, \sigma, \mathcal{A}$ 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. ◀

Corollary 25. 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 24 to show that $\eta; C', \sigma', \mathcal{A}' \rightsquigarrow_{\mathcal{G}} C', \sigma'', \mathcal{A}''$ for some $\mathcal{A}''$ and σ''. ◀

Theorem 26, alternative proof. 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 $\vdash^{\mathcal{G}}_{\mathcal{A}} C$ by applying the rules in Figure 7 bottom-up. When we meet a term of the form $\texttt{def}\ X = C_2\ \texttt{in}\, C_1$, we focus on C_1 first, and leave $\mathcal{A}_X$ (see rule $\lfloor\text{CC}|\text{Def}\rceil$) 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 $\mathcal{L} = [\cdot \vdash^{\mathcal{G}}_{\mathcal{A}} C]$.
2. While $\mathcal{L}$ is not empty:
   - **a.** Remove the first pending judgement $\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} C$ from $\mathcal{L}$.
   - **b.** If C is $\mathbf{0}$, proceed to the next iteration.
   - **c.** If C is of the form $\texttt{if}\,\texttt{p}.e\,\texttt{then}\, C_1\,\texttt{else}\, C_2$, then add $\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} C_1$ and $\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}} C_2$ at the beginning of $\mathcal{L}$.
   - **d.** If C is of the form $\texttt{def}\ X = C_2\ \texttt{in}\, C_1$, then add $\Gamma, (X : \mathcal{A}_X) \vdash^{\mathcal{G}}_{\mathcal{A}} C_1$ and $\Gamma, (X : \mathcal{A}_X) \vdash^{\mathcal{G}}_{\mathcal{A}} C_2$, *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.[7]
   - **f.** Otherwise, C is of the form $\tilde{\eta}\,\texttt{thru}\,\gamma; C'$. Consider all possible ways of rewriting C as $\tilde{\eta}'\,\texttt{thru}\,\gamma; C'$ 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

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

$$\frac{e \downarrow^{\rho_{\mathtt{p}}} v}{\mathtt{p} \triangleright_{\rho_{\mathtt{p}}} \mathtt{o}!\langle e\rangle; B_{\mathtt{p}} \mid \mathtt{q} \triangleright_{\rho_{\mathtt{q}}} \mathtt{i}?x; B_{\mathtt{q}}, \mu \xrightarrow{\mathtt{p.o} \to \mathtt{q.i}} \mathtt{p} \triangleright_{\rho_{\mathtt{p}}} B_{\mathtt{p}} \mid \mathtt{q} \triangleright_{\rho_{\mathtt{q}}[x \mapsto v]} B_{\mathtt{q}}, \mu} \ \lfloor \text{CP}|\text{SyncVal} \rceil$$

$$\frac{e \downarrow^{\rho} v}{\mathtt{p} \triangleright_{\rho} \mathtt{o}!\langle e\rangle; B, \mu \xrightarrow{\mathtt{p.o} \to m} \mathtt{p} \triangleright_{\rho} B, \mu[m \mapsto v]} \ \lfloor \text{CP}|\text{SendVal} \rceil$$

$$\frac{\mu(m) = v}{\mathtt{q} \triangleright_{\rho} \mathtt{i}?x; B, \mu \xrightarrow{m \to \mathtt{q.i}} \mathtt{q} \triangleright_{\rho[x \mapsto v]} B, \mu} \ \lfloor \text{CP}|\text{RecvVal} \rceil$$

$$\frac{j \in I}{\mathtt{p} \triangleright_{\rho_{\mathtt{p}}} \mathtt{o} \oplus \ell_j; B \mid \mathtt{q} \triangleright_{\rho_{\mathtt{q}}} \mathtt{i}\&\{\ell_i : B_i\}_{i \in I}, \mu \xrightarrow{\mathtt{p.o} \to \mathtt{q.i}} \mathtt{p} \triangleright_{\rho_{\mathtt{p}}} B \mid \mathtt{q} \triangleright_{\rho_{\mathtt{q}}} B_j, \mu} \ \lfloor \text{CP}|\text{SyncSel} \rceil$$

$$\frac{}{\mathtt{p} \triangleright_{\rho} \mathtt{o} \oplus \ell; B, \mu \xrightarrow{\mathtt{p.o} \to m} \mathtt{p} \triangleright_{\rho} B, \mu} \ \lfloor \text{CP}|\text{SendSel} \rceil$$

$$\frac{\mu(m) = \ell_j \qquad j \in I}{\mathtt{q} \triangleright_{\rho} \mathtt{i}\&\{\ell_i : B_i\}_{i \in I}, \mu \xrightarrow{m \to \mathtt{q.i}} \mathtt{q} \triangleright_{\rho} B_j, \mu} \ \lfloor \text{CP}|\text{RecvSel} \rceil$$

$$\frac{N_1, \mu \xrightarrow{\phi_1} N_1', \mu' \qquad N_2, \mu' \xrightarrow{\phi_2} N_2', \mu''}{(N_1 \mid N_2), \mu \xrightarrow{\phi_1 \cup \phi_2} (N_1' \mid N_2'), \sigma'', \mu''} \ \lfloor \text{CP}|\text{Join} \rceil$$

Figure 11 Semantics of communications (process level).

the affirmative case compute s' such that $s \xrightarrow{\phi}_{\gamma} s'$ and add $\Gamma \vdash^{\mathcal{G}}_{\mathcal{A}[\gamma \to \langle s', \mu' \rangle]} \tilde{\eta}'' \, \mathtt{thru} \, \gamma; C'$ at the beginning of $\mathcal{L}$. If no such transitions exist, return NO.

3. Return YES.

Termination of this algorithm is straight-forward: the sum of the sizes of all the choreographies in $\mathcal{L}$ stricly 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 $\mathtt{p}\langle e\rangle$ `->` $\mathtt{q}.x$ and $\mathtt{p}$ `->` $\mathtt{q}[\ell]$ count as 2, while $\mathtt{q}.x?v$ and $\mathtt{q}[\ell]$ 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 $\vdash^{\mathcal{G}}_{\mathcal{A}} C$, 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 $\vdash^{\mathcal{G}}_{\mathcal{A}} C$ 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 $\nvdash^{\mathcal{G}}_{\mathcal{A}} C$. 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 $\nvdash^{\mathcal{G}}_{\mathcal{A}} C$. ◀

Complete semantics of Connected Processes

Figure 11 gives the full definition of the auxiliary reduction from rule $\lfloor$CP|Com$\rceil$. Figure 12 gives the semantics of CP. Rule $\lfloor$CP|Struct$\rceil$ uses the structural precongruence relation $\preceq$ (including associativity and commutativity of $|$), defined in the standard way [22].

$$\frac{i = 1 \text{ if } e \downarrow^{\rho} \mathsf{true}, \quad i = 2 \text{ otherwise}}{\mathtt{p} \triangleright_{\rho} \mathtt{if}\, e\, \mathtt{then}\, B_1\, \mathtt{else}\, B_2, A \rightsquigarrow_{\mathcal{C}} \mathtt{p} \triangleright_{\rho} B_i, A} \lfloor \mathrm{CP|Cond} \rceil$$

$$\frac{\mathtt{p} \triangleright_{v} B_1 \mid N, A \rightsquigarrow_{\mathcal{C}} \mathtt{p} \triangleright_{v} B_1' \mid N', A'}{\mathtt{p} \triangleright_{v} \mathtt{def}\, X = B_2\, \mathtt{in}\, B_1 \mid N, A \rightsquigarrow_{\mathcal{C}} \mathtt{p} \triangleright_{v} \mathtt{def}\, X = B_2\, \mathtt{in}\, B_1' \mid N', A'} \lfloor \mathrm{CP|Ctx} \rceil$$

$$\frac{N, A \rightsquigarrow_{\mathcal{C}} N', A'}{N \mid M, A \rightsquigarrow_{\mathcal{C}} N' \mid M, A'} \lfloor \mathrm{CP|Par} \rceil \qquad \frac{N \preceq M \quad M, A \rightsquigarrow_{\mathcal{C}} M', A' \quad M' \preceq N'}{N, A \rightsquigarrow_{\mathcal{C}} N', A'} \lfloor \mathrm{CP|Struct} \rceil$$

Figure 12 Connected Processes, Semantics.