Causally Consistent Reversible Choreographies: A Monitors-as-Memories Approach

Claudio Antares Mezzina
IMT School for Advanced Studies Lucca
Lucca, Italy
claudio.mezzina@imtlucca.it

Jorge A. Pérez[*]
University of Groningen
Groningen, The Netherlands
j.a.perez@rug.nl

ABSTRACT

Under a reversible semantics, computation steps can be undone. This paper addresses the integration of reversible semantics into a process model of multiparty protocols (*choreographies*). Building upon the *monitors-as-memories* approach that we developed in prior work for reversible binary protocols, we present a reversible process framework for *multiparty* communication, which improves on prior models by seamlessly integrating *asynchrony*, *decoupled rollbacks*, and *process passing*. As main technical result, we prove that our multiparty, reversible semantics is *causally-consistent*.

CCS CONCEPTS

• **Theory of computation** → **Process calculi**; **Type structures**; **Operational semantics**; **Program analysis**; • **Software and its engineering** → *Distributed programming languages*; *Concurrent programming languages*;

KEYWORDS

Concurrency, process calculi, reversible semantics, multiparty session types, causal consistency

ACM Reference Format:
Claudio Antares Mezzina and Jorge A. Pérez. 2017. Causally Consistent Reversible Choreographies: A Monitors-as-Memories Approach. In *Proceedings of PPDP'17, Namur, Belgium, October 9–11, 2017,* 12 pages.
https://doi.org/10.1145/3131851.3131864

1 INTRODUCTION

This paper is about *reversible computation* in the context of models of concurrency for *communication-centric* software systems, i.e., collections of distributed software components whose concurrent interactions are governed by reciprocal dialogues or *protocols*.

Building upon process calculi techniques, these models provide a rigorous footing for message-passing concurrency; on top of them, many (static) analysis techniques based on *(behavioral) types* and *contracts* have been put forward to enforce key safety and liveness properties [14]. Reversibility is an appealing notion in concurrency at large [19], but especially so in communication-centric scenarios: it may elegantly abstract fault-tolerant communicating systems that react to unforeseen circumstances (say, local failures) by "undoing" computation steps so as to reach a consistent previous state.

In communication-centric software systems, protocols specify the intended communication structures among interacting components. We focus on process calculi equipped with behavioral types, which use those protocols as types to enforce communication correctness. The interest is in different flavors of *protocol conformance*, i.e., properties that ensure that each component respects its ascribed protocol. The integration of reversibility in models of communication-centric systems has been addressed from various angles (cf. [2, 20, 26, 27]). Focusing on *session types* [12, 13] (a well established class of behavioral types), Tiezzi and Yoshida [26] were the first to integrate reversibility into a session π-calculus, following the seminal approach of Danos and Krivine [6]; in [26], however, session types are not used in the definition of reversible communicating systems, nor play a role in establishing their properties.

Triggered by this observation, our prior work [20, 22] develops a *monitors-as-memories* approach. The idea is to use *monitors* (runtime entities that enact protocol actions) as the *memories* needed to record and undo communication steps. There is a monitor for each protocol participant; the monitor includes a session type that describes the intended protocol. We use a *cursor* to "mark" the current protocol state in the type; the cursor can move to the future (enacting protocol actions) but also to the past (reversing protocol actions). The result is a streamlined process framework in which the key properties of a reversible semantics can be established with simple proofs, because session types narrow down the spectrum of possible process behaviors, allowing only those forward and backward actions that correspond to the declared protocols. The most significant of such properties is *causal consistency* [6], considered as the "right" criterion for reversing concurrent processes [19]. Intuitively, causal consistency ensures that reversible steps lead to system states that could been have reached by performing forward steps only. That is, causally consistent reversibility does not lead to extraneous states, not reachable through ordinary computations.

The framework in [20, 22], however, accounts only for reversible π-calculus processes implementing *binary sessions*, i.e., protocols between exactly two partners. Also, it considers *synchronous communication* instead of the more general (and practical) *asynchronous (queue-based) communication.* Hence, our prior work rules out an important class of real-life protocols, namely the *choreographies* that describe interaction scenarios among multiple parties without a single point of control. In *multiparty session types* [13], these

[*]Also with CWI, Amsterdam.

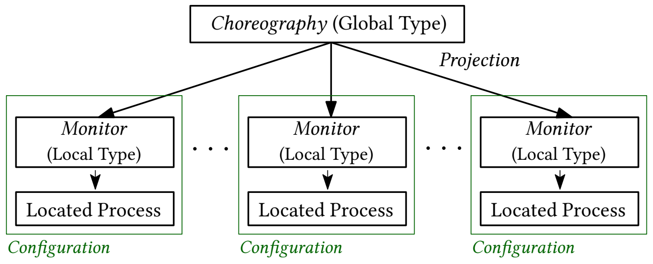

Figure 1: Our process model of multiparty communications.

choreographies are represented by a *global type* that can be projected as *local types* to obtain each participant's contribution to the entire interaction. Moving from binary to multiparty sessions is a significant jump in expressiveness; in fact, global types offer a convenient declarative description of the entire communication scenario. However, the multiparty case also entails added challenges, as two levels of abstraction, global and local, should be considered for (reversible) protocols and their implementations. Hence, it is far from obvious that our monitors-as-memories approach to reversibility and causal consistency extend to the multiparty case.

This paper makes the following contributions:

1. We introduce a process model for reversible, multiparty sessions with asynchrony (as in [16]), process passing [15, 24] and decoupled rollbacks (§ 2). We define forward and backward semantics for multiparty processes by extending the monitors-as-memories approach to both global types and their implementations.
2. We prove that reversibility in our model is causally consistent (Theorem 4.18). The proof is challenging as we must appeal to an alternative reversible semantics with *atomic rollbacks*, which we show to coincide with the decoupled rollbacks (Theorem 4.10).
3. We formally connect reversibility at the (declarative) level of global types and that at the (operational) level of processes monitored by local types with cursors (Theorem 4.22).

We stress that asynchrony, process passing, and decoupled rollbacks are not considered in prior works [8, 20, 22, 27]. Asynchrony and decoupled rollbacks are delicate issues in a reversible multiparty setting—we do not know of other asynchronous calculi with reversible semantics, nor featuring the same combination of constructs. The formal connection between global and local levels of abstraction (Theorem 4.22) is also unique to our multiparty setting.

Organization. This paper is organized as follows. In § 2, we introduce our process model of reversible choreographies. We illustrate the model by means of an example in § 3. In § 4 we establish causal consistency by relating decoupled and atomic semantics, and connect reversibility at global and local levels. § 5 discusses extensions to our framework, while § 6 contrasts with related works. § 7 collects some concluding remarks and directions for future work. Omitted definitions/proofs can be found in an online technical report [21].

2 REVERSIBLE CHOREOGRAPHIES

Fig. 1 depicts the ingredients of our two-level model of *choreographies* and *configurations/processes*. Choreographies are defined in terms of *global types*, which declaratively describe a protocol among two or more participants. A global type can be *projected* onto each participant so as to obtain its corresponding *local type*, i.e., a session type that abstracts a participant's contribution to the global protocol. (Below we often use 'choreographies' and 'global types' as synonyms.) The semantics of global types is given in terms of forward and backward transition systems (Fig. 3). There is a *configuration* for each protocol participant: it includes a *located process* that specifies asynchronous communication behavior, subject to a *monitor* that enables forward/backward steps at run-time based on the local type. The semantics of configurations is given in terms of forward and backward reduction relations (Figs. 6 and 7).

Remark 1 (Colors). *Throughout the paper, we use colors to improve readability. In particular, elements in blue belong to a forward semantics; elements in red belong to a backward semantics.*

2.1 Global and Local Types

2.1.1 Syntax. Let us write $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{A}, \mathsf{B} \ldots$ to denote (protocol) *participants*. The syntax of global types $(G, G', \ldots)$ and local types $(T, T', \ldots)$ is standard [13] and defined as follows:

$$\begin{aligned} G, G' &::= \mathsf{p} \to \mathsf{q} : \langle U \rangle.G \mid \mu X.G \mid X \mid \mathsf{end} \\ U, U' &::= \mathsf{bool} \mid \mathsf{nat} \mid \cdots \mid T \to \diamond \\ T, T' &::= \mathsf{p}!\langle U \rangle.T \mid \mathsf{p}?\langle U \rangle.T \mid \mu X.T \mid X \mid \mathsf{end} \end{aligned}$$

Global type $\mathsf{p} \to \mathsf{q} : \langle U \rangle.G$ says that p may send a value of type U to q, and then continue as G. we assume that $\mathsf{p} \neq \mathsf{q}$. For the sake of presentation, here we do not consider labeled choices in global types; these can be easily accommodated—see § 5 and [21]. Global recursive and terminated protocols are denoted $\mu X.G$ and end, respectively. We write $\mathsf{pa}(G)$ to denote the set of participants in G. Value types U include basic first-order values (constants), but also *higher-order* values: abstractions from names to processes. (We write $\diamond$ to denote the type of processes.) Local types $\mathsf{p}!\langle U \rangle.T$ and $\mathsf{p}?\langle U \rangle.T$ denote, respectively, an output and input of value of type U by p. We use α to denote type prefixes $\mathsf{p}?(U)$, $\mathsf{p}!\langle U \rangle$. Terminated and recursive local types are denoted end and $\mu X.T$, respectively.

As usual, we consider only recursive types $\mu X.G$ (and $\mu X.T$) in which X occurs guarded in G (and T). We shall take an equi-recursive view of (global and local) types, and so we consider two types with the same regular tree as equal.

Global and local types are related by *projection*: following [13], the projection of global type G onto participant r, written $G{\downarrow}_{\mathsf{r}}$, is defined in Fig. 2. Intuitively, $G{\downarrow}_{\mathsf{r}}$ denotes the (local) contribution of r to the overall choreographic behavior that G declaratively specifies. As such, the projection of a directed communication $\mathsf{p} \to \mathsf{q} : \langle U \rangle.G'$ onto r depends on r's involvement, which is reflected locally as an input or output local type (if r is indeed involved) or as the terminated local protocol (otherwise). The projection of recursive choreographies follows a similar principle; recursive variables and the terminated choreography are projected as expected.

2.1.2 Semantics of Choreographies. The semantics of global types (Fig. 3) comprises forward and backward transition rules. To express backward steps, we require some auxiliary notions. We use *global contexts*, ranged over by $\mathbb{G}, \mathbb{G}', \ldots$ with holes $\bullet$, to record previous actions:

$$\mathbb{G} \;::=\; \bullet \mid \mathbb{G}[\mathsf{p} \to \mathsf{q} : \langle U \rangle.\mathbb{G}]$$

$$(\mathsf{p} \to \mathsf{q} : \langle U \rangle.G)\downarrow_{\mathsf{r}} = \begin{cases} \mathsf{q}!\langle U \rangle.(G\downarrow_{\mathsf{r}}) & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}?\langle U \rangle.(G\downarrow_{\mathsf{r}}) & \text{if } \mathsf{r} = \mathsf{q} \\ (G\downarrow_{\mathsf{r}}) & \text{if } \mathsf{r} \neq \mathsf{q}, \mathsf{r} \neq \mathsf{p} \end{cases}$$

$$(\mu X.G)\downarrow_{\mathsf{r}} = \begin{cases} \mu X.G\downarrow_{\mathsf{r}} & \text{if } \mathsf{r} \text{ occurs in } G \\ \mathsf{end} & \text{otherwise} \end{cases}$$

$$X\downarrow_{\mathsf{r}} = X \qquad \mathsf{end}\downarrow_{\mathsf{r}} = \mathsf{end}$$

Figure 2: Projection of a global type G onto a participant r.

$$\text{(FVal1)}\quad \mathbb{G}[\,{\wedge\!\!\wedge}\mathsf{p} \to \mathsf{q} : \langle U \rangle.G] \hookrightarrow \mathbb{G}[\mathsf{p} \to {\wedge\!\!\wedge}\mathsf{q} : \langle U \rangle.G]$$

$$\text{(FVal2)}\quad \mathbb{G}[\mathsf{p} \to {\wedge\!\!\wedge}\mathsf{q} : \langle U \rangle.G] \hookrightarrow \mathbb{G}[\mathsf{p} \to \mathsf{q} : \langle U \rangle.\,{\wedge\!\!\wedge}G]$$

$$\text{(BVal1)}\quad \mathbb{G}[\mathsf{p} \to {\wedge\!\!\wedge}\mathsf{q} : \langle U \rangle.G] \rightharpoonup \mathbb{G}[\,{\wedge\!\!\wedge}\mathsf{p} \to \mathsf{q} : \langle U \rangle.G]$$

$$\text{(BVal2)}\quad \mathbb{G}[\mathsf{p} \to \mathsf{q} : \langle U \rangle.\,{\wedge\!\!\wedge}G] \rightharpoonup \mathbb{G}[\mathsf{p} \to {\wedge\!\!\wedge}\mathsf{q} : \langle U \rangle.G]$$

Figure 3: Semantics of Global Types (Forward & Backwards).

We also use *global types with history*, ranged over by $\mathsf{H}, \mathsf{H}', \ldots$, to record the current protocol state. This state is denoted by the *cursor* ${\wedge\!\!\wedge}$, which we introduced in [20]:

$$\mathsf{H}, \mathsf{H}' \quad ::= \quad {\wedge\!\!\wedge}G \mid G\,{\wedge\!\!\wedge} \mid \mathsf{p} \to {\wedge\!\!\wedge}\mathsf{q} : \langle U \rangle.G \mid \mathsf{p} \to \mathsf{q} : \langle U \rangle.\,{\wedge\!\!\wedge}G$$

Intuitively, directed exchanges such as $\mathsf{p} \to \mathsf{q} : \langle U \rangle.G$ have three *intermediate states*, characterized by the decoupled involvement of p and q in the intended asynchronous model. The *first state*, denoted ${\wedge\!\!\wedge}\mathsf{p} \to \mathsf{q} : \langle U \rangle.G$, describes the situation prior to the exchange. The *second state* represents the point in which p has sent a value of type U but this message has not yet reached q; this is denoted $\mathsf{p} \to {\wedge\!\!\wedge}\mathsf{q} : \langle U \rangle.G$. The *third state* represents the point in which q has received the message from p and the continuation G is ready to execute; this is denoted by $\mathsf{p} \to \mathsf{q} : \langle U \rangle.\,{\wedge\!\!\wedge}G$. Using these intermediate states is central to precisely characterize the intended asynchronous semantics for processes.

These intuitions come in handy to describe the forward and backward transition rules in Fig. 3. For a forward directed exchange of a value, Rule (FVal1) formalizes the transition from the first to the second state; Rule (FVal2) denotes the transition from the second to the third state. Rules (BVal1) and (BVal2) undo the step performed by Rules (FVal1) and (FVal2), respectively.

2.2 Processes and Configurations

2.2.1 Syntax. The syntax of processes and configurations is given in Fig. 4. For processes $P, Q, \ldots$ we follow closely the syntax of HOπ, the core higher-order session π-calculus [15]. The syntax of configurations builds upon that of processes.

Names a, b, c (resp. s, s') range over shared (resp. session) names. We use session names indexed by participants, denoted $s_{[\mathsf{p}]}, s_{[\mathsf{q}]}$. Names n, m are session or shared names. First-order values v, v' include base values and constants. Variables are denoted by x, y, and recursive variables are denoted by X, Y. The syntax of values V includes shared names, first-order values, but also name abstractions

$$\begin{array}{rcl} u, w & ::= & n \mid x, y, z \qquad\qquad n, n' \;\; ::= \;\; a, b \mid s_{[\mathsf{p}]} \\ v, v' & ::= & \mathtt{tt} \mid \mathtt{ff} \mid \cdots \\ V, W & ::= & a, b \mid x, y, z \mid v, v' \mid \lambda x.\,P \\ P, Q & ::= & u!\langle V \rangle.P \mid u?(x).P \mid P \mid Q \mid X \mid \mu X.P \mid V\,u \mid (\nu\, n)P \mid \mathbf{0} \\ M, N & ::= & \ell\,\{a!\langle x \rangle.P\} \mid \ell\,\{a?(x).P\} \mid M \mid N \mid (\nu\, n)M \mid \mathbf{0} \\ & & \mid \boxed{\ell_{[\mathsf{p}]} : \wr P \int} \mid \boxed{s_{[\mathsf{p}]} \lfloor H \cdot \widetilde{x} \cdot \sigma \rfloor^{\spadesuit}} \\ & & \mid \boxed{s : (h_i \star h_o)} \mid \boxed{k \lfloor (V\,u)\,,\,\ell \rfloor} \\ \spadesuit & ::= & \blacklozenge \mid \lozenge \qquad h \;\; ::= \;\; \epsilon \mid h \circ (\mathsf{p}\,,\,\mathsf{q}\,,\,m) \qquad m \;\; ::= \;\; V \\ & & \\ \alpha & ::= & \mathsf{q}?(U) \mid \mathsf{q}!\langle U \rangle \\ T, S & ::= & \mathsf{end} \mid \alpha.S \\ H, K & ::= & {\wedge\!\!\wedge}S \mid S\,{\wedge\!\!\wedge} \mid \alpha_1.\cdots.\alpha_n.\,{\wedge\!\!\wedge}S \end{array}$$

Figure 4: Syntax of processes P, Q, configurations M, N, local types T, S, local types with history H, K. Constructs given in $\boxed{\textbf{boxes}}$ appear only at run-time.

(higher-order values) $\lambda x.\,P$, where P is a process. As shown in [15], abstraction passing suffices to express name passing (*delegation*).

Process terms include prefixes for sending and receiving values V, written $u!\langle V \rangle.P$ and $u?(x).P$, respectively. In an improvement with respect to [20, 22], here we consider parallel composition of processes $P \mid Q$ and recursion $\mu X.P$ (which binds the recursive variable X in process P). Process $V\,u$ is the application which substitutes name u on the abstraction V. Constructs for name restriction $(\nu\, n)P$ and inaction $\mathbf{0}$ are standard. Session restriction $(\nu\, s)P$ simultaneously binds all the participant endpoints in P. We write $\mathtt{fv}(P)$ and $\mathtt{fn}(P)$ to denote the sets of free variables and names in P. We assume V in $u!\langle V \rangle.P$ does not include free recursive variables X. If $\mathtt{fv}(P) = \emptyset$, we call P *closed*.

Building upon processes, the syntax of configurations $M, N, \ldots$ includes constructs for *session initiation*:

- configuration $\ell\,\{a!\langle x \rangle.P\}$ denotes the *request* of a service identified with a implemented in P as x;
- conversely, configuration $\ell\,\{a?(x).P\}$ denotes service *acceptance*.

In these constructs, identifiers $\ell, \ell', \ldots$ denote a *location* or *site* (as in, e.g., the distributed π-calculus [11]). Locations indexed by participants, useful in run-time expressions, are denoted $\ell_{[\mathsf{p}]}, \ell_{[\mathsf{q}]}$. Configurations also include inaction $\mathbf{0}$, parallel composition $M \mid N$, name restriction $(\nu\, n)M$, as well as the following *run-time elements*:

- *Running processes* are of the form $\ell_{[\mathsf{p}]} : \wr P \int$, where ℓ is a location that hosts a process P that implements participant p.
- *Monitors* are of the form $s_{[\mathsf{p}]} \lfloor H \cdot \widetilde{x} \cdot \sigma \rfloor^{\spadesuit}$ where s is the session being monitored, p is a participant, H is a history session type (i.e. a session type with "memory"), $\widetilde{x}$ is a set of free variables, and the *store* σ records the value of such variables (see Def. 2.1). These four elements allow us to track the current protocol and state of the monitored process.

  Also, each monitor has a *tag* $\spadesuit$, which can be either *empty* (denoted '$\lozenge$') or *full* (denoted '$\blacklozenge$'). When created all monitors have an empty tag; a full tag indicates that the running process associated to the monitor is currently involved in a decoupled reversible step. We often omit the empty tag (so we write $s_{[\mathsf{p}]}\lfloor H \cdot \widetilde{x} \cdot \sigma \rfloor$ instead of $s_{[\mathsf{p}]}\lfloor H \cdot \widetilde{x} \cdot \sigma \rfloor^{\diamondsuit}$) and write $s_{[\mathsf{p}]}\lfloor H \cdot \widetilde{x} \cdot \sigma \rfloor^{\blacklozenge}$ to emphasize the reversible (red) nature of a monitor with full tag.

- Following [16], we have *message queues* of the form $s : (h_i \star h_o)$, where s is a session, h_i is the input part of the queue, and h_o is the output part of the queue. Each queue contains messages of the form $(\mathsf{p}, \mathsf{q}, m)$ (read: "message m is sent from p to q"). As we will see, the effect of an output prefix in a process is to place the message in its corresponding output queue; conversely, the effect of an input prefix is to obtain the first message from its input queue. Messages in the queue are *never consumed*: a process reads a message $(\mathsf{p}, \mathsf{q}, m)$ by moving it from the (tail of) queue h_o to the (top of) queue h_i. This way, the delimiter '$\star$' distinguishes the *past* of the queue from its *future*.
- We use *running functions* of the form $k\lfloor (V\,u), \ell \rfloor$ to reverse applications $V\,u$. While k is a fresh identifier (key) for this term, ℓ is the location of the running process that contains the application.

We shall write $\mathcal{P}$ and $\mathcal{M}$ to indicate the set of processes and configurations, respectively. We call *agent* an element of the set $\mathcal{A} = \mathcal{M} \cup \mathcal{P}$. We let P, Q to range over $\mathcal{P}$; also, we use L, M, N to range over $\mathcal{M}$ and A, B, C to range over $\mathcal{A}$.

2.2.2 A Decoupled Semantics for Configurations. We define a reduction relation on configurations, coupled with a structural congruence on processes and configurations. Our reduction semantics defines a *decoupled* treatment for reversing communication actions within a protocol. Reduction is thus defined as $\longrightarrow \subset \mathcal{M} \times \mathcal{M}$, whereas structural congruence is defined as $\equiv \subset \mathcal{P}^2 \cup \mathcal{M}^2$. We require auxiliary definitions for *contexts*, *stores*, and *type contexts*.

Evaluation contexts are configurations with one hole '$\bullet$', as defined by the following grammar:

$$\mathbb{E} \quad ::= \quad \bullet \mid M \mid \mathbb{E} \mid (\nu\, n)\,\mathbb{E}$$

General contexts $\mathbb{C}$ are processes or configurations with one hole $\bullet$: they are obtained by replacing one occurrence of $\mathbf{0}$ (either as a process or as a configuration) with $\bullet$. A congruence on processes and configurations is an equivalence $\mathfrak{R}$ that is closed under general contexts: $P \,\mathfrak{R}\, Q \implies \mathbb{C}[P] \,\mathfrak{R}\, \mathbb{C}[Q]$ and $M \mathfrak{R} N \implies \mathbb{C}[M] \,\mathfrak{R}\, \mathbb{C}[N]$. We define $\equiv$ as the smallest congruence on processes and configurations that satisfies the axioms in Fig. 5. Most axioms are standard and/or similar to those for the π-calculus [25]; the exception is $\ell_{[\mathsf{r}]} : \wr (\nu\, a)P \int \equiv (\nu\, a)\ell_{[\mathsf{r}]} : \wr P \int$, which allows a bound name to cross the boundaries of a running process, thus relating processes and configurations. A relation $\mathfrak{R}$ on configurations is *evaluation-closed* if it satisfies the following rules:

$$(\textsc{Ctx})\ \frac{M \,\mathfrak{R}\, N}{\mathbb{E}[M] \,\mathfrak{R}\, \mathbb{E}[N]} \qquad (\textsc{Eqv})\ \frac{M \equiv M' \qquad M' \,\mathfrak{R}\, N' \qquad N' \equiv N}{M \,\mathfrak{R}\, N}$$

The state of monitored processes is formalized as follows:

Definition 2.1. A store σ is a mapping from variables to values. Given a store σ, a variable x, and a value V, the *update* $\sigma[x \mapsto V]$ and the *reverse update* $\sigma \setminus x$ are defined as follows:

$$\sigma[x \mapsto V] = \begin{cases} \sigma \cup \{(x, V)\} & \text{if } x \notin \mathsf{dom}(\sigma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\sigma \setminus x = \begin{cases} \sigma_1 & \text{if } \sigma = \sigma_1 \cup \{(x, V)\} \\ \sigma & \text{otherwise} \end{cases}$$

$$A \mid \mathbf{0} \equiv A \quad A \mid B \equiv B \mid A \quad A \mid (B \mid C) \equiv (A \mid B) \mid C$$
$$A \mid (\nu\, n)B \equiv (\nu\, n)(A \mid B)\ (n \notin \mathsf{fn}(P)) \quad (\nu\, n)\mathbf{0} \equiv \mathbf{0}$$
$$\mu X.P \equiv P\{\mu X.P/X\} \quad A \equiv B \text{ if } A \equiv_\alpha B$$
$$\ell_{[\mathsf{r}]} : \wr(\nu\, a)P\int \equiv (\nu\, a)\ell_{[\mathsf{r}]} : \wr P \int$$

Figure 5: Structural Congruence

Together with local types with history, the following notion of type context allows us to keep the current protocol state inside monitors:

Definition 2.2. Let $k, k', \ldots$ denote fresh name identifiers. We define *type contexts* as (local) types with one hole, denoted "$\bullet$":

$$\mathbb{T}, \mathbb{S} \quad ::= \quad \bullet \mid \alpha.\mathbb{T} \mid k.\mathbb{T} \mid (\ell, \ell_1, \ell_2).\mathbb{T}$$

Type contexts $k.\mathbb{T}$ and $(\ell, \ell_1, \ell_2).\mathbb{T}$ will be instrumental in formalizing reversibility of name applications and thread spawning, respectively, which are not described by local types. This way, we will often have monitors of the form $s_{[\mathsf{p}]}\lfloor \mathbb{T}[\,^{\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor^{\blacklozenge}$, where $\mathbb{T}$ and S describe past and future protocol steps for p, respectively.

Abstraction passing can implement a form of *session delegation*, for received abstractions $\lambda x.\, P$ can contain free session names (indexed by participant identities). The following definition identifies those names:

Definition 2.3. Let h and p be a queue and a participant, respectively. Also, let $\{(\mathsf{q}_1, \mathsf{p}, \lambda x_1.\, P_1), \ldots, (\mathsf{q}_k, \mathsf{p}, \lambda x_k.\, P_k)\}$ denote the (possibly empty) set of messages in h containing abstractions sent to p. We write $\mathbf{roles}(\mathsf{p}, h)$ to denote the set of participant identities occurring in $P_1, \ldots, P_k$.

The reduction relation $\longrightarrow$ is defined as the union of the forward and backward reduction relations, denoted $\twoheadrightarrow$ and $\rightsquigarrow$, respectively. That is, $\longrightarrow = \twoheadrightarrow \cup \rightsquigarrow$. Relations $\twoheadrightarrow$ and $\rightsquigarrow$ are the smallest evaluation-closed relations satisfying the rules in Figs. 6 and 7. We indicate with $\longrightarrow^*$, $\twoheadrightarrow^*$, and $\rightsquigarrow^*$ the reflexive and transitive closure of $\longrightarrow$, $\twoheadrightarrow$ and $\rightsquigarrow$, respectively. We first discuss the forward reduction rules (Fig. 6), omitting empty tags $\diamondsuit$:

- ▶ Rule (Init) initiates a choreography G with n participants. Given the composition of one service request and $n-1$ service accepts (all along a, available in different locations ℓ_i), this rule sets up the run-time elements: running processes and monitors—one for each participant, with empty tag (omitted)—and the empty session queue. A unique session identifier (s in the rule) is also created. The processes are inserted in their respective running structures, and instantiated with an appropriate session name. Similarly, the local types for each participant are inserted in their respective monitor, with the cursor $^{\wedge}$ at the beginning.
- ▶ Rule (Out) starts the output of value V from p to q. Given an output-prefixed process as running process, and a monitor with a local type supporting an output action, reduction adds the

$$\textsc{(Init)}\ \frac{\mathsf{pa}(G) = \{\mathsf{p}_1, \cdots, \mathsf{p}_n\} \qquad \forall \mathsf{p}_i \in \mathsf{pa}(G).\, G{\downarrow_{\mathsf{p}_i}} = T_i}{\ell_1\, \{a!\langle x_1 : T_1\rangle.P_1\} \mid \prod_{i\in\{2,\cdots,n\}} \ell_i\, \{a?(x_i : T_i).P_i\} \twoheadrightarrow (\nu\, s)\left(\prod_{i\in\{1,\cdots,n\}} \ell_{i[\mathsf{p}_i]} : \wr P_i\{^{s_{[\mathsf{p}_i]}}/_{x_i}\} \int \mid s_{[\mathsf{p}_i]} \lfloor {\wedge} T_i \cdot x_i \cdot [x_i \mapsto a] \rfloor \mid s : (\epsilon \star \epsilon)\right)}$$

$$\textsc{(Out)}\ \frac{\mathsf{p} = \mathsf{r} \ \vee\ \mathsf{p} \in \mathbf{roles}(\mathsf{r}, h_i)}{\begin{array}{c}\ell_{[\mathsf{r}]} : \wr s_{[\mathsf{p}]}!\langle V\rangle.P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\,{\wedge}\mathsf{q}!\langle U\rangle.S] \cdot \widetilde{x} \cdot \sigma \rfloor \mid s : (h_i \star h_o) \\ \twoheadrightarrow \\ \ell_{[\mathsf{r}]} : \wr P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\mathsf{q}!\langle U\rangle.\ {\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor \mid s : (h_i \star h_o \circ (\mathsf{p}, \mathsf{q}, \sigma(V)))\end{array}}$$

$$\textsc{(In)}\ \frac{\mathsf{p} = \mathsf{r} \ \vee\ \mathsf{p} \in \mathbf{roles}(\mathsf{r}, h_i)}{\begin{array}{c}\ell_{[\mathsf{r}]} : \wr s_{[\mathsf{p}]}?(y).P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\,{\wedge}\mathsf{q}?\langle U\rangle.S] \cdot \widetilde{x} \cdot \sigma \rfloor \mid s : (h_i \star (\mathsf{q}, \mathsf{p}, V) \circ h_o) \\ \twoheadrightarrow \\ \ell_{[\mathsf{r}]} : \wr P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\mathsf{q}?\langle U\rangle.\ {\wedge}S] \cdot \widetilde{x}, y \cdot \sigma[y \mapsto V] \rfloor \mid s : (h_i \circ (\mathsf{q}, \mathsf{p}, V) \star h_o)\end{array}}$$

$$\textsc{(Beta)}\ \frac{\sigma(V) = \lambda x.\, P}{\ell_{[\mathsf{p}]} : \wr (V\, w) \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\,{\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor \twoheadrightarrow (\nu\, k)\ \left(\ell_{[\mathsf{p}]} : \wr P\{^{\sigma(w)}/_{x}\} \int \mid k \lfloor (V\, w), \ell \rfloor \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[k.\ {\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor\right)}$$

$$\textsc{(Spawn)}\ \frac{}{\ell_{[\mathsf{p}]} : \wr P \mid Q \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\,{\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor \twoheadrightarrow (\nu\, \ell_1, \ell_2)\ \left(\ell_{[\mathsf{p}]} : \wr \mathbf{0} \int \mid \ell_{1[\mathsf{p}]} : \wr P \int \mid \ell_{2[\mathsf{p}]} : \wr Q \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[(\ell, \ell_1, \ell_2).\ {\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor\right)}$$

Figure 6: Decoupled semantics for configurations: Forward reduction ($\twoheadrightarrow$).

$$\textsc{(RInit)}\ \frac{\mathsf{pa}(G) = \{\mathsf{p}_1, \cdots, \mathsf{p}_n\} \qquad \forall \mathsf{p}_i \in \mathsf{pa}(G).\, G{\downarrow_{\mathsf{p}_i}} = T_i \qquad Q_i = P_i\{^{s_{[\mathsf{p}_i]}}/_{x_i}\}}{(\nu\, s)\left(\prod_{i\in\{1,\cdots,n\}} \ell_{i[\mathsf{p}_i]} : \wr Q_i \int \mid s_{[\mathsf{p}_i]} \lfloor {\wedge} T_i \cdot x_i \cdot [x_i \mapsto a] \rfloor^{\diamond} \mid s : (\epsilon \star \epsilon)\right) \rightsquigarrow \ell_1\, \{a!\langle x_1 : T_1\rangle.P_1\} \mid \prod_{i\in\{2,\cdots,n\}} \ell_i\, \{a?(x_i : T_i).P_i\}}$$

$$\textsc{(RollS)}\ \frac{}{\begin{array}{c}s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\mathsf{q}?\langle U\rangle.\ {\wedge}T] \cdot \widetilde{x} \cdot \sigma_1 \rfloor^{\diamond} \mid s_{[\mathsf{q}]} \lfloor \mathbb{S}\,[\mathsf{p}!\langle U\rangle.\ {\wedge}S] \cdot \widetilde{y} \cdot \sigma_2 \rfloor^{\diamond} \mid s : (h_i \star h_o) \\ \rightsquigarrow \\ s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\mathsf{q}?\langle U\rangle.\ {\wedge}T] \cdot \widetilde{x} \cdot \sigma_1 \rfloor^{\blacklozenge} \mid s_{[\mathsf{q}]} \lfloor \mathbb{S}\,[\mathsf{p}!\langle U\rangle.\ {\wedge}S] \cdot \widetilde{y} \cdot \sigma_2 \rfloor^{\blacklozenge} \mid s : (h_i \star h_o)\end{array}}$$

$$\textsc{(ROut)}\ \frac{\mathsf{p} = \mathsf{r} \ \vee\ \mathsf{p} \in \mathbf{roles}(\mathsf{r}, h_i)}{\ell_{[\mathsf{r}]} : \wr P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\mathsf{q}!\langle U\rangle.\ {\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor^{\blacklozenge} \mid s : (h_i \star (\mathsf{p}, \mathsf{q}, V) \circ h_o) \rightsquigarrow \ell_{[\mathsf{r}]} : \wr s_{[\mathsf{p}]}!\langle V\rangle.P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\,{\wedge}\mathsf{q}!\langle U\rangle.S] \cdot \widetilde{x} \cdot \sigma \rfloor^{\diamond} \mid s : (h_i \star h_o)}$$

$$\textsc{(RIn)}\ \frac{\mathsf{p} = \mathsf{r} \ \vee\ \mathsf{p} \in \mathbf{roles}(\mathsf{r}, h_i)}{\begin{array}{c}\ell_{[\mathsf{r}]} : \wr P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\mathsf{q}?\langle U\rangle.\ {\wedge}S] \cdot \widetilde{x}, y \cdot \sigma \rfloor^{\blacklozenge} \mid s : (h_i \circ (\mathsf{q}, \mathsf{p}, V) \star h_o) \\ \rightsquigarrow \\ \ell_{[\mathsf{r}]} : \wr s_{[\mathsf{p}]}?(y).P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\,{\wedge}\mathsf{q}?\langle U\rangle.S] \cdot \widetilde{x} \cdot \sigma \setminus y \rfloor^{\diamond} \mid s : (h_i \star (\mathsf{q}, \mathsf{p}, V) \circ h_o)\end{array}}$$

$$\textsc{(RBeta)}\ \frac{}{(\nu\, k)\ \left(\ell_{[\mathsf{p}]} : \wr Q \int \mid k \lfloor (V\, w), \ell \rfloor \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[k.\ {\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor\right) \rightsquigarrow \ell_{[\mathsf{p}]} : \wr (V\, w) \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\,{\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor}$$

$$\textsc{(RSpawn)}\ \frac{}{(\nu\, \ell_1, \ell_2)\ \left(\ell_{[\mathsf{p}]} : \wr \mathbf{0} \int \mid \ell_{1[\mathsf{p}]} : \wr P \int \mid \ell_{2[\mathsf{p}]} : \wr Q \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[(\ell, \ell_1, \ell_2).\ {\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor\right) \rightsquigarrow \ell_{[\mathsf{p}]} : \wr P \mid Q \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}\,[\,{\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor}$$

Figure 7: Decoupled semantics for configurations: Backwards reduction ($\rightsquigarrow$).

message $(\mathsf{p}, \mathsf{q}, \sigma(V))$ to the output part of the session queue (where σ is the current store). Also, the cursor within the local type is moved accordingly. In this rule (but also in several other rules), premise $\mathsf{p} = \mathsf{r} \vee \mathsf{p} \in \mathbf{roles}(\mathsf{r}, h_i)$ allows performing actions on names previously received via abstraction passing.

- ▶ Rule (In) allows a participant p to receive a value V from q: it simply takes the first element of the output part of the queue and places it in the input part. The cursor of the local type and state in the monitor for p are updated accordingly.
- ▶ Rule (Beta) handles name applications. Reduction creates a fresh identifier (k in the rule) for the running function, which keeps (i) the structure of the process prior to application, and (ii) the identifier of the running process that "invokes" the application. Notice that k is recorded also in the monitor: this is necessary to undo applications in the proper order. To determine the actual abstraction and the name applied, we use σ.
- ▶ Rule (Spawn) handles parallel composition. Location ℓ is "split" into running processes with fresh identifiers (ℓ_1, ℓ_2 in the rule). This split is recorded in the monitor.

Now we comment on the backward rules (Fig. 7) which, in most cases, change the monitor tags from $\diamondsuit$ into ♦:

- ◀ Rule (RInit) undoes session establishment. It requires that local types for every participant are at the beginning of the protocol, and empty session queue and process stacks. Run-time elements are discarded; located service accept/requests are reinstated.
- ◀ Rule (RollS) starts to undo an input-output synchronization between p and q. Enabled when there are complementary session types in the two monitors, this rule changes the monitor tags from $\diamondsuit$ to ♦. This way, the undoing of input and output actions occurs in a decoupled way. Rule (RollC) is the analog of (RollS) but for synchronizations originated in labeled choices.
- ◀ Rule (ROut) undoes an output. This is only possible for a monitor tagged with ♦, exploiting the first message in the input queue. After reduction, the process prefix is reinstated, the cursor is adjusted, the message is removed from the queue, the monitor is tagged again with $\diamondsuit$. Rule (RIn) is the analog of Rule (ROut). In this case, we also need to update the state of store σ.
- ◀ Rule (RBeta) undoes β-reduction, reinstating the application. The running function disappears, using the information in the monitor (k in the rule). Rule (RSpawn) undoes the spawn of a parallel thread, using the identifiers in the monitor.

We now illustrate our reversible process model with an example.

3 EXAMPLE: THE THREE-BUYER PROTOCOL

We illustrate our framework by presenting a reversible variant of the *Three-Buyer protocol* (see, e.g., [5]) with abstraction passing (code mobility), one of the distinctive traits of our framework.

The protocol involves three buyers—Alice (A), Bob (B), and Carol (C)—who interact with a Vendor (V) as follows:

1. Alice sends a book title to Vendor, which replies back to Alice and Bob with a quote. Alice tells Bob how much she can contribute.
2. Bob notifies Vendor and Alice that he agrees with the price, and asks Carol to assist him in completing the protocol. To delegate his remaining interactions with Alice and Vendor to Carol, Bob sends her the code she must execute.
3. Carol continues the rest of the protocol with Vendor and Alice as if she were Bob. She sends Bob's address (contained in the mobile code she received) to Vendor.
4. Vendor answers to Alice and Carol (who represents Bob) with the delivery date.

We formalize this protocol is as the global type G below. We write $\mathsf{p} \to \{\mathsf{q}_1, \mathsf{q}_2\} : \langle U \rangle.G$ as a shorthand notation for $\mathsf{p} \to \mathsf{q}_1 : \langle U \rangle.\mathsf{p} \to \mathsf{q}_2 : \langle U \rangle.G$ (and similarly for local types). We write $\{\!\{\diamond\}\!\}$ to denote the type $\mathsf{end} \to \diamond$, associated to a *thunk process* $\lambda x.\,P$ with $x \notin \mathtt{fn}(P)$, written $\{\!\{P\}\!\}$. A thunk is an inactive process; it can be activated by applying to it a dummy name of type end, denoted $*$. This way, we have $(\{\!\{P\}\!\}\, *) \twoheadrightarrow P$.

$$
\begin{aligned}
G = {} & \mathsf{A} \to \mathsf{V} : \langle \mathsf{title} \rangle.\mathsf{V} \to \{\mathsf{A}, \mathsf{B}\} : \langle \mathsf{price} \rangle.\mathsf{A} \to \mathsf{B} : \langle \mathsf{share} \rangle. \\
& \mathsf{B} \to \{\mathsf{A}, \mathsf{V}\} : \langle \mathsf{OK} \rangle. \\
& \mathsf{B} \to \mathsf{C} : \langle \mathsf{share} \rangle.\mathsf{B} \to \mathsf{C} : \langle \{\!\{\diamond\}\!\} \rangle. \\
& \mathsf{B} \to \mathsf{V} : \langle \mathsf{address} \rangle.\mathsf{V} \to \mathsf{B} : \langle \mathsf{date} \rangle.\mathsf{end}
\end{aligned}
$$

where price and share are base types treated as integers, and title, OK, address, and date are base types treated as strings.

Then we have the following projections of G onto local types:

$$
\begin{aligned}
G{\downarrow}_{\mathsf{V}} &= \mathsf{A}?\langle \mathsf{title} \rangle.\{\mathsf{A}, \mathsf{B}\}!\langle \mathsf{price} \rangle.\mathsf{B}?\langle \mathsf{OK} \rangle.\mathsf{B}?\langle \mathsf{address} \rangle.\mathsf{B}!\langle \mathsf{date} \rangle.\mathsf{end} \\
G{\downarrow}_{\mathsf{A}} &= \mathsf{V}!\langle \mathsf{title} \rangle.\mathsf{V}?\langle \mathsf{price} \rangle.\mathsf{B}!\langle \mathsf{share} \rangle.\mathsf{B}?\langle \mathsf{OK} \rangle.\mathsf{end} \\
G{\downarrow}_{\mathsf{B}} &= \mathsf{V}?\langle \mathsf{price} \rangle.\mathsf{A}?\langle \mathsf{share} \rangle.\{\mathsf{A}, \mathsf{V}\}!\langle \mathsf{OK} \rangle.\mathsf{C}!\langle \mathsf{share} \rangle.\mathsf{C}!\langle \{\!\{\diamond\}\!\} \rangle. \\
&\qquad \mathsf{V}!\langle \mathsf{address} \rangle.\mathsf{V}?\langle \mathsf{date} \rangle.\mathsf{end} \\
G{\downarrow}_{\mathsf{C}} &= \mathsf{B}?\langle \mathsf{share} \rangle.\mathsf{B}?\langle \{\!\{\diamond\}\!\} \rangle.\mathsf{end}
\end{aligned}
$$

We now give processes for each participant:

$$
\begin{aligned}
\text{Vendor} &= d!\langle x : G{\downarrow}_{\mathsf{V}} \rangle.x?(t).x!\langle price(t) \rangle.x!\langle price(t) \rangle. \\
&\qquad x?(ok).x?(a).x!\langle date \rangle.\mathbf{0} \\
\text{Alice} &= d?(y : G{\downarrow}_{\mathsf{A}}).y!\langle \text{‘Logicomix’} \rangle.y?(p).y!\langle h \rangle.y?(ok).\mathbf{0} \\
\text{Bob} &= d?(z : G{\downarrow}_{\mathsf{B}}).z?(p).z?(h).z!\langle ok \rangle.z!\langle ok \rangle.z!\langle h \rangle. \\
&\qquad z!\Big\langle \{\!\{ z!\langle \text{‘Lucca, 55100’} \rangle.z?(d).\mathbf{0} \}\!\} \Big\rangle.\mathbf{0} \\
\text{Carol} &= d?(w : G{\downarrow}_{\mathsf{C}}).w?(h).w?(code).(code\, *)
\end{aligned}
$$

where $price(\cdot)$ returns a value of type price given a title. Observe how Bob's implementation sends part of its protocol to Carol in the form of a thunk containing his session name and address. This is how abstraction passing may serve to implement session delegation.

The whole system, given by configuration M below, is obtained by placing these process implementations in appropriate locations:

$$
M = \ell_1\,\{\text{Vendor}\} \mid \ell_2\,\{\text{Alice}\} \mid \ell_3\,\{\text{Bob}\} \mid \ell_4\,\{\text{Carol}\}
$$

From M, the session starts with an application of Rule (Init):

$$
\begin{aligned}
M \twoheadrightarrow (\nu\, s)\Big(& \ell_{1[\mathsf{V}]} : \wr V_1\{^{s[\mathsf{V}]}/x\} \int \mid s_{[\mathsf{V}]} \lfloor\, {}^{\wedge}G{\downarrow}_{\mathsf{V}} \cdot x \cdot [x \mapsto d] \rfloor^{\diamondsuit} \\
& \mid \ell_{2[\mathsf{A}]} : \wr A_1\{^{s[\mathsf{A}]}/y\} \int \mid s_{[\mathsf{A}]} \lfloor\, {}^{\wedge}G{\downarrow}_{\mathsf{A}} \cdot y \cdot [y \mapsto d] \rfloor^{\diamondsuit} \\
& \mid \ell_{3[\mathsf{B}]} : \wr B_1\{^{s[\mathsf{B}]}/z\} \int \mid s_{[\mathsf{B}]} \lfloor\, {}^{\wedge}G{\downarrow}_{\mathsf{B}} \cdot z \cdot [z \mapsto d] \rfloor^{\diamondsuit} \\
& \mid \ell_{4[\mathsf{C}]} : \wr C_1\{^{s[\mathsf{C}]}/w\} \int \mid s_{[\mathsf{C}]} \lfloor\, {}^{\wedge}G{\downarrow}_{\mathsf{C}} \cdot w \cdot [w \mapsto d] \rfloor^{\diamondsuit} \\
& \mid s : (\epsilon \star \epsilon) \Big) = M_1
\end{aligned}
$$

where V_1, A_1, B_1, and C_1 stand for the continuation of processes Vendor, Alice, Bob, and Carol after the service request/declaration. So, e.g., $A_1 = y!\langle \text{‘Logicomix’} \rangle.y?(p).y!\langle h \rangle.y?(ok).\mathbf{0}$. Notice also how session initialization instantiates variable z in the thunk contained in Bob's implementation with endpoint $s_{[\mathsf{B}]}$. We use configuration M_1 to illustrate some forward and backward reductions.

From M_1 we could either undo the reduction (using Rule (RInit)) or execute the communication from Alice to Vendor (using two rules: (Out) and (In)). This latter option would be as follows:

$$\begin{aligned}
M_1 \twoheadrightarrow (\nu\, s)(\, &\ell_{2[\mathsf{A}]} : \wr s_{[\mathsf{A}]}?(p).s_{[\mathsf{A}]}!\langle h\rangle.s_{[\mathsf{A}]}?(ok).\mathbf{0}\int \\
&\mid s_{[\mathsf{A}]}\lfloor \mathsf{V}!\langle\mathsf{title}\rangle.\,{}^{\wedge}\mathsf{V}?\langle\mathsf{price}\rangle.\mathsf{B}!\langle\mathsf{share}\rangle.\mathsf{B}?\langle\mathsf{OK}\rangle.\mathsf{end}\cdot y\cdot[y\mapsto d]\rfloor^{\diamond} \\
&\mid N_2 \mid s : (\epsilon \star (\mathsf{A}\,,\ \mathsf{V}\,,\ \text{`Logicomix'}))\ ) = M_2
\end{aligned}$$

where N_2 stands for the running processes and monitors for Vendor, Bob, and Carol, not involved in the reduction. We now have:

$$\begin{aligned}
M_2 \twoheadrightarrow (\nu\, s)(\, &\ell_{1[\mathsf{V}]} : \wr s_{[\mathsf{V}]}!\langle \mathit{price}(t)\rangle.s_{[\mathsf{V}]}!\langle \mathit{price}(t)\rangle.s_{[\mathsf{V}]}?(ok). \\
&\quad s_{[\mathsf{V}]}?(a).s_{[\mathsf{V}]}!\langle \mathit{date}\rangle.\mathbf{0}\int \\
&\mid s_{[\mathsf{V}]}\lfloor \mathsf{A}?\langle\mathsf{title}\rangle.\,{}^{\wedge}\{\mathsf{A},\mathsf{B}\}!\langle\mathsf{price}\rangle.T_{\mathsf{V}}\cdot x, t\cdot\sigma_3\rfloor^{\diamond} \mid N_3 \\
&\mid s : ((\mathsf{A}\,,\ \mathsf{V}\,,\ \text{`Logicomix'}) \star \epsilon)\ ) = M_3
\end{aligned}$$

where $\sigma_3 = [x \mapsto d], [t \mapsto \text{`Logicomix'}]$ is the resulting store, $T_{\mathsf{V}} = \mathsf{B}?\langle\mathsf{OK}\rangle.\mathsf{B}?\langle\mathsf{address}\rangle.\mathsf{B}!\langle\mathsf{date}\rangle.\mathsf{end}$, and N_3 stands for the participants not involved in the reduction. Observe that the cursors in monitors $s_{[\mathsf{V}]}$ and $s_{[\mathsf{A}]}$ have evolved, and that message from A to V has now been moved to the input queue.

We illustrate reversibility by showing how to return to M_1 starting from M_3. We need to apply three rules: (RollS), (RIn), and (ROut). Reversibility is decoupled in the sense that there is no fixed order in which the latter two rules should be applied; below we give just a possible sequence. First, Rule (RollS) modifies the tags of monitors $s_{[\mathsf{V}]}$ and $s_{[\mathsf{A}]}$, leaving the rest unchanged:

$$\begin{aligned}
M_3 \rightsquigarrow (\nu\, s)(\, &\ell_{1[\mathsf{V}]} : \wr s_{[\mathsf{V}]}!\langle \mathit{price}(t)\rangle.s_{[\mathsf{V}]}!\langle \mathit{price}(t)\rangle.s_{[\mathsf{V}]}?(ok). \\
&\quad s_{[\mathsf{V}]}?(a).s_{[\mathsf{V}]}!\langle \mathit{date}\rangle.\mathbf{0}\int \\
&\mid s_{[\mathsf{V}]}\lfloor \mathsf{A}?\langle\mathsf{title}\rangle.\,{}^{\wedge}\{\mathsf{A},\mathsf{B}\}!\langle\mathsf{price}\rangle.T_{\mathsf{B}}\cdot x, t\cdot\sigma_3\rfloor^{\blacklozenge} \\
&\mid \ell_{2[\mathsf{A}]} : \wr s_{[\mathsf{A}]}?(p).s_{[\mathsf{A}]}!\langle h\rangle.s_{[\mathsf{A}]}?(ok).\mathbf{0}\int \\
&\mid s_{[\mathsf{A}]}\lfloor \mathbb{T}_4\,[\,{}^{\wedge}\mathsf{V}?\langle\mathsf{price}\rangle.\mathsf{B}!\langle\mathsf{share}\rangle.\mathsf{B}?\langle\mathsf{OK}\rangle.\mathsf{end}]\cdot y\cdot[y\mapsto d]\rfloor^{\blacklozenge} \\
&\mid N_4 \mid s : ((\mathsf{A}\,,\ \mathsf{V}\,,\ \text{`Logicomix'}) \star \epsilon)\ ) = M_4
\end{aligned}$$

where $\mathbb{T}_4\,[\bullet] = \mathsf{V}!\langle\mathsf{title}\rangle.\bullet$ and , as before, N_4 represents participants not involved in the reduction. M_4 has several possible forward and backward reductions. One particular reduction uses Rule (RIn) to undo the input at V:

$$\begin{aligned}
M_4 \rightsquigarrow (\nu\, s)(\, &\ell_{1[\mathsf{V}]} : \wr s_{[\mathsf{V}]}?(t).s_{[\mathsf{V}]}!\langle \mathit{price}(t)\rangle.s_{[\mathsf{V}]}!\langle \mathit{price}(t)\rangle. \\
&\quad s_{[\mathsf{V}]}?(ok).s_{[\mathsf{V}]}?(a).s_{[\mathsf{V}]}!\langle \mathit{date}\rangle.\mathbf{0}\int \\
&\mid s_{[\mathsf{V}]}\lfloor\, {}^{\wedge}\mathsf{A}?\langle\mathsf{title}\rangle.\{\mathsf{A},\mathsf{B}\}!\langle\mathsf{price}\rangle.T_{\mathsf{B}}\cdot x\cdot[x\mapsto d]]\rfloor^{\diamond} \\
&\mid \ell_{2[\mathsf{A}]} : \wr s_{[\mathsf{A}]}?(p).s_{[\mathsf{A}]}!\langle h\rangle.s_{[\mathsf{A}]}?(ok).\mathbf{0}\int \\
&\mid s_{[\mathsf{A}]}\lfloor \mathbb{T}_4\,[\,{}^{\wedge}\mathsf{V}?\langle\mathsf{price}\rangle.\mathsf{B}!\langle\mathsf{share}\rangle.\mathsf{B}?\langle\mathsf{OK}\rangle.\mathsf{end}]\cdot y\cdot[y\mapsto d]]\rfloor^{\blacklozenge} \\
&\mid N_4 \mid s : (\epsilon \star (\mathsf{A}\,,\ \mathsf{V}\,,\ \text{`Logicomix'}))\ ) = M_5
\end{aligned}$$

Just as an application of Rule (RollS) need not be immediately followed by an application of Rule (RIn), an application of Rule (RIn) need not be immediately followed by an application of Rule (ROut). A particular reduction from M_5 undoes the output at A:

$$\begin{aligned}
M_5 \rightsquigarrow (\nu\, s)(\, &\ell_{1[\mathsf{V}]} : \wr s_{[\mathsf{V}]}?(t).s_{[\mathsf{V}]}!\langle \mathit{price}(t)\rangle.s_{[\mathsf{V}]}!\langle \mathit{price}(t)\rangle. \\
&\quad s_{[\mathsf{V}]}?(ok).s_{[\mathsf{V}]}?(a).s_{[\mathsf{V}]}!\langle \mathit{date}\rangle.\mathbf{0}\int \\
&\mid s_{[\mathsf{V}]}\lfloor {}^{\wedge}\mathsf{A}?\langle\mathsf{title}\rangle.\{\mathsf{A},\mathsf{B}\}!\langle\mathsf{price}\rangle.T_{\mathsf{B}}\cdot x\cdot[x\mapsto d]\rfloor^{\diamond} \\
&\mid \ell_{2[\mathsf{A}]} : \wr s_{[\mathsf{A}]}!\langle\text{`Logicomix'}\rangle.s_{[\mathsf{A}]}?(p).s_{[\mathsf{A}]}!\langle h\rangle.s_{[\mathsf{A}]}?(ok).\mathbf{0}\int \\
&\mid s_{[\mathsf{A}]}\lfloor {}^{\wedge}\mathsf{V}!\langle\mathsf{title}\rangle.\mathsf{V}?\langle\mathsf{price}\rangle.\mathsf{B}!\langle\mathsf{share}\rangle.\mathsf{B}?\langle\mathsf{OK}\rangle.\mathsf{end}\cdot y\cdot[y\mapsto d]\rfloor^{\diamond} \\
&\mid N_4 \mid s : (\epsilon \star \epsilon)\ ) = M_6
\end{aligned}$$

Clearly, $M_6 = M_1$. Summing up, the synchronization realized by the (forward) reduction sequence $M_1 \twoheadrightarrow M_2 \twoheadrightarrow M_3$ can be reversed by the (backward) reduction sequence $M_3 \rightsquigarrow M_4 \rightsquigarrow M_5 \rightsquigarrow M_6$.

To illustrate abstraction passing, let us assume that M_3 above follows a sequence of forward reductions until the configuration:

$$\begin{aligned}
M_7 = (\nu\, s)(\, &\ell_{3[\mathsf{B}]} : \wr s_{[\mathsf{B}]}!\big\langle \{\!\{s_{[\mathsf{B}]}!\langle\text{`Lucca, 55100'}\rangle.s_{[\mathsf{B}]}?(d).\mathbf{0}\}\!\}\big\rangle.\mathbf{0}\int \\
&\mid s_{[\mathsf{B}]}\lfloor \mathbb{T}_7\,[\,{}^{\wedge}\mathsf{C}!\langle\{\!\{\diamond\}\!\}\rangle.\mathsf{V}!\langle\mathsf{address}\rangle.\mathsf{V}?\langle\mathsf{date}\rangle.\mathsf{end}]\cdot z, p, h\cdot\sigma_7\rfloor^{\diamond} \\
&\mid \ell_{4[\mathsf{C}]} : \wr s_{[\mathsf{C}]}?(\mathit{code}).(\mathit{code}\,*)\int \\
&\mid s_{[\mathsf{C}]}\lfloor \mathbb{T}_8\,[\,{}^{\wedge}\mathsf{B}?\langle\{\!\{\diamond\}\!\}\rangle.\mathsf{end}]\cdot w, h\cdot\sigma_8\rfloor^{\diamond} \mid N_5 \mid s : (h_7 \star \epsilon)\ )
\end{aligned}$$

where $\mathbb{T}_7\,[\bullet]$, σ_7, $\mathbb{T}_8\,[\bullet]$, σ_8, and h_7 capture past interactions as follows:

$$\begin{aligned}
\mathbb{T}_7\,[\bullet] &= \mathsf{V}?\langle\mathsf{price}\rangle.\mathsf{A}?\langle\mathsf{share}\rangle.\{\mathsf{A},\mathsf{V}\}!\langle\mathsf{OK}\rangle.\mathsf{C}!\langle\mathsf{share}\rangle.\bullet \\
\sigma_7 &= [z\mapsto d], [p\mapsto \mathit{price}(\text{`Logicomix'})], [h\mapsto 120] \\
\mathbb{T}_8\,[\bullet] &= \mathsf{B}?\langle\mathsf{share}\rangle.\,\bullet \qquad \sigma_8 = [w\mapsto d], [h\mapsto 120] \\
h_7 &= (\mathsf{A}\,,\ \mathsf{V}\,,\ \text{`Logicomix'}) \\
&\quad \circ (\mathsf{V}\,,\ \mathsf{A}\,,\ \mathit{price}(\text{`Logicomix'})) \circ (\mathsf{V}\,,\ \mathsf{B}\,,\ \mathit{price}(\text{`Logicomix'})) \\
&\quad \circ (\mathsf{A}\,,\ \mathsf{B}\,,\ 120) \circ (\mathsf{B}\,,\ \mathsf{A}\,,\ \text{`ok'}) \circ (\mathsf{B}\,,\ \mathsf{V}\,,\ \text{`ok'}) \circ (\mathsf{B}\,,\ \mathsf{C}\,,\ 120)
\end{aligned}$$

If $M_7 \twoheadrightarrow\ \twoheadrightarrow M_8$ by using Rules (Out) and (In) we would have a higher-order communication:

$$\begin{aligned}
M_8 =\ (\nu\, s)(\, &\ell_{3[\mathsf{B}]} : \wr\mathbf{0}\int \\
&\mid s_{[\mathsf{B}]}\lfloor \mathbb{T}_7\,[\mathsf{C}!\langle\{\!\{\diamond\}\!\}\rangle.\,{}^{\wedge}\mathsf{V}!\langle\mathsf{address}\rangle.\mathsf{V}?\langle\mathsf{date}\rangle.\mathsf{end}]\cdot z, p, h\cdot\sigma_7\rfloor^{\diamond} \\
&\mid \ell_{4[\mathsf{C}]} : \wr(\mathit{code}\,*)\int \\
&\mid s_{[\mathsf{C}]}\lfloor \mathbb{T}_8\,[\mathsf{B}?\langle\{\!\{\diamond\}\!\}\rangle.\,{}^{\wedge}\mathsf{end}]\cdot w, h, \mathit{code}\cdot\sigma_9\rfloor^{\diamond} \\
&\mid N_5 \mid s : (h_7 \circ (\mathsf{B}\,,\ \mathsf{C}\,,\ \{\!\{s_{[\mathsf{B}]}!\langle\text{`Lucca, 55100'}\rangle.s_{[\mathsf{B}]}?(d).\mathbf{0}\}\!\}) \star \epsilon)\ )
\end{aligned}$$

where $\sigma_9 = \sigma_8[\mathit{code} \mapsto \{\!\{s_{[\mathsf{B}]}!\langle\text{`Lucca, 55100'}\rangle.s_{[\mathsf{B}]}?(d).\mathbf{0}\}\!\}]$. We now may apply Rule (Beta) to obtain the actual code sent from B to C:

$$\begin{aligned}
M_8 \twoheadrightarrow\ (\nu\, s)(\nu\, k)(\, &\ell_{4[\mathsf{C}]} : \wr s_{[\mathsf{B}]}!\langle\text{`Lucca, 55100'}\rangle.s_{[\mathsf{B}]}?(d).\mathbf{0}\int \mid N_6 \\
&\mid s_{[\mathsf{B}]}\lfloor \mathbb{T}_7\,[\mathsf{C}!\langle\{\!\{\diamond\}\!\}\rangle.\,{}^{\wedge}\mathsf{V}!\langle\mathsf{address}\rangle.\mathsf{V}?\langle\mathsf{date}\rangle.\mathsf{end}]\cdot z, p, h\cdot\sigma_7\rfloor^{\diamond} \\
&\mid k\lfloor(\mathit{code}\,*)\,,\ \ell_4\rfloor \mid s_{[\mathsf{C}]}\lfloor \mathbb{T}_8\,[\mathsf{B}?\langle\{\!\{\diamond\}\!\}\rangle.k.\,{}^{\wedge}\mathsf{end}]\cdot w, h, \mathit{code}\cdot\sigma_9\rfloor^{\diamond} \\
&\mid s : (h_7 \circ (\mathsf{B}\,,\ \mathsf{C}\,,\ \{\!\{s_{[\mathsf{B}]}!\langle\text{`Lucca, 55100'}\rangle.s_{[\mathsf{B}]}?(d).\mathbf{0}\}\!\}) \star \epsilon)\ ) = M_9
\end{aligned}$$

where N_6 is for the rest of the system. Notice that this reduction has added a running function on a fresh k, which is also used within the type stored in the monitor $s_{[\mathsf{C}]}$.

The reduction $M_8 \twoheadrightarrow M_9$ completes the code mobility from B to C: the now active thunk will execute B's implementation from C's location. Observe that Bob's identity B is "hardwired" in the sent thunk; there is no way for C to execute the code by referring to a participant different from B. This justifies the premise $\mathsf{p} = \mathsf{r} \vee \mathsf{p} \in \mathbf{roles}(\mathsf{r}, h_i)$ present in Rules (Out) and (In) (and in their backward counterparts): when executing previously received mobile code, the participant mentioned in the location (i.e., C) and that mentioned in the located process (i.e., B) may differ. Further forward reductions from M_9 will modify the cursor in the type stored in monitor $s_{[\mathsf{B}]}$ based on the process located at $\ell_{4[\mathsf{C}]}$.

Having introduced our process model and its reversible semantics, we now move on to establish its key properties.

4 MAIN RESULTS

We now establish our main result: we prove that reversibility in our model of choreographic, asynchronous communication is *causally consistent*. We proceed in three steps:

a) First, we introduce an alternative *atomic* semantics and show that it corresponds, in a tight technical sense, to the decoupled semantics in § 2.2.2 (Theorems 4.7 and 4.10).
b) Second, in the light of this correspondence, we establish causal consistency for the atomic semantics, following the approach of Danos and Krivine [7] (Theorem 4.18).
c) Finally, we state a fine-grained, bidirectional connection between the semantics of (high-level) global types with the decoupled semantics of (low-level) configurations (Theorem 4.22).

As a result of these steps, we may transfer causal consistency to choreographies expressed as global types.

4.1 Atomic Semantics vs. Decoupled Semantics

Our main insight is that causal consistency for asynchronous communication can be established by considering a *coarser* synchronous reduction relation. We define *atomic* versions of the forward and backward reduction relations, relying on the rules in Fig. 8.

The *forward atomic reduction*, denoted $\Rrightarrow$, is the smallest evaluation-closed relation that satisfies Rule (AC) (Fig. 8), together with Rules (Init), (Beta), and (Spawn) (Fig. 6). Similarly, the *backward atomic reduction*, denoted $\Lleftarrow$, is the smallest evaluation-closed relation that satisfies Rule (RAC) (Fig. 8), together with Rules (RInit), (RBeta), and (RSpawn) (Fig. 7). We then define the atomic reduction relation $\rightarrowtail$ as $\Rrightarrow \cup \Lleftarrow$.

We start by introducing *reachable* configurations:

Definition 4.1. A configuration M is *initial* if $M \equiv (\nu\, \tilde{n}) \prod_i \ell_i \, \{P_i\}$. A configuration is *reachable*, if it is derived from an initial configuration by using $\longrightarrow$. A configuration is *atomically reachable*, if it is derived from an initial configuration by using $\rightarrowtail$.

To relate the decoupled semantics $\longrightarrow$ (cf. § 2.2.2) with the atomic reduction $\rightarrowtail$ (just defined), we introduce the concept of *stable* configuration. Roughly speaking, in a stable configuration there are no "ongoing" reduction steps. In the forward case, an ongoing step is witnessed by non-empty output queues (which should eventually become empty to complete a synchronization); in the backward case, an ongoing step is witnessed by a marked monitor (which should be eventually unmarked when a synchronization is undone). This way, e.g., in the example of § 3 configurations M_3 and M_7 are stable, whereas M_2 and M_4 are not stable. Reduction $\rightarrowtail$ will move between stable configurations only. We therefore have:

Definition 4.2. A configuration M is *stable*, written $\mathsf{sb}(M)$, if

$$M \equiv \prod_i \ell_i \, \{P_i\} \mid (\nu\, s\tilde{a})\Big(\prod_j \ell_{j_{[\mathsf{p}_j]}} : \wr P_j \int \mid s_{[\mathsf{p}_i]} \lfloor T_i \cdot \widetilde{x}_i \cdot \sigma_i \rfloor^{\diamondsuit} \mid s : (h_1 \star \epsilon)\Big)$$

Reduction $\longrightarrow$ does not preserve stability, but it can be recovered:

Lemma 4.3. *Given M a stable configuration then*

- *if $M \twoheadrightarrow N$ with $\neg\mathsf{sb}(N)$ then there exists an N' such that $N \twoheadrightarrow N'$ and $\mathsf{sb}(N')$;*
- *if $M \rightsquigarrow N$ with $\neg\mathsf{sb}(N)$ then there exists an N' such that $N \rightsquigarrow \, \rightsquigarrow N'$ and $\mathsf{sb}(N')$.*

We may then have:

Corollary 4.4. *If $\mathsf{sb}(M)$ and $M \longrightarrow^* N$ with $\neg\mathsf{sb}(N)$, then there exists an N' such that $N \longrightarrow^* N'$ with $\mathsf{sb}(N')$.*

Proof. By induction on the reduction sequence $M \longrightarrow^* N$. □

We now show the Loop lemma [7], which ensures that every reduction step can be reverted. This lemma will be crucial both in proving a correspondence between atomic and decoupled semantics, and in showing causal consistency of the atomic semantics.

Lemma 4.5 (Loop). *Let M, N be stable and atomically reachable configurations. Then $M \Rrightarrow N$ if and only if $N \Lleftarrow M$.*

Proof. By induction on the derivation of $M \Rrightarrow N$ for the if direction, and on the derivation of $N \Lleftarrow M$ for the converse. □

The following lemma allow us to "rearrange" atomic reduction steps; it will be useful to connect atomic and decoupled reductions.

Lemma 4.6 (Swap). *Let M be a reachable configuration, then:*

- *If $M \twoheadrightarrow^* N_1$ using Rule (Out), and $N_1 \twoheadrightarrow N_2$ by using Rule (In) then $M \twoheadrightarrow \, \twoheadrightarrow N \twoheadrightarrow^* N_2$, for some N;*
- *If $M \rightsquigarrow^* N_1$ using Rule (ROut), and $N_1 \rightsquigarrow N_2$ by using Rule (RIn) then $M \rightsquigarrow \, \rightsquigarrow N \rightsquigarrow^* N_2$, for some N.*

The following theorem is a first connection between decoupled and atomic reductions; its proof is immediate from their definitions:

Theorem 4.7 (Relating $\longrightarrow$ and $\rightarrowtail$). *Let M and N be stable configurations. We have :*

- *$M \Rrightarrow N$ if and only if either $M \twoheadrightarrow N$ or $M \twoheadrightarrow \, \twoheadrightarrow N$;*
- *$M \Lleftarrow N$ if and only if either $M \rightsquigarrow N$ or $M \rightsquigarrow \, \rightsquigarrow \, \rightsquigarrow N$.*

We now embark ourselves in providing a tighter formal connection between $\longrightarrow$ and $\rightarrowtail$, using *back-and-forth barbed bisimulations* [18]. We shall work with binary relations on configurations, written $\mathfrak{R} \subseteq \mathcal{M} \times \mathcal{M}$. We now adapt the classical notion of barbs [25] to our setting: rather than communication subjects (which are hidden/unobservable names in intra-session communications), it suffices to use participant identities as observables:

Definition 4.8 (Barbs). A reachable configuration M has a barb p, written $M \downarrow_{\mathsf{p}}$, if $M \equiv (\nu\, \tilde{n})(N \mid \ell_{[\mathsf{r}]} : \wr P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{S}[\, ^{\wedge}T] \cdot \widetilde{x} \cdot \sigma \rfloor)$ where $P \equiv s_{[\mathsf{p}]}!\langle V \rangle.Q \mid R$ and $T = \mathsf{q}!\langle U \rangle.T_1$.

(AC)

$$\frac{\mathsf{p} = \mathsf{r}_1 \vee \mathsf{p} \in \mathbf{roles}(\mathsf{r}_1, h_i) \qquad \mathsf{q} = \mathsf{r}_2 \vee \mathsf{q} \in \mathbf{roles}(\mathsf{r}_2, h_i)}{\begin{array}{c}\ell_{1[\mathsf{r}_1]} : \wr s_{[\mathsf{p}]}!\langle V\rangle.P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}[\,{}^{\wedge}\mathsf{q}!\langle U\rangle.S] \cdot \widetilde{x} \cdot \sigma \rfloor \mid \ell_{2[\mathsf{r}_2]} : \wr s_{[\mathsf{q}]}?(y).Q \int \mid s_{[\mathsf{q}]} \lfloor \mathbb{S}[\,{}^{\wedge}\mathsf{p}?\langle U\rangle.T] \cdot \widetilde{x} \cdot \sigma \rfloor \mid s : (h_i \star h_o) \\ \Rrightarrow \\ \ell_{1[\mathsf{r}_1]} : \wr P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}[\mathsf{q}!\langle U\rangle.\,{}^{\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor \mid \ell_{2[\mathsf{r}_2]} : \wr Q \int \mid s_{[\mathsf{q}]} \lfloor \mathbb{S}[\mathsf{p}?\langle U\rangle.\,{}^{\wedge}T] \cdot \widetilde{x}, y \cdot \sigma[y \mapsto V] \rfloor \mid s : (h_i \circ (\mathsf{q}, \mathsf{p}, V) \star h_o)\end{array}}$$

(RAC)

$$\frac{\mathsf{p} = \mathsf{r}_1 \vee \mathsf{p} \in \mathbf{roles}(\mathsf{r}_1, h_i) \qquad \mathsf{q} = \mathsf{r}_2 \vee \mathsf{q} \in \mathbf{roles}(\mathsf{r}_2, h_i)}{\begin{array}{c}\ell_{1[\mathsf{r}_1]} : \wr P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}[\mathsf{q}!\langle U\rangle.\,{}^{\wedge}S] \cdot \widetilde{x} \cdot \sigma \rfloor \mid \ell_{2[\mathsf{r}_2]} : \wr Q \int \mid s_{[\mathsf{q}]} \lfloor \mathbb{S}[\mathsf{p}?\langle U\rangle.\,{}^{\wedge}T] \cdot \widetilde{x}, y \cdot \sigma[y \mapsto V] \rfloor \mid s : (h_i \circ (\mathsf{q}, \mathsf{p}, V) \star h_o) \\ \Lleftarrow \\ \ell_{1[\mathsf{r}_1]} : \wr s_{[\mathsf{p}]}!\langle V\rangle.P \int \mid s_{[\mathsf{p}]} \lfloor \mathbb{T}[\,{}^{\wedge}\mathsf{q}!\langle U\rangle.S] \cdot \widetilde{x} \cdot \sigma \rfloor \mid \ell_{2[\mathsf{r}_2]} : \wr s_{[\mathsf{p}]}?(y).Q \int \mid s_{[\mathsf{q}]} \lfloor \mathbb{S}[\,{}^{\wedge}\mathsf{p}?\langle U\rangle.T] \cdot \widetilde{x} \cdot \sigma \rfloor \mid s : (h_i \star h_o)\end{array}}$$

Figure 8: Atomic semantics for configurations: Forward and backward reduction ($\Rrightarrow$ and $\Lleftarrow$).

Notice that our definition of barbs is connected to the notion of stability: since in $M \downarrow_{\mathsf{p}}$ we require a monitor with empty tag, this ensures that p is not involved in an ongoing backward step. In a way, this allows us to consider just *forward barbs* (as in [1]).

We now adapt the definition of weak barbed back-and-forth (bf) bisimulation and congruence [18] in order to work with decoupled and atomic reduction semantics:

Definition 4.9. A relation $\mathfrak{R}$ is a *(weak) barbed bf simulation* if whenever $M\mathfrak{R}N$

(1) $M \downarrow_{\mathsf{p}}$ implies $N \longrightarrow^{*} \downarrow_{\mathsf{p}}$;
(2) $M \Rrightarrow M_1$ implies $N \twoheadrightarrow^{*} N_1$, with $M_1 \mathfrak{R} N_1$;
(3) $M \Lleftarrow M_1$ implies $N \rightsquigarrow^{*} N_1$, with $M_1 \mathfrak{R} N_1$.

A relation $\mathfrak{R}$ is a *(weak) barbed bisimulation* if $\mathfrak{R}$ and $\mathfrak{R}^{-1}$ are weak bf barbed simulations. The largest weak barbed bisimulation is *(weak) barbed bisimilarity*, noted $\approx$. M and N are *(weakly) barbed congruent*, written $\dot{\approx}$, if for each context $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are atomically reachable configurations, then $\mathbb{C}[M] \approx \mathbb{C}[N]$.

We now may state our second connection between decoupled and atomic reductions:

Theorem 4.10. *For any atomically reachable configuration M, we have that $M \dot{\approx} M$.*

Proof (Sketch). It suffices to show that the following relation is a bf weak bisimulation:

$$\mathfrak{R} = \{(M, N) \mid M \twoheadrightarrow^{*} N \text{ via Rule (Out)} \wedge M \rightsquigarrow^{*} N \text{ via Rule (ROut)}\}$$

The analysis uses Corollary 4.4, Theorem 4.7, the Loop and Swap Lemmas (Lemmas 4.5 and 4.6). See [21] for details. □

By observing that the set of atomic configurations is a subset of reachable configurations, this result can also be stated as full abstraction. Let f be the (injective, identity) mapping from atomically reachable configurations to reachable configurations. We then have:

Corollary 4.11 (Full Abstraction). *Let f be the injection from atomically reachable configurations to reachable configurations, and let M, N be two atomically reachable configurations. Then we have $f(M) \dot{\approx} f(N)$ if and only if $M \dot{\approx} N$.*

Proof. From Theorem 4.10 we have $M \dot{\approx} f(M)$ and $N \dot{\approx} f(N)$. The thesis follows then by transitivity of $\dot{\approx}$. □

The results above ensure that the loss of atomicity preserves the reachability of configurations yet does not make undesired configurations reachable.

4.2 Causal Consistency

Theorems 4.7 and 4.10 allow us to focus on the atomic reduction $\rightarrowtail$ for the purposes of establishing causal consistency. We adapt the approach of [7] (developed for a reversible CCS) to our higher-order session π-calculus with asynchronous communication. Causal consistency concerns traces of *transitions*:

Definition 4.12. A *transition* t is a triplet of the form $t : M \overset{\eta}{\Rightarrow} N$ where $M \rightarrowtail N$ and the *transition stamp* η is defined as follows:

- $\eta = \{\ell_1, \cdots, \ell_n\}$, if Rule (Init) or (RInit) is used;
- $\eta = \{\mathsf{p}, \mathsf{q}\}$, if Rule (AC) or (RAC) is used;
- $\eta = \{\ell, \mathsf{p}\}$, if one of Rules (Beta), (Spawn), (RBeta) or (RSpawn) is used.

Given $t : M \overset{\eta}{\Rightarrow} N$, we say M and N are its source and target (written $\mathtt{src}(t)$ and $\mathtt{trg}(t)$), respectively. A transition $t : M \overset{\eta}{\Rightarrow} N$ is *forward* if $M \Rrightarrow N$ and *backward* if $M \Lleftarrow N$. Given $t : M \overset{\eta}{\Rightarrow} N$, its *inverse*, denoted $t_{\bullet}$, is the transition $t_{\bullet} : N \overset{\eta}{\Rightarrow} M$. Two transitions are *coinitial* if they have the same source; *cofinal* if they have the same target; *composable* if the target of the first one is the source of the other. Given coinitial transitions $t_1 : M \overset{\eta_1}{\Longrightarrow} N_1$ and $t_2 : M \overset{\eta_2}{\Longrightarrow} N_2$, we define t_2/t_1 (read "t_2 after t_1") as $N_1 \overset{\eta_2}{\Longrightarrow} N_3$, i.e., the transition with stamp η_2 that starts from the target of t_1. A *trace* is a sequence of pairwise composable transitions. We let t and ρ range over transitions and traces, respectively. Notions of target, source, composability and inverse extend naturally to traces. We write ε_M to denote the empty trace with source M, and $\rho_1; \rho_2$ to denote the composition of two composable traces ρ_1 and ρ_2. Two important classes of transitions are *conflicting* and *concurrent* ones:

Definition 4.13. Two coinitial transitions $t_1 : M \overset{\eta_1}{\Longrightarrow} M_1$ and $t_2 : M \overset{\eta_2}{\Longrightarrow} M_2$ are said to be in *conflict* if $\eta_1 \cap \eta_2 \neq 0$. Two transitions are *concurrent* if they are not in conflict.

A property that a reversible semantics should enjoy is the so-called Square Lemma [7], which may be informally described as follows. Assume a configuration from which two transitions are possible: if these transitions are concurrent then the order in which they are executed does not matter, and the same configuration is reached.

Lemma 4.14 (Square). *If $t_1 : M \overset{\eta_1}{\Longrightarrow} M_1$ and $t_2 : M \overset{\eta_2}{\Longrightarrow} M_2$ are coinitial and concurrent transitions, then there exist cofinal transitions $t_2/t_1 = M_1 \overset{\eta_2}{\Longrightarrow} N$ and $t_1/t_2 = M_2 \overset{\eta_1}{\Longrightarrow} N$.*

Definition 4.15. We define $\asymp$ as the least equivalence between traces that is closed under composition and that obeys: i) $t_1; t_2/t_1 \asymp t_2; t_1/t_2$; ii) $t; t_\bullet \asymp \varepsilon_{\mathtt{src}(t)}$; iii) $t_\bullet; t \asymp \varepsilon_{\mathtt{trg}(t)}$.

Intuitively, $\asymp$ says that: (a) given two concurrent transitions, the traces obtained by swapping their execution order are equivalent; (b) a trace consisting of opposing transitions is equivalent to the empty trace. The proof of causal consistency follows that in [7], but with simpler arguments because of our simpler transition stamps. The following lemma says that, up to causal equivalence, traces can be rearranged so as to reach the maximum freedom of choice, first going only backwards, and then going only forward.

Lemma 4.16 (Rearranging). *Let ρ be a trace. There are forward traces ρ', ρ'' such that $\rho \asymp \rho'_\bullet; \rho''$.*

Proof. By lexicographic induction on the length of ρ and on the distance between the beginning of ρ and the earliest pair of opposing transitions in ρ. The analysis uses both the Loop Lemma (Lemma 4.5) and the Square Lemma (Lemma 4.14). □

If trace ρ_1 and forward trace ρ_2 start from the same configuration and end up in the same configuration, then ρ_1 may contain some "local steps", not present in ρ_2, which must be eventually reversed—otherwise there would be a difference with respect to ρ_2. Hence, ρ_1 could be *shortened* by removing such local steps and their corresponding reverse steps.

Lemma 4.17 (Shortening). *Let ρ_1, ρ_2 be coinitial and cofinal traces, with ρ_2 forward. Then, there exists a forward trace ρ'_1 of length at most that of ρ_1 such that $\rho'_1 \asymp \rho_1$.*

Proof. By induction on the length of ρ_1, using Square and Rearranging Lemmas (Lemmas 4.14, 4.16). The proof uses the forward trace ρ_2 as guideline for shortening ρ_1 into a forward trace, relying on the fact that ρ_1, ρ_2 share the same source and target. □

We may now state our main result:

Theorem 4.18 (Causal consistency). *Let ρ_1 and ρ_2 be coinitial traces, then $\rho_1 \asymp \rho_2$ if and only if ρ_1 and ρ_2 are cofinal.*

Proof. The 'if' direction follows by definition of $\asymp$ and trace composition. The 'only if' direction uses Square, Rearranging and Shortening Lemmas (Lemmas 4.14, 4.16, 4.17). □

4.3 Connecting Reversible Choreographies and Reversible Configurations

We now relate choreographies and configurations to connect the two levels of abstraction for reversible global protocols. For convenience, we focus on first-order global types (i.e., without abstraction passing), relying on a simple characterization of the *well-formed processes* that implement a given local type. We write $P \bowtie_x T$ to denote that P implements the local type T along variable x—see [21] for a definition. We may then define the configurations that implement a global type with history. First, an auxiliary definition:

$$(\text{Sw1})\ \frac{\{\mathsf{p}_1, \mathsf{q}_1\} \# \{\mathsf{p}_2, \mathsf{q}_2\}}{\mathsf{p}_1 \to \mathsf{q}_1 : \langle U_1 \rangle.(\mathsf{p}_2 \to \mathsf{q}_2 : \langle U_2 \rangle.G) \approx_{\mathsf{sw}} \mathsf{p}_2 \to \mathsf{q}_2 : \langle U_2 \rangle.(\mathsf{p}_1 \to \mathsf{q}_1 : \langle U_1 \rangle.G)}$$

Figure 9: Swapping on global types G. We write $A \# B$ if A and B are disjoint sets.

Definition 4.19. We say the global type with history H is *reachable* if it can be obtained from a global type G via a sequence of $\hookrightarrow$ and $\rightharpoonup$ transitions (cf. Fig. 3).

Definition 4.20. Let G be a global type, with $\mathsf{pa}(G) = \{\mathsf{p}_1, \cdots, \mathsf{p}_n\}$. We say that configuration M *initially implements* G if we have

$$M \equiv (\nu\, s)\Big(\prod_{i \in \{1, \cdots, n\}} \ell_{i\,[\mathsf{p}_i]} : \wr P_i\{s_{[\mathsf{p}_i]}/x_i\} \int \mid s_{[\mathsf{p}_i]} \llcorner \,{}^{\wedge}G{\downarrow}_{\mathsf{p}_i} \cdot x_i \cdot \sigma_i \lrcorner \mid s : (\epsilon \star \epsilon)\Big)$$

with $P_i \bowtie_{x_i} G{\downarrow}_{\mathsf{p}_i}$, for all $i \in \{1, \cdots, n\}$, for some stores $\sigma_1, \ldots, \sigma_n$. A configuration N *implements* the global type with history H, written $N \bowtie \mathsf{H}$, if there exist M, G such that (i) H is reachable from G, (ii) M initially implements G, and (iii) N is reachable from M.

The last ingredient required is a *swapping relation* over global types, denoted $\approx_{\mathsf{sw}}$, which enables behavior-preserving transformations among causally independent communications.

Definition 4.21 (Swapping). We define $\approx_{\mathsf{sw}}$ as the smallest congruence on G that satisfies the rules in Fig. 9 (where we omit the symmetric of (Sw1)). We extend $\approx_{\mathsf{sw}}$ to global types with history H as follows: $\mathbb{G}[\,{}^{\wedge}G_1] \approx_{\mathsf{sw}} \mathbb{G}'[\,{}^{\wedge}G_2]$ if $\mathbb{G}[\mathsf{end}] \approx_{\mathsf{sw}} \mathbb{G}'[\mathsf{end}]$ and $G_1 \approx_{\mathsf{sw}} G_2$.

We may now relate (i) transitions in the semantics of (high-level) global types (with history) with (ii) reductions in the semantics of their (low-level) process implementations. We write $M \rightsquigarrow^j M'$ to denote a sequence of $j \geq 0$ reduction steps (if $j = 0$ then $M = M'$).

Theorem 4.22. *Let H be a reachable, first-order global type with history (cf. § 2.1.2).*

a) If $M \bowtie \mathsf{H}$ and $\mathsf{H} \hookrightarrow \mathsf{H}'$ then $M \twoheadrightarrow M'$ and $M' \bowtie \mathsf{H}'$, for some M'. If $M \bowtie \mathsf{H}$ and $\mathsf{H} \rightharpoonup \mathsf{H}'$ then $M \rightsquigarrow^j M'$ (with $j = 1$ or $j = 2$) and $M' \bowtie \mathsf{H}'$, for some M'.
b) Suppose $M \bowtie \mathsf{H}$. For all configurations N_i such that $M \twoheadrightarrow N_i$ there exist $\mathsf{H}_i, \mathsf{H}'_i, \mathsf{H}''$, and M', such that $\mathsf{H} \approx_{\mathsf{sw}} \mathsf{H}_i \hookrightarrow \mathsf{H}'_i$, $N_i \bowtie \mathsf{H}'_i$, $N_i \twoheadrightarrow^* M'$, $\mathsf{H}'_i \hookrightarrow^* \mathsf{H}''$, and $M' \bowtie \mathsf{H}''$ (and similarly for $\rightsquigarrow$, $\rightharpoonup$).

Proof. By induction on the transitions/reductions. See [21] for details. □

Theorem 4.22 captures an asymmetry between global types and configurations. While Part (a) shows that a configuration closely mimics the behavior of its associated global type, Part (b) shows that a configuration may have more immediate behaviors than those described its associated global type: this is because a configuration may include several independent (and immediate) reductions (N_i above), which are matched by the global type only up to swapping.

Summing up, we have that Theorem 4.18 ensures that reversibility in the atomic semantics is causally consistent. Theorem 4.10 transfers this result to decoupled semantics; since by Theorem 4.22 decoupled semantics defines a sound local implementation, we conclude that reversibility for global types is also causally consistent.

5 EXTENSIONS

We briefly discuss extensions to our framework (name passing, interleaved protocols, types, labeled choices) that would allow us to relate it with known typed frameworks for monitored networks (without reversibility) based on multiparty sessions [3].

Name passing. As already mentioned, our framework does not include name passing, which is known to be representable, in a fully abstract way, using name abstractions [15]. Primitive support for name passing is not difficult, but would entail notational burden.

Interleaved sessions. The framework presented here handlers process implementations for a single multiparty session. To support multiple, interleaved sessions we could adapt the approach we developed in [20, 22], which allows reversing different sessions simply by recording the sequence in which they are established.

Types. We do not foresee difficulties to strengthen Theorem 4.22 to cover global types with higher-order values. Such an extension would entail replacing $P \bowtie_x T$ with a type system for multiparty, higher-order sessions, which could be obtained by adapting known type systems for binary, higher-order sessions [15, 23].

Labeled Choices. We have not considered global types of the form $\mathsf{p} \to \mathsf{q} : \{l_i : G_i\}_{i \in I}$ which, given a finite index set I and pairwise different *labels* l_i, specify that: p may choose label l_i, communicate this selection to q, and then continue as choreography G_i.

Accommodating standard forms of labeled choices is straightforward but entails additional technical machinery, which we have omitted here for the sake of readability—see [21] for full details. Main required additions are in global/local types, as well as in processes and their reduction semantics, for we use running processes of the form $\ell_{[\mathsf{p}]} : \wr \mathtt{C} \,;\, P \int$, where $\mathtt{C}$ is a list of processes that allows us to record/reinstate the discarded labeled alternatives. Each of the reduction semantics given ($\twoheadrightarrow$, $\rightsquigarrow$, $\Rrightarrow$, and $\Lleftarrow$) extends accordingly, with expected modifications in auxiliary notions (e.g., type contexts), and using additional reduction rules. Proofs of our main results require minimal and/or unsurprising modifications.

6 RELATED WORK

Reversibility in concurrency has received much attention recently. An overview of the literature on the intersection between reversibility and behavioral contracts/types appears in [22, § 7]. Within this research line, the works most related to ours are [4, 8, 27].

Tiezzi and Yoshida [27] study the cost of implementing different ways of reversing binary and multiparty sessions; since they work in a *synchronous* setting, these alternatives are simpler or incomparable to our asynchronous, decoupled rollback. Dezani-Ciancaglini and Giannini [8] develop typed multiparty sessions with *checkpoints*, points in the global protocol to which computation may return. While our reversible actions are embedded in/guaranteed by the semantics, rollbacks in [8] should specify the name of the checkpoint to which computation should revert. Defining reversibility in [8] requires modifying both processes and types. In contrast, we consider standard untyped processes and local types (with cursors) as monitors. While we show causal consistency with a direct proof, in [8] causal consistency follows indirectly, as a consequence of typing. Reversibility in our model is fine-grained in that we allow reversible actions concerning exactly two of the protocol participants; in [8] when a checkpoint is taken, also parties not related with that choice are forced to return to a checkpoint. Building upon [8], Castellani et al. [4] have recently proposed a synchronous calculus of concurrent, reversible multiparty sessions, which is more expressive than standard calculi for multiparty sessions. This new calculus is equipped with a type system that ensures fidelity, forward and backward progress, as well as causal consistency, which in [4] is distinguished from the stronger *full reversibility* that we establish here for our asynchronous model, i.e., the ability to get back to an exact past state by deleting all its effects.

7 CONCLUDING REMARKS

We presented a process framework of reversible, multiparty asynchronous communication, built upon session-based concurrency. As illustrated in § 3, the distinguishing features of our framework (decoupled rollbacks and abstraction passing, including delegation) endow it with substantial expressiveness, improving on prior works.

Our processes/configurations are untyped, but their (reversible) behavior is governed by monitors derived from local (session) types. In our view, our monitored approach to reversibility is particularly appropriate for specifying and reasoning about systems with components whose behavior may not be statically analyzed (e.g., legacy components or services available as black-boxes). A monitored approach is general enough to support also the analysis of reversible systems that combine typed and untyped components.

We proved that our reversible semantics is *causally consistent*, which ensures that reversing a computation leads to a state that could have been reached by performing only forward steps. The proof is challenging (and, in our view, also interesting), as we must resort to an alternative atomic semantics for rollbacks (Fig. 8). We then connected reversibility at the level of process/configurations with that at the level of global types, therefore linking the operational and declarative specifications typical of choreographic approaches to correct communication-centric software systems.

In future work, we plan to extend our framework with so-called *reversibility modes* [22], which implement *controlled reversibility* [17] by specifying how many times a particular protocol step can be reversed—zero, one, or infinite times (currently all actions can be reversed infinite times). In a related vein (and following [4, 8]), we plan to explore variants of our model in which certain protocol branches are "forgotten" after they have been reversed. This modification is delicate, because it would weaken the notion of causal consistency; in particular, relating (i) states reached via a rollback and (ii) previous past states would require special care—we believe that a notion of subtyping [9, 10] could ease this task.

ACKNOWLEDGMENTS

We are grateful to Mariangiola Dezani-Ciancaglini and to the anonymous reviewers for their useful remarks and suggestions, which led to substantial improvements.

This work was partially supported by EU COST through actions IC1201 (Behavioral Types for Reliable Large-Scale Software Systems), IC1402 (Runtime Verification beyond Monitoring), and IC1405 (Reversible Computation – Extending Horizons of Computing). Pérez is also affiliated to the NOVA Laboratory for Computer Science and Informatics (supported by FCT grant NOVA LINCS PEst/UID/CEC/04516/2013), Universidade Nova de Lisboa, Portugal.

REFERENCES

[1] C. Aubert and I. Cristescu. Contextual equivalences in configuration structures and reversibility. *J. Log. Algebr. Meth. Program.*, 86(1):77–106, 2017.

[2] F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro. Reversible client/server interactions. *Formal Asp. Comput.*, 28(4):697–722, 2016.

[3] L. Bocchi, T. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. *Theor. Comput. Sci.*, 669:33–58, 2017.

[4] I. Castellani, M. Dezani-Ciancaglini, and P. Giannini. Concurrent reversible sessions. In *Proc. of CONCUR'17*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[5] M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida. A gentle introduction to multiparty asynchronous session types. In M. Bernardo and E. B. Johnsen, editors, *Formal Methods for Multicore Programming*, volume 9104 of *LNCS*, pages 146–178. Springer, 2015.

[6] V. Danos and J. Krivine. Reversible communicating systems. In P. Gardner and N. Yoshida, editors, *CONCUR 2004*, LNCS, pages 292–307. Springer, 2004.

[7] V. Danos and J. Krivine. Transactions in RCCS. In M. Abadi and L. de Alfaro, editors, *CONCUR 2005*, pages 398–412, 2005.

[8] M. Dezani-Ciancaglini and P. Giannini. Reversible multiparty sessions with checkpoints. In D. Gebler and K. Peters, editors, *EXPRESS/SOS 2016*, volume 222 of *EPTCS*, pages 60–74, 2016.

[9] S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[10] S. J. Gay. Subtyping supports safe session substitution. In S. Lindley, C. McBride, P. W. Trinder, and D. Sannella, editors, *A List of Successes That Can Change the World*, volume 9600 of *LNCS*, pages 95–108. Springer, 2016.

[11] M. Hennessy. *A Distributed Pi-Calculus.* CUP, 2007.

[12] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[13] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

[14] 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.

[15] D. Kouzapas, J. A. Pérez, and N. Yoshida. On the relative expressiveness of higher-order session processes. In P. Thiemann, editor, *ESOP 2016*, volume 9632 of *LNCS*, pages 446–475. Springer, 2016.

[16] D. Kouzapas, N. Yoshida, R. Hu, and K. Honda. On asynchronous eventful session semantics. *Mathematical Structures in Computer Science*, 26(2):303–364, 2016.

[17] I. Lanese, C. A. Mezzina, and J.-B. Stefani. Controlled reversibility and compensations. In R. Glück and T. Yokoyama, editors, *Reversible Computation, RC 2012. Revised Papers*, volume 7581 of *LNCS*, pages 233–240. Springer, 2013.

[18] I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversibility in the higher-order π-calculus. *Theor. Comput. Sci.*, 625:25–84, 2016.

[19] I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-consistent reversibility. *Bulletin of the EATCS*, 114, 2014.

[20] C. A. Mezzina and J. A. Pérez. Reversible sessions using monitors. In D. A. Orchard and N. Yoshida, editors, *PLACES 2016*, volume 211 of *EPTCS*, pages 56–64, 2016.

[21] C. A. Mezzina and J. A. Pérez. Causally consistent reversible choreographies. *CoRR*, abs/1703.06021, 2017.

[22] C. A. Mezzina and J. A. Pérez. Reversibility in session-based concurrency: A fresh look. *J. Log. Algebr. Meth. Program.*, 90:2–30, 2017.

[23] D. Mostrous and N. Yoshida. Two session typing systems for higher-order mobile processes. In S. R. D. Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 321–335. Springer, 2007.

[24] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms.* PhD thesis, University of Edinburgh, 1992.

[25] D. Sangiorgi and D. Walker. On barbed equivalences in pi-calculus. In K. G. Larsen and M. Nielsen, editors, *CONCUR 2001*, volume 2154 of *LNCS*, pages 292–304. Springer, 2001.

[26] F. Tiezzi and N. Yoshida. Reversible session-based pi-calculus. *J. Log. Algebr. Meth. Program.*, 84(5):684–707, 2015.

[27] F. Tiezzi and N. Yoshida. Reversing single sessions. In S. J. Devitt and I. Lanese, editors, *Reversible Computation - 8th International Conference, RC 2016*, volume 9720 of *LNCS*, pages 52–69. Springer, 2016.