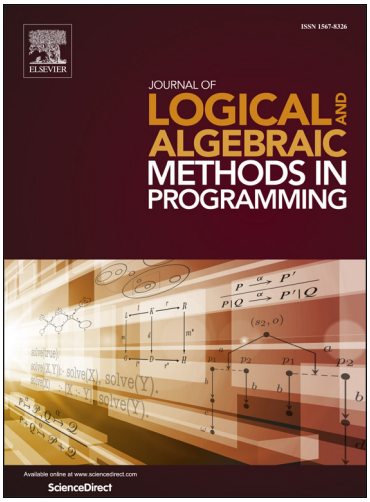
Accepted Manuscript



Reversibility in session-based concurrency: A fresh look

Claudio Antares Mezzina, Jorge A. Pérez

PII: S2352-2208(16)30063-3
DOI: http://dx.doi.org/10.1016/j.jlamp.2017.03.003
Reference: JLAMP 174

To appear in: *Journal of Logical and Algebraic Methods in Programming*

Received date: 25 July 2016
Revised date: 2 March 2017
Accepted date: 15 March 2017

Please cite this article in press as: C.A. Mezzina, J.A. Pérez, Reversibility in session-based concurrency: A fresh look, *J. Log. Algebraic Methods Program.* (2017), http://dx.doi.org/10.1016/j.jlamp.2017.03.003

Reversibility in Session-Based Concurrency: A Fresh Look

Claudio Antares Mezzina[a], Jorge A. Pérez[b]

[a] *IMT, School for Advanced Studies Lucca, Italy*
[b] *University of Groningen & CWI, Amsterdam, The Netherlands*

Abstract

Much research has studied foundations for correct and reliable *communication-centric software systems*. A salient approach to correctness uses verification based on *session types* to enforce structured communications; a recent approach to reliability uses *reversible actions* as a way of reacting to unanticipated events or failures. In this paper, we develop a simple observation: the semantic machinery required to define asynchronous (queue-based), monitored communications can also support reversible protocols. We propose a framework of session communication in which monitors support reversibility of (untyped) processes. Main novelty in our approach are *session types with present and past*, which allow us to streamline the semantics of reversible actions. We prove that reversibility in our framework is causally consistent, and define ways of using monitors to control reversible actions.

Keywords: Concurrency, Reversible Computation, Behavioral Types, Process Calculi

1. Introduction

Much research has studied foundations for reliable *communication-centric* software systems, cf. [1, 2, 3, 4]. Our interest is in programming models that support the analysis of message-passing programs building on foundations offered by core calculi for concurrency. While early such models focused on (static) verification of protocol correctness, as enforced by properties such as safety, fidelity, and progress (deadlock-freedom), extensions of the basic models with external mechanisms have been proposed to enforce protocol correctness even in the presence of unanticipated events, such as failures or new requirements. Such mechanisms include, e.g., exceptions, interruptions and compensations [5, 6, 7], adaptation [8], and monitors [9]. They also include *reversible semantics* [10, 11, 12], the main topic of this paper.

Comprehensive approaches to correctness and reliability, which enforce both kinds of requirements, seem indispensable in the principled design of communication-centric software systems. As these systems are typically built using heterogeneous services whose provenance/correctness cannot always be certified in advance, static validation techniques (such as type systems) fall short. Correctness must then be guaranteed by mechanisms that (dynamically) inspect the (visible) behavior of interacting services and take action if they deviate from prescribed communication protocols.

In this work, we aim at uniform approaches to correct and reliable communicating systems. We address the interplay between concurrent models of reversible computation [13, 14] and *session-based concurrency* [1]. In reversible models of concurrency the usual *forward* semantics is coupled with a *backward* semantics that enable to "undo" process actions. In this setting, a central correctness criterion is *causal consistency*, which ensures that a computational step is reversed only when all its causes (if any) have already been reversed. In this way, causally consistent reversibility leads to states of the system that could have been reached by performing forward steps only. In session-based concurrency, concurrent interactions between processes can be conceptually divided in two phases: first, processes requesting/offering protocols seek a compatible partner; subsequently, the (compatible) partners establish a session and interact following the stipulated protocols. Session protocols are *resource-aware*: the first phase defines non-deterministic interactions along unrestricted names; the second one uses deterministic interaction sequences along linear names.

Following the seminal work of Danos and Krivine [13], a key technical device in formalizing reversible semantics are *memories*: these are run-time constructs that make it possible to revert actions. Memories are the bulk of a reversible model; their definition and maintenance requires care, as demonstrated by Tiezzi and Yoshida [10, 12], who were the first to adapt known approaches to reversible semantics [13, 15] into session-based concurrency. Using different kinds of memories (recording events for actions, choices, and forking), their work shows that the standard (untyped) reduction semantics for the session π-calculus satisfies causal consistency.

While insightful, the route to reversibility in session-based concurrency taken in [12] is somewhat unsatisfactory, for session types do not play any role in the underlying (reversible) semantics nor in the proof of causal consistency. If one considers that session types offer a compact abstraction of the communication behavior of the channels/names in a process, then it is natural to think of them as auxiliary mechanisms in the definition of forward and backward reduction semantics. That is, the communication structures given by session types already contain valuable information for enabling causally consistent reversible semantics. If one further considers that once a session is established processes behave deterministically, as dictated by their session protocols, then it is natural to expect that reversibility and causal consistency in session-based concurrency arise more orderly than in untyped models of concurrency, such as those in, e.g., [13, 14].

Following these considerations, in this paper we investigate to what extent session types can streamline the definition of reversible, causally consistent semantics for interacting processes. Our main discovery is that external mechanisms typically used to support asynchronous (queue-based) and monitored semantics in session-based concurrency (cf. [16, 17, 18, 19, 20]) can also effectively support the definition of reversible sessions. In such semantics, monitors are run-time devices that register the current state of the session protocols implemented in and executed by a process. We explore a fresh approach to reversibility by using *monitors as memories*. The key idea is simple: we exploit the type information in monitors to define the reversible semantics of session processes. Since these types enable and guide process behavior, we may uniformly define forward and backward reductions by carefully controlling such types and their associated run-time information.

Contributions. The main contributions of this paper are the following:

- We define a fresh approach to reversible semantics in session-based concurrency by exploiting monitors as uniform memories that enable and support backward communication steps.
- We show that the reversible semantics in our approach is *causally consistent*, directly exploiting the disciplined interaction scenario naturally induced by session-based concurrency.
- We show that our approach can be extended to enforce *controlled reversibility* by using enriched session types (rather than explicit process constructs) for guiding process behavior.

To highlight the merits of our approach, we rely on a core process model without recursion nor asynchrony, which are important in modeling but largely orthogonal to our reversible semantics. These and other features can be accommodated in our approach while retaining its essence.

In our view, the use of monitors for defining reversible semantics has at least two significant implications. First, it is encouraging to discover that monitor-based semantics—introduced in [16, 17, 18] for asynchronous communications with events and used in [19, 20] to define run-time adaptation—may also inform the semantics of reversible protocols. Monitors have also been used for enforcing security properties (such as information flow [21, 22]) and for assigning blame to deviant session processes [9]. Therefore, monitor-based semantics encompass an array of seemingly distinct concerns in structured communications. Second, we see our work as a first step towards validation techniques for communication and reversibility based on run-time verification. Session frameworks with run-time verification have been developed in, e.g., [23, 6]. As these works do not support reversibility, our work may lead to enhancements for their dynamic verification techniques. Indeed, since the framework in [23, 6] introduces constructs for delimiting interruptible sub-protocols, one could re-use such constructs (and their underlying semantic mechanisms, such as type memories) to safely enable reversible actions within distributed protocols.

Outline. This paper is structured as follows. In the following section we motivate further the key ideas of our development. Section 3 presents the syntax and operational semantics of our process model with sessions and reversibility, and illustrates it via a running example. The main property of our model, *causal consistency* (Theorem 4.1), is established in Section 4. Section 5 discusses an extension of our framework to enforce controlled reversible actions. Other extensions and enhancements (including asynchrony, delegation, and recursion) are discussed in Section 6. Section 7 elaborates on related works. Section 8 closes the paper by collecting some concluding remarks and highlighting some directions for future work. The appendix (Appendix A) collects omitted proofs.

This paper is a revised and substantially extended version of the workshop paper [24] and the short communication [25]: here we offer full technical details, new examples, and an extended account of related works. In particular, Section 3 has been streamlined and extended to handle reversible labeled choices, not supported in [24]. Moreover, the content of Sections 4, 5, and 6 is new to this paper.

2. Overview

Our approach can be seen as an optimization of the reduction semantics for sessions. In this section, we illustrate it via approximate reduction rules, which omit unimportant notational details.

In reduction semantics for session-based concurrency, such as those in [16, 17, 18], the reduction rule for intra-session communication relies on *monitors* containing session types ($S, T, \ldots$) and message queues ($\tilde{h}_1, \tilde{h}_2, \ldots$):

$$\overline{s}\langle v\rangle.P \parallel \overline{s}\lfloor !U.S_1 \cdot \tilde{h}_1\rfloor \parallel s(x).Q \parallel s\lfloor ?U.S_2 \cdot \tilde{h}_2\rfloor \longrightarrow P \parallel \overline{s}\lfloor S_1 \cdot \tilde{h}_1\rfloor \parallel Q \parallel s\lfloor S_2 \cdot \tilde{h}_2, v\rfloor \quad (1)$$

In (1), processes $\overline{s}\langle v\rangle.P$ and $s(x).Q$ denote output and input along *session endpoints* $\overline{s}$ and s, respectively. (Notice that $\overline{s}$ and s are *dual* endpoints.) Thus, the reduction rule above concerns four elements that interact under parallel composition (denoted by '$\parallel$'): input and output processes and their associated monitors. Also, the session type $!U.S_1$ (resp. $?U.S_2$) abstracts a protocol that decrees the output (resp. input) of a value of type U that is followed by a protocol represented by session type S_1 (resp. S_2). Given an endpoint s, process $s\lfloor S \cdot \tilde{h}\rfloor$ is a *monitor*, where S and $\tilde{h}$ are the session type and *message queue* for s, respectively. Observe how the communicated value v is placed in the queue of endpoint s; a subsequent synchronization between the monitor and Q (not shown) should lead to the expected resulting process (i.e., $Q\{v/x\}$).

In the approach of [16, 17, 18], reduction rules use session types to enable communication actions: a reduction step can only occur if the actions (in the processes) correspond to the intended protocols (in the monitor types). After the synchronization is realized, portions of both processes and monitor types are consumed. This way, reduction steps have an effect on both processes and the session types in the monitors. Our approach consists in keeping, rather than consuming, these monitor types. For this to work, we need to distinguish the part of the protocol that has been already executed (its past), from the protocol that still needs to execute (its present). We thus introduce session types with *present and past*: the type $S\,\hat{}\,T$ says that the communication actions abstracted by S are past protocol actions, whereas actions in T are present steps. Using this insight, we may refine (1) as follows:

$$\begin{aligned}\overline{s}\langle v\rangle.P \parallel \overline{s}\lfloor T\,\hat{}\,!U.S_1 \cdot \tilde{h}_1\rfloor \parallel s(x).Q \parallel s\lfloor T'\,\hat{}\,?U.S_2 \cdot \tilde{h}_2\rfloor \twoheadrightarrow \\ P \parallel \overline{s}\lfloor T.!U\,\hat{}\,S_1 \cdot \tilde{h}_1\rfloor \parallel Q \parallel s\lfloor T'.?U\,\hat{}\,S_2 \cdot \tilde{h}_2, v\rfloor \end{aligned} \quad (2)$$

This is a *forward* reduction rule, denoted in the sequel by $\twoheadrightarrow$. In monitors $\overline{s}\lfloor T\,\hat{}\,!U.S_1 \cdot \tilde{h}_1\rfloor$ and $s\lfloor T'\,\hat{}\,?U.S_2 \cdot \tilde{h}_2\rfloor$, we write T and T' to denote past protocol actions. The monitors rely on type-checking to enable forward and backward computations; they may also implement asynchronous communication. Observe that we use the cursor $\hat{}$ to preserve output and input protocol actions (noted $!U$ and $?U$, respectively). Notice also that considering this enhanced form of session types in monitors does not affect the syntax or semantics of processes.

Based on (2), we may state a corresponding *backward* reduction rule, denoted in the sequel by $\rightsquigarrow$, which reverts the synchronization at the level of processes, types, and message queues:

$$\begin{aligned}P \parallel \overline{s}\lfloor T_1.!U\,\hat{}\,S_1 \cdot \tilde{h}_1\rfloor \parallel Q \parallel s\lfloor T_2.?U\,\hat{}\,S_2 \cdot \tilde{h}_2, v\rfloor \rightsquigarrow \\ \overline{s}\langle v\rangle.P \parallel \overline{s}\lfloor T_1\,\hat{}\,!U.S_1 \cdot \tilde{h}_1\rfloor \parallel s(x).Q \parallel s\lfloor T_2\,\hat{}\,?U.S_2 \cdot \tilde{h}_2\rfloor \end{aligned} \quad (3)$$

We stress that rules (2) and (3) are an approximate illustration of our approach. For instance, in our reversible operational semantics, the forward reduction rule for communication records also the input parameter x, which is necessary in the reversible rule to reinstate the original input prefix.

One main technical contribution of this work is a core framework for session communication and reversibility whose monitored semantics follows the spirit of rules (2) and (3). In addition to input-output behaviors, we consider *labeled choices*, typical of session-based concurrency, which are expressed at the level of processes by branching and selection constructs (external and internal choices, respectively): process $s \triangleright \{l_1{:}P_1, \cdots, l_n{:}P_n\}$ offers n alternative behaviors (branches) $P_1, \ldots, P_n$, identified by pairwise distinct labels $l_1, \ldots, l_n$; this process is expected to synchronize with a process $s \triangleleft l_j.Q$ (with $j \in 1..n$), which performs a selection. Branching and selection processes are governed by session types $\&\{l_1{:}S_1, \cdots, l_n{:}S_n\}$ and $\oplus\{l_1{:}S_1, \cdots, l_n{:}S_n\}$, respectively.

We illustrate the usual (forward) reduction rule for labeled choice (adapted to a monitored semantics) by means of an example:

$$\begin{array}{c} \bar{s} \triangleleft l_1.P \parallel \bar{s}\lfloor \oplus\{l_1{:}S_1\,,\, l_2{:}S_2\} \cdot \tilde{h}_1 \rfloor \parallel s \triangleright \{l_1 : Q_1\,;\, l_2 : Q_2\} \parallel s\lfloor \&\{l_1{:}T_1\,;\, l_2{:}T_2\} \cdot \tilde{h}_2 \rfloor \longrightarrow \\ P \parallel \bar{s}\lfloor S_1 \cdot \tilde{h}_1 \rfloor \parallel Q_1 \parallel s\lfloor T_1 \cdot \tilde{h}_2, l_1 \rfloor \end{array} \quad (4)$$

In (4), process $\bar{s} \triangleleft l_1.P$ selects the branch labeled by l_1 on session s, while process $s \triangleright \{l_1 : Q_1\,;\, l_2 : Q_2\}$ waits for a synchronization on s that selects either l_1 or l_2. One effect of such a synchronization is that the two processes evolve into P and Q_1, respectively, and that the message involved (i.e., the label l_1) is stored into the queue of the process on the right. Another effect of the synchronization is both label l_2 and process P_2 are discarded. In this case, our approach consists in using the *static choice contexts* and *type contexts*, which we sketch in the following rule:

$$\begin{array}{c} \bar{s} \triangleleft l_1.P \parallel \bar{s}\lfloor \oplus\{l_1{:}S_1\,,\, l_2{:}S_2\} \cdot \tilde{h}_1 \rfloor \parallel s \triangleright \{l_1 : Q_1\,;\, l_2 : Q_2\} \parallel s\lfloor \&\{l_1{:}T_1\,;\, l_2{:}T_2\} \cdot \tilde{h}_2 \rfloor \twoheadrightarrow \\ P \parallel \bar{s}\lfloor \oplus\{l_1{:}\hat{}\, S_1\,;\, l_2{:}S_2\} \cdot \tilde{h}_1 \rfloor \parallel s \triangleright \{l_1 : Q_1\,;\, l_2 : \langle Q_2 \rangle\} \parallel s\lfloor \&\{l_1{:}\hat{}\, T_1\,;\, l_2{:}T_2\} \cdot \tilde{h}_2, l_1 \rfloor \end{array} \quad (5)$$

Above, process $s \triangleright \{l_1 : Q_1\,;\, l_2 : \langle Q_2 \rangle\}$ uses a static choice context to indicate that the current running process is Q_1, and that process Q_2 has been discarded by a previous choice. That is, Q_2 in $\langle Q_2 \rangle$ can be seen as living in an inactive context in which it cannot execute; keeping it is useful just to maintain the state of the process before the choice. The effect of the selection is recorded also at the level of session types, types $\&\{l_1{:}\hat{}\, T_1\,;\, l_2{:}T_2\}$ and $\oplus\{l_1{:}\hat{}\, S_1\,;\, l_2{:}S_2\}$, respectively; this way, we ensure consistent information between processes and types, useful to rebuild them in a reversible reduction. Since now the cursor $\hat{}$ is used within a particular branch of types for labeled choice, the session types with present and past adopt a more general structure, which is captured by type contexts.

Using these new elements, in the backward rule for labeled choice both processes within inactive contexts and its associated session types are restored:

$$\begin{array}{rc} & P \parallel \bar{s}\lfloor \oplus\{l_1{:}\hat{}\, S_1\,;\, l_2{:}S_2\} \cdot \tilde{h}_1 \rfloor \parallel \{l_1 : Q_1\,;\, l_2 : \langle Q_2 \rangle\} \parallel s\lfloor \&\{l_1{:}\hat{}\, T_1\,;\, l_2{:}T_2\} \cdot \tilde{h}_2, l_1 \rfloor \\ \rightsquigarrow & \bar{s} \triangleleft l_1.P \parallel \bar{s}\lfloor \oplus\{l_1{:}S_1\,,\, l_2{:}S_2\} \cdot \tilde{h}_1 \rfloor \parallel s \triangleright \{l_1 : Q_1\,;\, l_2 : Q_2\} \parallel s\lfloor \&\{l_1{:}T_1\,;\, l_2{:}T_2\} \cdot \tilde{h}_2 \rfloor \end{array} \quad (6)$$

In our framework, session processes occur within *configurations*, denoted $\langle P \cdot \sigma \cdot \widetilde{u} \rangle_\delta$, where P is a session process, σ denotes its state, $\widetilde{u}$ is a list that collects information on the actions already performed, and δ is a unique identifier. Notice that while state conveniently implements substitutions, monitors handle both communication and reversibility, using session types with present and past, as motivated above. We support session establishment and the consistent use of sent values and open variables in the state (cf. v and x in (2) and (3)). Our semantics enjoys the so-called *Loop Lemma* [13, 10, 12] (Lemma 3.2, Page 17), which offers a basic consistency guarantee for the interplay of forward and backward actions, and *causal consistency* (Theorem 4.1, Page 22), which characterizes admissible rollbacks (i.e., sequences of backward steps) which are consistent and flexible.

3. Syntax and Semantics

Here we present our framework of session processes with monitored, reversible semantics. We introduce its syntax and semantics, illustrate it via an example, and establish its basic properties.

We assume the following denumerable infinite mutually disjoint sets: the set $\mathcal{S}$ of *session names* (or *endpoints*), ranged over by $s, r, \ldots$; the set $\mathcal{C}$ of *channels*, ranged over by $a, b, \ldots$; the set of *variables* $\mathcal{X}$, ranged over by $x, y, \ldots$. The set $\mathcal{N} = \mathcal{S} \cup \mathcal{C}$ is called the set of *names*; we use $m, n, \ldots$ to range over $\mathcal{N}$. We also assume a set of *labels* $\mathcal{L}$, ranged over $l, l', \ldots$, which will be used to denote labeled choices. We assume a total bijection over $\mathcal{S}$, noted $\overline{\cdot}$, relating endpoints with their duals such that, for any $s \in \mathcal{S}$, we have $\overline{s} \neq s$ and $\overline{\overline{s}} = s$. We use k, k' (and their duals) to range over $\mathcal{S} \cup \mathcal{X}$. Moreover, we use $\widetilde{o}$ to denote a finite sequence of objects (e.g., names) $o_1, \ldots, o_n$, and ε to denote the empty sequence. We sometimes treat sequences as a set or as an ordered list. To define and handle configurations (see below), we shall require *named sequences*: given a sequence of session names $\widetilde{s}$ and a $\kappa \in \mathcal{N}$, we say that $\kappa : \widetilde{s}$ is a named sequence. We write δ, $\delta', \ldots$ to denote finite, possibly empty named sequences of session names. Given $\delta_1 = \kappa_1 : \widetilde{s}_1$ and $\delta_2 = \kappa_2 : \widetilde{s}_2$, we write $\delta_1 \cap \delta_2$ to stand for $\{\kappa_1, \widetilde{s}_1\} \cap \{\kappa_2, \widetilde{s}_2\}$.

3.1. Syntax

Main ingredients in our approach are configurations, processes, and protocol types, whose syntax is given in Figure 1. We explain these ingredients next.

Configurations and Processes. The syntax of configurations includes the empty configuration $\mathbf{0}$, the *running* process $\langle P \cdot \sigma \cdot \widetilde{u} \rangle_\delta$, a monitor $s\lfloor S \cdot \widetilde{x} \rfloor$, the name restriction $\nu n.M$, and parallel composition $M \parallel N$. Running processes and monitors are central to our approach:

- A running process $\langle P \cdot \sigma \cdot \widetilde{u} \rangle_\delta$ is univocally identified by δ, the named sequence of session endpoints occurring in P. The local store σ is a list of pairs of the form $\{x, \widetilde{v}\}$ (see Def. 3.5); the list $\widetilde{u}$ collects the subjects of actions already performed by P.
- A monitor $s\lfloor S \cdot \widetilde{x} \rfloor$ is identified by the session name s, contains a session type S that describes the structured behavior of s and a list $\widetilde{x}$ containing all the variables used by the process. As we will see, for each monitor $s\lfloor S \cdot \widetilde{x_1} \rfloor$ there will be a dual monitor $\overline{s}\lfloor T \cdot \widetilde{x_2} \rfloor$.

$$
\begin{array}{lrl}
(\textit{names}) & n, m ::= & a, b \mid s, \overline{s} \\
(\textit{subjects}) & u ::= & n \mid k \\
(\textit{expressions}) & e ::= & a \mid v \mid l_i \mid \mathtt{op}(e_1, \ldots, e_n) \\
(\textit{configurations}) & M, N ::= & \mathbf{0} \mid \langle P \cdot \sigma \cdot \widetilde{u} \rangle_{\delta} \mid s\lfloor H \cdot \widetilde{e} \rfloor \mid \nu n.M \mid M \parallel N \\
(\textit{processes}) & P, Q ::= & u(x : S).P \mid u\langle x : S \rangle.P \mid k\langle e \rangle.P \mid k(x).P \mid \\
 & & k \triangleleft l.P \mid k \triangleright \{l_1 : P_1, l_2 : P_2\} \mid \nu a.P \mid \mathbf{0} \\
(\textit{message types}) & U ::= & \mathsf{bool} \mid \mathsf{int} \mid \ldots \\
(\textit{actions}) & \alpha, \beta ::= & !U \mid ?U \\
(\textit{session types}) & S, T ::= & \mathtt{end} \mid \alpha.S \mid \oplus \{l_1{:}S_1 \, , \, l_2{:}S_2\} \mid \&\{l_1{:}S_1 \, , \, l_2{:}S_2\} \\
(\textit{history types}) & H, K ::= & \hat{}\, S \mid S \hat{} \mid \alpha_1. \cdots .\alpha_n.\hat{}\, S \mid \oplus \{l_i{:}S_i \, , \, l_j{:}H_j\} \mid \&\{l_i{:}S_i \, , \, l_j{:}H_j\}
\end{array}
$$

Figure 1: Syntax of Configurations, Processes, and Types.

Notice that keeping the store σ in the process is useful when values can be shared among different sessions implemented in the process. This is the reason why the store is kept in the process and not in the monitor of a session. As it will become clear later on, the list $\widetilde{u}$ in the running process and the list $\widetilde{x}$ in the monitor will be used to record previously performed actions and reconstruct the process structure accordingly.

The syntax of processes follows standard lines: we consider the idle process $\mathbf{0}$, restriction over channels, as well as complementary prefixes for session establishment (noted $u(x : S).P$ and $u\langle x : S\rangle.Q$, where S is a session type) and prefixes for intra-session communication, namely $k(x).P$ and $k\langle e\rangle.P$ (input and output of expressions) and $k \triangleleft l.P$ and $k \triangleright \{l_1{:}P_1, l_2{:}P_2\}$ (selection and branching). We consider binary labeled choice for simplicity; the extension to the n-ary case is unsurprising. We write $\mathcal{U}$ to denote the set of possible basic values (e.g., integers and booleans); this way, $\mathcal{V} = \mathcal{X} \cup \mathcal{U}$ is the set of values that processes can exchange. We use v, w (and their decorated versions) to range over $\mathcal{V}$.

We will 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 (and their decorated versions) to range over $\mathcal{P}$; also, we use L, M, N to range over $\mathcal{M}$ and A, B, C to range over $\mathcal{A}$. In a formula/statement involving agents in $\mathcal{A}$ we silently assume that all of them are processes or are configurations, but not both.

Types. The syntax of types assumes a set of message types (or *sorts*) ranged over U. We also assume that elements in $\mathcal{U}$ include all possible values belonging to sorts.

In essence, the syntax of types S corresponds to (finite) binary session types [1]: we consider constructs for communication (input and output) and labeled choice (branching and selection). The type $!U.S$ indicates that the owner of the monitor may send a value of type U and then proceed with the behavior prescribed by S. Similarly, the type $?U.S$ says that the owner of the monitor may receive a value of type U, and then proceed with a behavior described by S. A monitor with selection type $\oplus\{l_1{:}S_1\ ,\ l_2{:}S_2\}$ may *either* select label l_1 and proceed as S_1, or select l_2 and proceed as S_2. Similarly, the branching type $\&\{l_1{:}S_1\ ,\ l_2{:}S_2\}$ says that the owner of the monitor may offer *both* label l_1 and then proceed as S_1, or offer label l_2 and then proceed as S_2.

In session types, *duality* is essential to (statically) ensure action compatibility between partners (and therefore, to guarantee absence of communication errors). We rely on a standard definition:

Definition 3.1 (Type Duality)**.** The dual of a session type S, denoted $\overline{S}$, is inductively defined as follows:

$$\begin{aligned}
&\overline{!U.S} = ?U.\overline{S} \qquad \overline{?U.S} = !U.\overline{S} \qquad \overline{\mathtt{end}} = \mathtt{end}\\
&\overline{\oplus\{l_1 : S_1\ ,\ l_2 : S_2\}} = \&\{l_1 : \overline{S_1}\ ,\ l_2 : \overline{S_2}\}\\
&\overline{\&\{l_1 : S_1\ ,\ l_2 : S_2\}} = \oplus\{l_1 : \overline{S_1}\ ,\ l_2 : \overline{S_2}\}
\end{aligned}$$

We will write $\mathsf{dual}(S_1, S_2)$ to indicate $S_1 = \overline{S_2}$.

A key novelty in our work is the use of *history (session) types*, i.e., session types with *present* and *past*. In essence, a history type is a session type enhanced with a separator (or cursor), denoted $\hat{}$. This way, a history type of the form $S_1\hat{}\,S_2$ indicates that S_1 is the past (already executed) behavior of the associated session, while S_2 represents the present behavior (yet to be executed). History types occur only at run-time; the intent is that each time that the process performs a forward computation the cursor will be moved forward by one action; as result of a reversible step, it will be moved backwards by one action.

Intuitively, a history type $\hat{}\,S$ describes a protocol whose actions have not been yet executed (as in, e.g., a just established session); dually, a type $S\hat{}$ corresponds to a protocol whose actions have all been executed. For protocols without labeled choices (as in [24]), history types can be seen as sequences of actions: type $\alpha_1.\cdots.\alpha_n.\hat{}\,S$ is the (intermediate) protocol state in which $n > 0$ communication actions (input or output) have been already performed, and the protocol abstracted by S is yet to be executed. For protocols with labeled choices, however, this scheme should be more general, as cursors need to be injected into labeled branches. Indeed, reversing protocols with labeled choices requires history types of the form $\oplus\{l_i{:}S_i\ ,\ l_j{:}H_j\}$ and $\&\{l_i{:}S_i\ ,\ l_j{:}H_j\}$ (where H_j is a history type, with $i, j \in \{1, 2\}$ and $i \neq j$), as choices made in selection and branching protocols must be recorded. Hence, we formulate history types as tree-like structures (see Figure 1).

Before formally presenting the operational semantics, we give some intuitions on the information carried by monitors. Consider the following configuration, with $s \in \delta$:

$$\left\langle P \cdot \sigma \cdot \widetilde{u}, k \right\rangle_{\delta} \parallel s\lfloor S.?U\hat{}\,T \cdot \widetilde{x}, x \rfloor$$

$$\text{(E.ParC)}\ A \parallel B \equiv B \parallel A \qquad \text{(E.ParA)}\ A \parallel (B \parallel C) \equiv (A \parallel B) \parallel C \qquad \text{(E.NilM)}\ A \parallel \mathbf{0} \equiv A$$

$$\text{(E.NewN)}\ \nu n.\mathbf{0} \equiv \mathbf{0} \qquad \text{(E.NewC)}\ \nu n.\nu m.A \equiv \nu m.\nu n.A \qquad \text{(E.NewP)}\ (\nu n.A) \parallel B \equiv \nu n.(A \parallel B)$$

$$\text{(E.}\alpha\text{)}\ A =_\alpha B \implies A \equiv B$$

Figure 2: Structural congruence

By inspecting the history type $S.?U\hat{}T$, we know that the last action of the process was an input of a value of type U; the information in the lists allows us to infer that the subject and object of the action were k and x, respectively. That is, the previous shape of the process was $k(x).P$.

3.2. Operational Semantics

The operational semantics of our calculus is defined via a reduction relation, coupled with a structural congruence relation; these relations are denoted $\longrightarrow$ and $\equiv$, respectively. The former is defined as a binary relation over configurations, i.e., $\longrightarrow \subset \mathcal{M} \times \mathcal{M}$, while the latter is defined as a binary relation over processes and configurations, i.e., $\equiv \subset \mathcal{P}^2 \cup \mathcal{M}^2$. We require the following definition of *contexts*.

Definition 3.2 (Contexts)**.** *Configuration contexts*, also called evaluation contexts, are configurations with one hole "•" defined by the following grammar: $\mathbb{E} ::= \bullet \mid (M \parallel \mathbb{E}) \mid \nu n.\mathbb{E}$. General contexts $\mathbb{C}$ are processes or configurations with one hole $\bullet$, and are obtained from processes or configurations 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 relation $\mathcal{R}$ that is closed under general contexts: $P\,\mathcal{R}\,Q \implies \mathbb{C}[P]\,\mathcal{R}\,\mathbb{C}[Q]$ and $M\mathcal{R}N \implies \mathbb{C}[M]\,\mathcal{R}\,\mathbb{C}[N]$. The relation $\equiv$ is defined as the smallest congruence, on processes and configurations, that satisfies rules in Figure 2. In defining the rules we adopt Barendregt's Variable Convention: if terms $t_1, \ldots, t_n$ occur in a certain context, then in these terms all bound identifiers and variables are chosen to be different from the free ones. This explains why in Rule (E.NewP) there is no check on free names.

A binary relation $\mathcal{R}$ on closed configurations is *evaluation-closed* if it satisfies the inference rules:

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

To enable reversible labeled choices, we define the following class of *choice contexts*:

Definition 3.3 (Choice Context)**.** *Choice contexts* are processes with one hole, denoted "•", and defined by the following grammar:

$$\mathbb{K}, \mathbb{H} ::= \bullet \mid k \triangleright \{l_i{:}\langle Q_i \rangle\,,\ l_j : \mathbb{K}\}$$

We use the brackets $\langle \cdot \rangle$ in context $k \triangleright \{l_i{:}\langle Q_i \rangle\ ,\ l_j : \bullet\}$ to explicitly indicate the inactive part of the process (i.e., the discarded branch). Choice contexts are defined for processes; we also require a counterpart in the type syntax:

Definition 3.4 (Context Type)**.** *Type contexts* are types with one hole, denoted "$\bullet$", defined by the following grammar:

$$\mathbb{T}, \mathbb{S} ::= \bullet \ \mid\ !U.\mathbb{T} \ \mid\ ?U.\mathbb{T} \ \mid\ \oplus\{l_i{:}S_i, l_j : \mathbb{T}\} \ \mid\ \&\{l_i{:}S_i, l_j : \mathbb{T}\}$$

The reduction relation $\longrightarrow$ is defined as the union of two relations, 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 Figures 3 and 4. Moreover, we indicate with $\longrightarrow^*$, $\twoheadrightarrow^*$, and $\rightsquigarrow^*$ the reflexive and transitive closure of $\longrightarrow$, $\twoheadrightarrow$, and $\rightsquigarrow$, respectively.

Before commenting the reduction relations we need some definitions in place. We first formally define the store σ present in running processes and the operations on stores.

Definition 3.5 (Local Store)**.** A local store σ is a mapping from variables to an ordered list of values. Given a store σ, a variable x, and a value v, we define the *update*, denoted $\sigma[x \mapsto v]$, and *reverse update*, denoted $\sigma \setminus x$, as follows:

$$\sigma[x \mapsto v] = \begin{cases} \sigma \cup \{x, v\} & \text{if } x \notin dom(\sigma) \\ \sigma_1 \cup \{x, \widetilde{v} \cdot v\} & \text{if } \sigma = \sigma_1 \cup \{x, \widetilde{v}\} \end{cases}$$

$$\sigma \setminus x = \begin{cases} \sigma_1 & \text{if } \sigma = \sigma_1 \cup \{x, v\} \\ \sigma_1 \cup \{x, \widetilde{v}\} & \text{if } \sigma = \sigma_1 \cup \{x, \widetilde{v} \cdot v\} \end{cases}$$

The *evaluation* of name n under a store σ, written $\sigma(n)$, is the value v if $\{n, v\} \in \sigma$ or $\{n, \widetilde{v} \cdot v\} \in \sigma$; otherwise, it is n itself.

We stress that the store maintains a correspondence between variables and *lists* of values in order to enable reversibility. The list represents at a given time the assignment history of the variable to which it corresponds, with the actual value of the variable being at the top of the list. If $\sigma(n) = n$ then n is not a variable.

Remark 3.1 (Store and Explicit Substitutions)**.** One challenge in defining reversible semantics for processes is how to treat substitutions, since in general a substitution is not a bijective function. There are at least two possibilities. One may create a copy of a process before applying a substitution and then replace the process with its copy when reverting the substitution [15]. Alternatively, one may use a store and mechanisms based on explicit substitutions [26, 27]. The first technique creates a memory each time a value is substituted; here we implement the second technique, which is more economical because it just remembers the pair variable/value for each substitution.

We now briefly discuss the rules in Figures 3 and 4:

(OPEN)

$$\frac{\mathsf{dual}(S,T) \qquad \overline{s} \notin \delta \qquad s \notin \delta' \qquad \sigma_1(u) = \sigma_2(u')}{\begin{array}{c}\left\langle \mathbb{K}[u\langle x:S\rangle.P] \cdot \sigma_1 \cdot \widetilde{u}_1 \right\rangle_{\delta} \parallel \left\langle \mathbb{H}[u'(y:T).Q] \cdot \sigma_2 \cdot \widetilde{u}_2 \right\rangle_{\delta'} \quad \twoheadrightarrow \\ (\nu s, \overline{s}).\left(\left\langle \mathbb{K}[P] \cdot \sigma_1[x \mapsto \overline{s}] \cdot \widetilde{u}_1, u \right\rangle_{\delta, \overline{s}} \parallel \overline{s}\lfloor \hat{} S \cdot x \rfloor \parallel \left\langle \mathbb{H}[Q] \cdot \sigma_2[y \mapsto s] \cdot \widetilde{u}_2, u' \right\rangle_{\delta', s} \parallel s\lfloor \hat{} T \cdot y \rfloor\right)\end{array}}$$

(COM)

$$\frac{\sigma_1(k) = s \qquad s \in \delta \qquad \sigma_2(k') = \overline{s} \qquad \overline{s} \in \delta' \qquad \sigma_2(e) = v}{\begin{array}{c}\left\langle \mathbb{K}[k(x).P] \cdot \sigma_1 \cdot \widetilde{u}_1 \right\rangle_{\delta} \parallel \left\langle \mathbb{H}[k'\langle e\rangle.Q] \cdot \sigma_2 \cdot \widetilde{u}_2 \right\rangle_{\delta'} \parallel s\lfloor \mathbb{T}[\hat{}?U.S] \cdot \widetilde{e}_1 \rfloor \parallel \overline{s}\lfloor \mathbb{S}[\hat{}!U.T] \cdot \widetilde{e}_2 \rfloor \\ \twoheadrightarrow \\ \left\langle \mathbb{K}[P] \cdot \sigma_1[x \mapsto v] \cdot \widetilde{u}_1, k \right\rangle_{\delta} \parallel \left\langle \mathbb{H}[Q] \cdot \sigma_2 \cdot \widetilde{u}_2, k' \right\rangle_{\delta'} \parallel s\lfloor \mathbb{T}[?U.\hat{} S] \cdot \widetilde{e}_1, x \rfloor \parallel \overline{s}\lfloor \mathbb{S}[!U.\hat{} T] \cdot \widetilde{e}_2, e \rfloor\end{array}}$$

(CHOICE)

$$\frac{\sigma_1(k) = s \qquad s \in \delta \qquad \sigma_2(k') = \overline{s} \qquad \overline{s} \in \delta' \qquad \mathbb{H}'[\bullet] = \mathbb{H}[k' \triangleright \{l_j : \langle Q_j \rangle \, , \, l_i : \bullet\}]}{\begin{array}{c}\left\langle \mathbb{K}[k \triangleleft l_i.P] \cdot \sigma_1 \cdot \widetilde{u}_1 \right\rangle_{\delta} \parallel \left\langle \mathbb{H}[k' \triangleright \{l_j : Q_j \, , \, l_i : Q_i\}] \cdot \sigma_2 \cdot \widetilde{u}_2 \right\rangle_{\delta'} \parallel \\ s\lfloor \mathbb{T}[\hat{} \oplus \{l_j{:}S_j \, , \, l_i{:}S_i\}] \cdot \widetilde{e}_1 \rfloor \parallel \overline{s}\lfloor \mathbb{S}[\hat{}\&\{l_j{:}T_j \, , \, l_i{:}T_i\}] \cdot \widetilde{e}_2 \rfloor \quad \twoheadrightarrow \\ \left\langle \mathbb{K}[P] \cdot \sigma_1 \cdot \widetilde{u}_1, k \right\rangle_{\delta} \parallel \left\langle \mathbb{H}'[Q_i] \cdot \sigma_2 \cdot \widetilde{u}_2, k' \right\rangle_{\delta'} \parallel \\ s\lfloor \mathbb{T}[\oplus\{l_j{:}S_j, l_i{:}\hat{} S_i\}] \cdot \widetilde{e}_1, l_i \rfloor \parallel \overline{s}\lfloor \mathbb{S}[\&\{l_j{:}T_j, l_i{:}\hat{} T_i\}] \cdot \widetilde{e}_2 \rfloor\end{array}}$$

Figure 3: Operational Semantics: Forward Reduction Semantics ($\twoheadrightarrow$).

- Rule OPEN is the forward rule for session establishment. It creates two fresh, dual endpoints and their associated monitors. Each monitor stores a session type; each store records the mapping between the name of the endpoint and the associated channel. The monitor records the variable used by the process to refer to the endpoint, while the name on which the session has started is put on top of the process subject list. The new endpoints are recorded also in the named sequences δ and δ'. Establishing a new session requires type duality (cf. Definition 3.1) and that the two processes refer to the same name (cf. condition $\sigma_1(u) = \sigma_2(u')$). Note that, thanks to Barendregt's Variable Convention, there is no need to check whether the two endpoints are fresh in the two contexts $\mathbb{K}$ and $\mathbb{H}$.
- Rule OPEN* is the opposite of Rule OPEN. In order to revert a session creation, the rule checks that the types of the monitors corresponding to the two endpoints on top of the named list are at their initial position (e.g., $\hat{} S$ and $\hat{} T$). Moreover, the variable lists should contain just one element each. A further check on the names contained on the top of the subjects lists guarantees that the reverted session is the right one. Let us note that since the names list δ is ordered, from a process point the session to be closed correspond to the last opened one (that is the top of the named list). The rule garbage-collects the two endpoints and their associated monitors, and restores the prefixes in the processes.
- Rule COM describes intra-session communication. Two running processes can communicate if

(OPEN*)

$$\frac{\mathsf{dual}(S,T) \qquad \sigma_1(x)=\overline{s} \qquad \sigma_2(y)=s \qquad \sigma_1(u)=\sigma_2(u')}{(\nu s,\overline{s}).\left(\left\langle \mathbb{K}[P]\cdot\sigma_1\cdot\widetilde{u}_1,u\right\rangle_{\delta,\overline{s}} \parallel \overline{s}\lfloor\hat{}\,S\cdot x\rfloor \parallel \left\langle \mathbb{H}[Q]\cdot\sigma_2\cdot\widetilde{u}_2,u'\right\rangle_{\delta',s} \parallel s\lfloor\hat{}\,T\cdot y\rfloor\right) \rightsquigarrow \left\langle \mathbb{K}[u\langle x:S\rangle.P]\cdot\sigma_1\setminus x\cdot\widetilde{u}_1\right\rangle_{\delta} \parallel \left\langle \mathbb{H}[u'(y:T).Q]\cdot\sigma_2\setminus y\cdot\widetilde{u}_2\right\rangle_{\delta'}}$$

(COM*)

$$\frac{\sigma_1(k)=s \qquad s\in\delta \qquad \sigma_2(k')=\overline{s} \qquad \overline{s}\in\delta'}{\left\langle \mathbb{K}[P]\cdot\sigma_1\cdot\widetilde{u}_1,k\right\rangle_{\delta} \parallel \left\langle \mathbb{H}[Q]\cdot\sigma_2\cdot\widetilde{u}_2,k'\right\rangle_{\delta'} \parallel s\lfloor\mathbb{T}[?U.\hat{}\,S_1]\cdot\widetilde{e}_1,x\rfloor \parallel \overline{s}\lfloor\mathbb{S}[!U.\hat{}\,S_2]\cdot\widetilde{e}_2,e\rfloor \rightsquigarrow \left\langle \mathbb{K}[k(x).P]\cdot\sigma_1\setminus x\cdot\widetilde{u}_1\right\rangle_{\delta} \parallel \left\langle \mathbb{H}[k\langle e\rangle.Q]\cdot\sigma_2\cdot\widetilde{u}_2\right\rangle_{\delta'} \parallel s\lfloor\mathbb{T}[\hat{}\,?U.S_1]\cdot\widetilde{e}_1\rfloor \parallel \overline{s}\lfloor\mathbb{S}[\hat{}\,!U.S_2]\cdot\widetilde{e}_2\rfloor}$$

(CHOICE*)

$$\frac{\sigma_1(k)=s \qquad s\in\delta \qquad \sigma_2(k')=\overline{s} \qquad \overline{s}\in\delta' \qquad \mathbb{H}'[\bullet]=\mathbb{H}[k'\triangleright\{l_j:\langle Q_j\rangle\,,\,l_i:\bullet\}]}{\begin{array}{c}\left\langle \mathbb{K}[P]\cdot\sigma_1\cdot\widetilde{u}_1,k\right\rangle_{\delta} \parallel \left\langle \mathbb{H}'[Q_i]\cdot\sigma_2\cdot\widetilde{u}_2,k'\right\rangle_{\delta'} \parallel \\ s\lfloor\mathbb{T}[\oplus\{l_j{:}S_j,l_i{:}\hat{}\,S_i\}]\cdot\widetilde{e}_1,l_i\rfloor \parallel \overline{s}\lfloor\mathbb{S}[\&\{l_j{:}T_j,l_i{:}\hat{}\,T_i\}]\cdot\widetilde{e}_2\rfloor \quad \rightsquigarrow \\ \left\langle \mathbb{K}[k\triangleleft l_i.P]\cdot\sigma_1\cdot\widetilde{u}_1\right\rangle_{\delta} \parallel \left\langle \mathbb{H}[k'\triangleright\{l_j:Q_j\,,\,l_i:Q_i\}]\cdot\sigma_2\cdot\widetilde{u}_2\right\rangle_{\delta'} \parallel \\ s\lfloor\mathbb{T}[\hat{}\,\oplus\{l_j{:}S_j\,,\,l_i{:}S_i\}]\cdot\widetilde{e}_1\rfloor \parallel \overline{s}\lfloor\mathbb{S}[\hat{}\,\&\{l_j{:}T_j\,,\,l_i{:}T_i\}]\cdot\widetilde{e}_2\rfloor\end{array}}$$

Figure 4: Operational Semantics: Backward Reduction Semantics ($\rightsquigarrow$).

they refer to the same session. The sent value v is obtained by evaluating e under the sender store σ_2. As a result, the store of the receiver is updated with a new value, bound to the read variable: $\sigma_1[x \mapsto v]$. Also, monitor types are moved one step forward, and both read and sent variables are put on the top of the list in their respective monitors; the same occurs for the session names which are put in the subjects lists. This way we keep information about the prefixes (noted $k(x).-$ and $k'\langle e\rangle.-$ in the rule).

- Rule COM* undoes a communication: the sent value (along with the variable used to read it) is eliminated from the store of the receiving process. The variable list of the monitor keeps information on which variable to unbound: indeed, it is the variable at the top of this list the one that has to be eliminated, as we want to revert the input of its associated value. Moreover, information contained in the subjects list of the processes is used to recover output and input prefixes. Notice that the information about the kind of prefix to be built again (input or output) is given by the type of the monitor. A further consequence of undoing a communication is that the session types are moved one step backward.

- Rule CHOICE is the forward rule for the choice. A choice is made by a running process that selects a branch (identified by a label l_i) of those offered by a complementary process in the same session. Subsequently, the two processes proceed into executing the selected branch, and the remaining branches (l_j in the rule) are discarded. To keep track of these non selected branches,

we use *static* choice contexts (cf. Definition 3.3), inserting process Q_j in the brackets $\langle \cdot \rangle$. As in the rule for communication, the information needed to restore the selection and branching prefixes is recorded in the corresponding lists, and the cursors in their types are moved forward by one position. In this case, this means moving the cursor inside the selected branch.

- Rule CHOICE* undoes a choice. For this reverse step to be enabled, the types of both monitors have to be in initial position inside the same branch (denoted l_i in the rule). The two cursors are moved backwards by one position to the point of the protocol that preceded the choice. The prefix of the monitored process that performed the selection is restored by using the information contained in the variable and subject lists. The process that offered the choice reverts it by enabling again the discarded branch, i.e., the process contained into the brackets $\langle \cdot \rangle$ is restored.

3.3. Example

We illustrate our approach by means of a simple protocol: a variation of the *two buyers protocol* [2]. The protocol involves three participants: a Buyer, a Seller, and a Buyer's Friend. Buyer first sends to Seller the title of a book he is interested in buying. Seller replies with the price of the book and offers two choices to the Buyer: either to continue with the purchase or to quit the session. If the Buyer is willing to finalize the purchase, then Seller awaits for additional information (*shipping address* and *order confirmation*) from Buyer, before providing a *delivery date*. Once Buyer receives the price and accepts it, he contacts Friend in order to get a loan and finalize the purchase.

The set of structured interactions of Buyer with Seller and Friend can be described by the following session types:

$$S_a : ?\mathbf{str}.!\mathbf{int}.\&\{ok{:}?\mathbf{str}.?\mathbf{int}.!\mathbf{cal}.\mathtt{end}\,,\ quit{:}\mathtt{end}\} \qquad S_b : ?\mathbf{int}.!\mathbf{int}.\mathtt{end}$$

Above, S_a describes the interaction between Buyer and Seller, from Seller's perspective; also, S_b represents the interaction between Buyer and Friend, from Friend's perspective. We write T_a and T_b to denote the dual session types of S_a and S_b, respectively.

We now proceed to examine some possible process implementations for Buyer, Seller, and Friend. The behavior of Buyer may be specified by the following process:

$$\begin{aligned}
\textsc{Buyer} \triangleq\ & a\langle z : T_a\rangle.z\langle \text{“}dune\text{”}\rangle.z(prc).\\
& \mathbf{if}(\mathtt{decide}(prc))\\
& \quad \mathbf{then}\ z \triangleleft ok.b\langle w : T_b\rangle.w\langle \mathtt{loan}(prc)\rangle.w(cash).\\
& \quad\quad z\langle addr\rangle.z\langle cash\rangle.z(date).\mathbf{0}\\
& \quad \mathbf{else}\ z \triangleleft quit.\mathbf{0}
\end{aligned}$$

For simplicity, we have used the process $\mathbf{if}\, e\, \mathbf{then}\, P\, \mathbf{else}\, Q$, which has the expected meaning. Although it is not present in the our syntax, it is similar to a labeled choice and its reversible semantics (forward and backward rules, associated contexts) can be formalized in a straightforward manner.

Process BUYER involves the creation of two interleaved sessions: the first one is established with the prefix $a\langle z : T_a\rangle$, which explicitly declares the protocol to be executed with Seller's implementation; the second session is established with Friend's implementation through the prefix $b\langle w : T_b\rangle$. Implementations for Seller and Friend can be specified by the following processes:

$$\begin{aligned}\text{SELLER} &\triangleq a(z : S_a).z(title).z\langle \texttt{quote}(title)\rangle.\\ &\qquad z \triangleright \{ok{:}z(addr).z(payment).z\langle \texttt{date}(addr)\rangle.\mathbf{0}\,,\; quit{:}\mathbf{0}\}\\ \text{FRIEND} &\triangleq b(w : S_b).w(amount).w\langle loan\rangle.\mathbf{0}\end{aligned}$$

Above, we use functions $\texttt{loan}()$, $\texttt{quote}()$ and $\texttt{date}()$ to abstract away from the amount of money to be borrowed, the price of the book, and the delivery date, respectively. The overall system specification is then given by the parallel composition of configurations containing the three processes (in what follows, ϵ denotes the empty list):

$$\text{SYSTEM} \triangleq \left\langle \text{BUYER} \cdot \epsilon \cdot \epsilon \right\rangle_{\kappa_1:\epsilon} \parallel \left\langle \text{SELLER} \cdot \epsilon \cdot \epsilon \right\rangle_{\kappa_2:\epsilon} \parallel \left\langle \text{FRIEND} \cdot \epsilon \cdot \epsilon \right\rangle_{\kappa_3:\epsilon}$$

In the following, we will indicate with BUYER_i (resp. SELLER_i and FRIEND_i) the process BUYER after performing its i-th action. We will follow a similar convention with types.

The first forward reduction of SYSTEM is establishing a session between Buyer and Seller, using the fact that T_a and S_a are dual types. We have:

$$\begin{aligned}\text{SYSTEM} \twoheadrightarrow (\nu s, \overline{s}).\Big(&\left\langle \text{BUYER}_1 \cdot \{z, s\} \cdot a\right\rangle_{\kappa_1:s} \parallel s\lfloor \hat{}\, T_a \cdot z\rfloor \parallel\\ &\left\langle \text{SELLER}_1 \cdot \{z, \overline{s}\} \cdot a\right\rangle_{\kappa_2:\overline{s}} \parallel \overline{s}\lfloor \hat{}\, S_a \cdot z\rfloor \parallel \left\langle \text{FRIEND} \cdot \epsilon \cdot \epsilon\right\rangle_{\kappa_3:\epsilon}\Big) = M_0 \qquad (7)\end{aligned}$$

Once a session is established two monitors are created, one per endpoint: their task is to discipline the behavior of the process holding the endpoint. For example, the behavior of Buyer in session s has to obey session type S_a. Following S_a, Buyer then sends to Seller the request for the book, and so the entire system evolves as follows:

$$\begin{aligned}M_0 \twoheadrightarrow (\nu s, \overline{s}).\Big(&\left\langle \text{BUYER}_2 \cdot (\{z, s\}) \cdot a, z\right\rangle_{\kappa_1:s} \parallel s\lfloor !\texttt{str} \hat{}\, T_{a_1} \cdot z, \text{“}dune\text{”}\rfloor \parallel\\ &\left\langle \text{SELLER}_2 \cdot (\{z, \overline{s}\}, \{title, \text{“}dune\text{”}\}) \cdot a, z\right\rangle_{\kappa_2:\overline{s}} \parallel\\ &\overline{s}\lfloor ?\texttt{str} \hat{}\, S_{a_1} \cdot z, title\rfloor \parallel \left\langle \text{FRIEND} \cdot \epsilon \cdot \epsilon\right\rangle_{\kappa_3:\epsilon}\Big) = M \qquad (8)\end{aligned}$$

One effect of the reduction is that both types register the action and move forward. Another effect is that the information needed to restore back the consumed prefixes is stored into the running configurations and the related monitors. The forward step in (8) can be reverted by moving backward

the monitor types, restoring the prefixes, and deleting the read value from the receiver store:

$$M \rightsquigarrow (\nu s, \overline{s}).\Big(\big\langle z\langle \text{"}dune\text{"}\rangle.\textsc{Buyer}_2 \cdot \{z, s\} \cdot a\big\rangle_{\kappa_1:s} \parallel s\lfloor \hat{}\,!\mathtt{str}.T_{a_1} \cdot z\rfloor \parallel$$
$$\big\langle z(title).\textsc{Seller}_2 \cdot \{z, \overline{s}\} \cdot a\big\rangle_{\kappa_2:\overline{s}} \parallel \overline{s}\lfloor \hat{}\,?\mathtt{str}.S_{a_1} \cdot z\rfloor \parallel \big\langle \textsc{Friend} \cdot \epsilon \cdot \epsilon\big\rangle_{\kappa_3:\epsilon}\Big) \tag{9}$$

We can easily check that the configurations in (7) and (9) are equivalent. From M in (8) the interaction between Buyer and Seller can go on. Let us suppose that the Buyer decides that the price is fine and the session goes along the "ok" branch. Then we have the following configuration:

$$M \twoheadrightarrow^* (\nu s, \overline{s}).\Big(\big\langle \mathbb{H}[\textsc{Buyer}_{ok}] \cdot (\{z, s\}, \{prc, 10\}) \cdot a, z, z, z\big\rangle_{\kappa_1:s} \parallel$$
$$s\lfloor \mathbb{T}[\oplus\{ok\!:\hat{}\,T_a^{ok}\,,\ quit\!:\!\mathtt{end}\}] \cdot z, \text{“}dune\text{”}, prc, ok\rfloor \parallel$$
$$\big\langle \mathbb{K}[\textsc{Seller}_{ok}] \cdot (\{z, \overline{s}\}, \{title, \text{“}dune\text{”}\}) \cdot a, z, z, z\big\rangle_{\kappa_2:\overline{s}} \parallel$$
$$\overline{s}\lfloor \mathbb{S}[\&\{ok\!:\hat{}\,S_a^{ok}\,,\ quit\!:\!\mathtt{end}\}] \cdot z, title, \mathtt{quote}(title)\rfloor \parallel$$
$$\big\langle \textsc{Friend} \cdot \epsilon \cdot \epsilon\big\rangle_{\kappa_3:\epsilon}\Big) = M_1 \tag{10}$$

where $\textsc{Buyer}_{ok}$, $\textsc{Seller}_{ok}$, T_a^{ok} and S_a^{ok} are respectively the Buyer and the Seller processes after choosing ok and their corresponding types. Moreover, we have:

$$\mathbb{H}[\bullet] = \mathbf{if}(\mathtt{decide}(prc))\ \mathbf{then}\ \bullet\ \mathbf{else}\ \langle z \triangleleft quit\rangle \qquad \mathbb{T}[\bullet] = \ !\mathbf{str}.?\mathbf{int}.\bullet$$
$$\mathbb{K}[\bullet] = z \triangleright \{ok\!:\!\bullet\,,\ quit\!:\!\langle\mathbf{0}\rangle\} \qquad \mathbb{S}[\bullet] = \ ?\mathbf{str}.!\mathbf{int}.\bullet$$

From M_1 in (10), the Buyer can establish a new session with Friend:

$$M_1 \twoheadrightarrow (\nu s, \overline{s}, r, \overline{r}).\Big(\big\langle \mathbb{H}[\textsc{Buyer}_{ok_1}] \cdot (\{z, s\}, \{prc, 10\}, \{w, r\}) \cdot a, z, z, z, b\big\rangle_{\kappa_1:s,r} \parallel$$
$$s\lfloor \mathbb{T}[\oplus\{ok\!:\hat{}\,T_a^{ok}\,,\ quit\!:\!\mathtt{end}\}] \cdot z, \text{“}dune\text{”}, prc, ok\rfloor \parallel r\lfloor \hat{}\,T_b \cdot w\rfloor \parallel$$
$$\big\langle \mathbb{K}[\textsc{Seller}_{ok}] \cdot (\{z, \overline{s}\}, \{title, \text{“}dune\text{”}\}) \cdot a, z, z\big\rangle_{\kappa_2:\overline{s}} \parallel$$
$$\overline{s}\lfloor \mathbb{S}[\&\{ok\!:\hat{}\,S_a^{ok}\,,\ quit\!:\!\mathtt{end}\}] \cdot z, title, \mathtt{quote}(title), z\rfloor \parallel$$
$$\big\langle \textsc{Friend}_1 \cdot \{w, \overline{r}\} \cdot b\big\rangle_{\kappa_3:\overline{r}} \parallel \overline{r}\lfloor \hat{}\,S_b \cdot w\rfloor\Big) \tag{11}$$

Thus, the running process for Buyer is present in two sessions: one with Seller and another one with Friend, and has two associated monitors, identified by endpoints s, r. The list of subjects stored into the running process allows us to reverse session communications (possibly in different sessions) and session establishments exactly in the same order in which they were performed, thus respecting causality of actions. In this way, Buyer cannot undo a communication with Seller while the session with Friend is still established.

3.4. Basic Properties

Having illustrated the way in which our semantics intuitively respects causality of actions, we now move on to formally establish this property for our reversible framework. We will show that our framework satisfies the *Loop Lemma*, a property that gives us a basic guarantee of the consistency between forward and backward reductions. We require some auxiliary definitions, notations, and results. In the following, we write $\prod_{i\in I} A_i$ as a short-hand notation for $A_1 \parallel \cdots \parallel A_n$ with $I = \{1, \ldots, n\}$. Notice that, by convention, we assume that $\prod_{i\in I} A_i = \mathbf{0}$ if $I = \emptyset$.

Definition 3.6 (Initial and Reachable Configurations)**.** A configuration M is *initial* if

$$M \equiv \nu\widetilde{a}.\left(\prod_{i\in I}\left\langle P_i \cdot \sigma_i \cdot \epsilon\right\rangle_{\kappa_i:\epsilon}\right) \quad \text{with} \quad \forall i,j \in I \implies \kappa_i \neq \kappa_j$$

A configuration is *reachable* if it can be derived using $\longrightarrow$ from an initial configuration. That is, if M_0 is an initial configuration and $M_0 \longrightarrow^* M$, then M is reachable.

The following lemma establishes a normal form for configurations:

Lemma 3.1 (Normal Form)**.** *For any configuration M we have that:*

$$M \equiv \nu\widetilde{a}.\left(\prod_{i\in I}\left\langle \mathbb{K}_i\left[P_i\right] \cdot \sigma_i \cdot \widetilde{u}_i\right\rangle_{\delta_i} \parallel \prod_{j\in J} s_j\lfloor H_j \cdot \widetilde{e}_j\rfloor\right)$$

Proof. By induction on the structure of configurations/processes. □

We show that reachable configurations enjoy various structural properties, referred in the following to as *well-formedness*:

Definition 3.7 (Well-formed Configurations)**.** We say that configuration

$$M \equiv \nu\widetilde{a}.\left(\prod_{i\in I}\left\langle \mathbb{K}_i\left[P_i\right] \cdot \sigma_i \cdot \widetilde{u}_i\right\rangle_{\delta_i} \parallel \prod_{j\in J} s_j\lfloor H_j \cdot \widetilde{e}_j\rfloor\right)$$

is *well-formed* if

1. $\forall i,j \in I, i \neq j \implies \delta_i \cap \delta_j = \emptyset$;
2. $\forall i,j \in J, i \neq j \implies s_i \neq s_j$;
3. $\forall i \in J,\ \exists\, j \in J$ such that $s_i = \overline{s}_j$;
4. $\forall i \in I$
   (a) $\forall s \in \delta_i,\ \exists\, j \in J$ such that $s = s_j$

(b) $\forall s_1, s_2 \in \delta_i, s_1 \neq s_2$

A well-formed configuration enjoys four properties: (1) all the identifiers of running processes are *unique* and do not share session names; (2) all monitors have a unique session endpoint; (3) each monitor has a unique dual; (4) for each session endpoint contained into a running process identifier there exists a unique corresponding monitor bearing the same name. Therefore, although the syntax of Figure 1 allows to express configurations in which a session appears in more than one running process, because of condition (1) above we decree such configurations as ill-formed.

Well-formedness is preserved through structural equivalence and reduction:

Proposition 3.1. If M is well formed and $M \equiv N$ then N is well formed.

Proof. By induction on the derivation $M \equiv N$, with a case analysis on the last applied axiom. All cases are easy. □

Proposition 3.2. Any reachable configuration M is well formed.

Proof. By Definition 3.6, M is a reachable configuration if $M_0 \longrightarrow^* M$, for some initial configuration M_0. The proof is then by induction the length of the reduction sequence $M_0 \longrightarrow^* M$. See Appendix A.1 for details. □

We now prove the Loop Lemma, which shows that forward and backward reductions are the inverse of each other. Then we have:

Lemma 3.2 (Loop Lemma)**.** *Let M and N be reachable configurations. Then: $M \twoheadrightarrow N \iff N \rightsquigarrow M$.*

Proof. By induction on the derivation of $M \twoheadrightarrow N$ for the if direction, and on the derivation of $N \rightsquigarrow M$ for the converse. See Appendix A.2 for more details. □

We may then show the following:

Corollary 3.1. For any reachable configuration M, N, if $M \longrightarrow^* N$ then $N \longrightarrow^* M$.

Proof. The proof proceeds by induction on the length of the sequence of reductions $M \longrightarrow^* N$, using the Loop Lemma (Lemma 3.2). □

4. Causal Consistency

Up to here, we have introduced and illustrated our process framework and established a basic property of its forward and backward semantics. In this section we investigate *causal consistency*, a property of *sequences* of reductions. Intuitively, causal consistency characterizes a space for admissible rollbacks which are:

1. *Consistent*, in the sense that they do not lead to previously unreachable configurations; and

2. *Flexible*, so as to allow rearranging of reversed actions. This may enable us to obtain rollback sequences more efficient than those decreeing the naive reversal of each performed step.

As a consequence of causal consistency, the set of states reached by a backward computation are states that could have been reached by performing only forward computations.

While the Loop Lemma (Lemma 3.2) ensures *local* coherence for reductions (precisely, forward reductions with respect to backward ones, and viceversa), causal consistency is a *global* property, as it concerns *sequences* of reductions, or *traces*. It is formally expressed by Theorem 4.1 (Page 22). In its proof we adapt arguments from [13]. In particular, we instrument the reduction semantics $\longrightarrow$ with a *reduction stamp*, denoted η, which contains information useful to understand when two reductions are *concurrent*, i.e., they may be executed in parallel. As we will see, working in a session-based setting will enable us to have compact reduction stamps, simpler than in previous works, therefore streamlining associated proofs. We will then have *labeled reductions* of the form $M \xrightarrow{\eta} N$, where M and N are reachable configurations, $M \longrightarrow N$, and η is a reduction stamp, as defined next. We will find it convenient to use $t, t', \ldots$ to identify labeled reductions: we shall write $t : M \xrightarrow{\eta} N$ to this end.

Definition 4.1 (Reduction stamp)**.** Let M, N be reachable configurations such that $M \longrightarrow N$. We then write $M \xrightarrow{\eta} N$, where the *reduction stamp* η is defined as:

- $\eta = \{\kappa_i : s\,;\, \kappa_j : \overline{s}\}_{\twoheadrightarrow} \triangleq$

$$M \equiv \mathbb{E}\big[\big\langle P \cdot \sigma_i \cdot \widetilde{u}_i \big\rangle_{\kappa_i : \widetilde{s}_i} \parallel \big\langle Q \cdot \sigma_j \cdot \widetilde{u}_j \big\rangle_{\kappa_j : \widetilde{s}_j}\big]$$
$$N \equiv \mathbb{E}\big[\big\langle P' \cdot \sigma_i' \cdot \widetilde{u}_i, u \big\rangle_{\kappa_i : \widetilde{s}'_i} \parallel \big\langle Q' \cdot \sigma_j' \cdot \widetilde{u}_j, u' \big\rangle_{\kappa_j : \widetilde{s}'_j}\big]$$
$$M \twoheadrightarrow N \qquad \sigma_i'(u) = s \qquad \sigma_j'(u') = \overline{s}$$

- $\eta = \{\kappa_i : s\,;\, \kappa_j : \overline{s}\}_{\rightsquigarrow} \triangleq$

$$M \equiv \mathbb{E}\big[\big\langle P \cdot \sigma_i \cdot \widetilde{u}_i, u \big\rangle_{\kappa_i : \widetilde{s}_i} \parallel \big\langle Q \cdot \sigma_j \cdot \widetilde{u}_j, u' \big\rangle_{\kappa_j : \widetilde{s}_j}\big]$$
$$N \equiv \mathbb{E}\big[\big\langle P' \cdot \sigma_i' \cdot \widetilde{u}_i \big\rangle_{\kappa_i : \widetilde{s}'_i} \parallel \big\langle Q' \cdot \sigma_j' \cdot \widetilde{u}_j \big\rangle_{\kappa_j : \widetilde{s}'_j}\big]$$
$$M \rightsquigarrow N \qquad \sigma_i(u) = s \qquad \sigma_j(u') = \overline{s}$$

Thus, the stamp of a reduction contains the names of the named sequences and endpoints of the running processes involved into the reduction; it also records the direction of the reduction. We let λ (and its decorated variants) range over pairs of the form $\{\kappa_i : s\,;\, \kappa_j : \overline{s}\}$, which we sometimes treat as a set. This way, a stamp η can be of the form $\lambda_{\twoheadrightarrow}$ or $\lambda_{\rightsquigarrow}$. By a slight abuse of notation, given $\eta = \{\kappa_i : s\,;\, \kappa_j : \overline{s}\}_{\twoheadrightarrow}$ or $\eta = \{\kappa_i : s\,;\, \kappa_j : \overline{s}\}_{\rightsquigarrow}$, we write $\lambda(\eta)$ to denote the pair $\{\kappa_i : s\,;\, \kappa_j : \overline{s}\}$. The *inverse* of the stamp $\eta = \lambda_{\twoheadrightarrow}$ is defined as $\eta_\bullet = \lambda_{\rightsquigarrow}$, and viceversa.

Definition 4.2 (Notation and Terminology for Labeled Reductions)**.** Let $t : M \xrightarrow{\eta} N$ be a reduction.

- We say M is the *source* of the reduction (denoted $\texttt{source}(t)$), N is its *target* (denoted $\texttt{target}(t)$), and that the stamp η is the reduction label.
- Two reductions are said to be *coinitial* if they have the same source; *cofinal* if they have the same target; and *composable* if the target of the first reduction is the source of the other.
- We say that t is *forward* if $\eta = \lambda_{\twoheadrightarrow}$, and that is *backward* if $\eta = \lambda_{\rightsquigarrow}$.
- The *inverse* of t, denoted $t_\bullet$, is the reduction $t_\bullet : N \xrightarrow{\eta_\bullet} M$, and viceversa.
- Given coinitial reductions $t_1 : M \xrightarrow{\eta_1} N_1$ and $t_2 : M \xrightarrow{\eta_2} N_2$, we define t_2/t_1 (read "t_2 after t_1") as $N_1 \xrightarrow{\eta_2} N_3$, i.e., the reduction with stamp η_2 that starts from the target of t_1.

Example 4.1. We refer to the example of Section 3.3 in order to illustrate some of the above notions.

- If we take the transition $M_0 \twoheadrightarrow M$ given in (8) then we can derive the following labelled reduction $t : M_0 \xrightarrow{\{\kappa_1 : s\,;\, \kappa_2 : \overline{s}\}_{\twoheadrightarrow}} M$, which is a forward reduction.
- Similarly, from transition $M \rightsquigarrow M_0$ given in (9) we can derive $r : M \xrightarrow{\{\kappa_1 : s\,;\, \kappa_2 : \overline{s}\}_{\rightsquigarrow}} M_0$, which is a backward reduction. Clearly, $r = t_\bullet$ (and vice-versa).
- The reductions of (8) and (9) are composable, since the target of the first one is the origin of the second one.

The following definition ensures a consistent use of fresh names throughout (labeled) reductions, important to correctly identify configurations and processes based on their stamps:

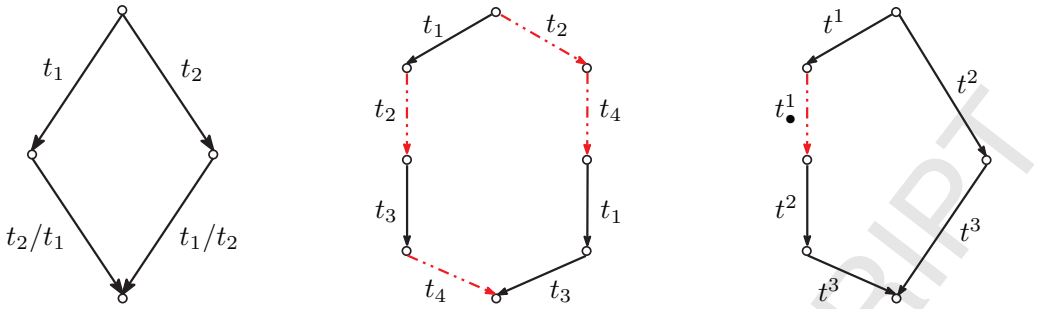
Definition 4.3 (Name-preserving labeled reductions)**.** We say that a (labeled) reduction t is *name-preserving* if:

- t is derived without using α-conversion;
- If t creates a pair of endpoints (cf. Rule OPEN) then these fresh names are chosen using a fixed function from the names of the running processes involved in t.

Intuitively, the second item in the above definition ensures that, for any $u_1, u_2, \kappa_1, \kappa_2$, whenever u_1 and u_2 synchronize to establish a session (connecting configurations identified by κ_1 and κ_2, respectively) then the pair of fresh endpoints will always be the same. This is relevant when the session is established more than once, e.g., when the reduction enabled by Rule OPEN is undone and performed again. From now on we will just consider only name-preserving reductions.

Definition 4.4 (Intersection of Stamps)**.** Let η_1 and η_2 be such that $\lambda(\eta_1) = \{\kappa_1 : s\,;\, \kappa_2 : \overline{s}\}$ and $\lambda(\eta_2) = \{\kappa_3 : t\,;\, \kappa_4 : \overline{t}\}$. We write $\lambda(\eta_1) \cap \lambda(\eta_2)$ to denote the intersection of the underlying sets:

$$\lambda(\eta_1) \cap \lambda(\eta_2) = \{\kappa_1, s, \kappa_2, \overline{s}\} \cap \{\kappa_3, t, \kappa_4, \overline{t}\}$$



(a) Square Lemma (Lemma 4.1)

(b) Rearranging Lemma (Lemma 4.2)

(c) Shortening Lemma (Lemma 4.3)

Figure 5: Graphical intuitions for the ingredients of the proof of causal consistency (Theorem 4.1). Black, solid arrows represent forward reductions; red, dashed arrows represent backward reductions. In (b), we assume that $t_1 \neq t_{2\bullet}$, $t_3 \neq t_{2\bullet}$, and $t_3 \neq t_{4\bullet}$.

We are now ready to give the key notions of concurrent and conflicting reductions:

Definition 4.5 (Conflicting and Concurrent Reductions)**.** Two coinitial reductions $t_1 : M \xrightarrow{\eta_1} N_1$ and $t_2 : M \xrightarrow{\eta_2} N_2$ are said to be in *conflict* if $\lambda(\eta_1) \cap \lambda(\eta_2) \neq \emptyset$. Two coinitial reductions are *concurrent* if they are not in conflict.

Example 4.2. Consider the following configuration:

$$M = \langle u_1\langle x : S\rangle.P \cdot \sigma_1 \cdot \epsilon\rangle_{\kappa_1:\epsilon} \parallel \langle u_2(y : T).Q_1 \cdot \sigma_2 \cdot \epsilon\rangle_{\kappa_2:\epsilon} \parallel \langle u_3(y : T).Q_2 \cdot \sigma_3 \cdot \epsilon\rangle_{\kappa_3:\epsilon}$$

with $\mathsf{dual}(S, T)$ and $\sigma_1(u_1) = \sigma_2(u_2) = \sigma_3(u_3)$. Clearly, there are two reductions from M, namely $t_1 : M \xrightarrow{\{\kappa_1:\epsilon\,;\,\kappa_2:\epsilon\}\twoheadrightarrow} M_1$ and $t_2 : M \xrightarrow{\{\kappa_2:\epsilon\,;\,\kappa_3:\epsilon\}\twoheadrightarrow} M_2$. Since the intersection of their respective stamps is not empty, t_1 and t_2 are conflicting reductions.

A property that a reversible calculus should enjoy is the so-called *Square Lemma*, which may be informally described as follows. Assume a configuration from which two reductions are possible: if these reductions are *concurrent* then the order in which they are executed does not matter, and the same configuration is reached. Formally we have the *Square Lemma* [13]:

Lemma 4.1 (Square Lemma)**.** *If $t_1 : M \xrightarrow{\eta_1} M_1$ and $t_2 : M \xrightarrow{\eta_2} M_2$ are coinitial and concurrent reductions, then there exist cofinal reductions $t_2/t_1 = M_1 \xrightarrow{\eta_2} N$ and $t_1/t_2 = M_2 \xrightarrow{\eta_1} N$.*

Proof. By case analysis on the form of t_1 and t_2. See Appendix A.3 for more details. □

In the above lemma, N is the configuration reached by executing both reductions t_1 and t_2. Figure 5a gives an illustration of the Square Lemma; reductions t_1 and t_2 are both concurrent and co-initial. Our goal is to show that our reversible semantics is causally consistent, as motivated earlier. To this end, we shall be interested in *traces*—sequences of reductions between reachable configurations.

Definition 4.6 (Traces)**.** A sequence of pairwise composable reductions is called a *trace*. We let ρ and its decorated variants range over traces.

Auxiliary notions defined for reductions extend naturally to traces:

Definition 4.7 (Notation and Terminology for Traces)**.** We assume the following notations and terminology:

- We write ϵ_M to denote the *empty trace* with source M.
- We write $\rho_1; \rho_2$ to denote the composition of two composable traces ρ_1 and ρ_2.
- The *inverse* of a trace $\rho = t^1; \ldots; t^n$ is defined as $\rho_\bullet = t^n_\bullet; \ldots; t^1_\bullet$.
- Given a trace $t^1; \ldots; t^n$, we say that reductions t^i and t^{i+1} $(1 \leq i \leq n-1)$ are *contiguous*. We say that they are *opposing* if $t^{i+1} = t'_\bullet$ and both t^i and t' have the same direction.
- A trace is *forward* (resp. *backward*) if it is composed of forward (resp. backward) reductions.

Moreover, we will write $\mathsf{len}(\rho)$ to denote the length of ρ, defined as expected.

We are now in a position to show that our reversible semantics is causally consistent. Following Lévy [28], we define a notion of *causal equivalence* between traces, noted $\asymp$, which abstracts away from the order in which concurrent reductions are executed.

Definition 4.8 (Causal Equivalence)**.** We define $\asymp$ as the least equivalence relation between traces that is closed under composition and that obeys the following rules:

$$t_1; t_2/t_1 \asymp t_2; t_1/t_2 \qquad t; t_\bullet \asymp \epsilon_{\mathtt{source}(t)} \qquad t_\bullet; t \asymp \epsilon_{\mathtt{target}(t)}$$

Intuitively, $\asymp$ says that that (i) if we have two concurrent reductions, then the traces obtained by swapping their execution order are equivalent, and (ii) a trace consisting of opposing reductions is equivalent to the empty trace (i.e., the two reductions cancel themselves out).

The proof of causal consistency proceeds along the same lines as in [13], but with simpler arguments because of the simpler form of our reduction 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. Figure 5b illustrates this situation: the trace depicted on the left-hand side (a combination of forward and backward reductions) is causally equivalent to the trace depicted on the right-hand side (where backward reductions appear first).

Lemma 4.2 (Rearranging Lemma)**.** *Given a trace ρ, there exist forward traces ρ' and ρ'' such that $\rho \asymp \rho'_\bullet; \rho''$.*

Proof. By lexicographic induction on $\mathsf{len}(\rho)$ and on the distance between the beginning of ρ and the earliest pair of opposing reductions in ρ. The analysis uses both the Loop Lemma (Lemma 3.2) and the Square Lemma (Lemma 4.1). For more details see Appendix A.4. □

The next lemma considers the following situation: 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" computations, not present in ρ_2, which must be undone at some later point in ρ_1—otherwise such computations would represent a difference with respect to the computations in ρ_2. This means that ρ_1 could be shortened by removing local computations and their corresponding reverse steps. Figure 5c exemplifies this situation: the trace on the left-hand side starts by doing and undoing a reduction; it is causally equivalent to the (coinitial and cofinal) trace in the right-hand side, which does simply not contain these forward and backward reductions.

Lemma 4.3 (Shortening Lemma)**.** *Let ρ_1 and ρ_2 be coinitial and cofinal traces, with ρ_2 forward. Then, there exists a forward trace ρ_1' such that $\rho_1' \asymp \rho_1$ and $\mathsf{len}(\rho_1') \leq \mathsf{len}(\rho_1)$.*

Proof (Sketch). The proof is by induction on $\mathsf{len}(\rho_1)$, using Square and Rearranging Lemmas (Lemmas 4.1 and 4.2). In the proof, the forward trace ρ_2 is the main guideline for shortening ρ_1 into a forward trace. Indeed, the proof relies crucially on the fact that ρ_1 and ρ_2 share the same source and target and that ρ_2 is a forward trace. See Appendix A.5 for details. □

We are now finally ready to state our main result:

Theorem 4.1 (Causal Consistency)**.** Let ρ_1 and ρ_2 be traces. Then $\rho_1 \asymp \rho_2$ if and only if ρ_1 and ρ_2 are coinitial and cofinal.

Proof. The 'if' direction follows by definition of causal equivalence and trace composition. The 'only if' direction exploits Square, Rearranging, and Shortening Lemmas (Lemmas 4.1, 4.2, and 4.3). See Appendix A.6 for details. □

As already discussed, the Loop Lemma (Lemma 3.2) and causal consistency (Theorem 4.1) serve different purposes: while the former is a "local" property connecting forward and backward steps, the latter is a property of sequences of reduction steps (traces). Therefore, in a way, the properties are independent from each other. The following example illustrates this point by exhibiting a configuration for which the Loop Lemma holds but that it is not causally consistent.

Example 4.3. Consider the following processes and types:

$$\begin{array}{llll} P_1 = z(x).z\langle \mathtt{f}(x)\rangle.\mathbf{0} & P_2 = z\langle v_1\rangle.z(x).\mathbf{0} & Q_1 = w(x).\mathbf{0} & Q_2 = w\langle \mathtt{dummy}\rangle.\mathbf{0} \\ T = ?\mathbf{int}.!\mathbf{int}.\mathtt{end} & S = !\mathbf{int}.?\mathbf{int}.\mathtt{end} & R = ?\mathbf{int}.\mathtt{end} & K = !\mathbf{int}.\mathtt{end} \end{array}$$

Moreover we indicate with type T^1 (resp. S^1) the type T (resp. S) after one step. Let M be the following configuration:

$$\begin{aligned} M = \quad & \left\langle P_1 \cdot \{z, s\} \cdot \epsilon \right\rangle_{\kappa_1 : s} \parallel s\lfloor \hat{} T \cdot \epsilon \rfloor \parallel \left\langle P_2 \cdot \{z, \overline{s}\} \cdot \epsilon \right\rangle_{\kappa_2 : \overline{s}} \parallel s\lfloor \hat{} S \cdot \epsilon \rfloor \\ & \parallel \left\langle Q_1 \cdot \{w, \overline{s}\} \cdot \epsilon \right\rangle_{\kappa_2 : \overline{s}} \parallel \overline{s}\lfloor \hat{} R \cdot \epsilon \rfloor \parallel \left\langle Q_2 \cdot \{w, s\} \cdot \epsilon \right\rangle_{\kappa_1 : s} \parallel s\lfloor \hat{} K \cdot \epsilon \rfloor \end{aligned}$$

Observe that M is not reachable because the κ_i are not pairwise distinct (cf. Definition 3.6). Consider now the following sequence of forward reductions from M:

$$\begin{aligned}
M \twoheadrightarrow\ & \left\langle z\langle \mathtt{f}(x)\rangle.\mathbf{0}\cdot\{z,s\},\{x,v_1\}\cdot z\right\rangle_{\kappa_1:s} \parallel s\lfloor T^1\cdot x\rfloor \parallel \left\langle z(x).\mathbf{0}\cdot\{z,\overline{s}\}\cdot z\right\rangle_{\kappa_2:\overline{s}} \parallel \overline{s}\lfloor S^1\cdot v_1\rfloor \\
& \parallel \left\langle Q_1\cdot\{w,\overline{s}\}\cdot\epsilon\right\rangle_{\kappa_2:\overline{s}} \parallel \overline{s}\lfloor R\hat{}\cdot\epsilon\rfloor \parallel \left\langle Q_2\cdot\{w,s\}\cdot\epsilon\right\rangle_{\kappa_1:s} \parallel s\lfloor\hat{}K\cdot\epsilon\rfloor = M_1 \\
\twoheadrightarrow\ & \left\langle \mathbf{0}\cdot\{z,s\},\{x,v_1\}\cdot z,z\right\rangle_{\kappa_1:s} \parallel s\lfloor T\hat{}\cdot x,\mathtt{f}(x)\rfloor \parallel \left\langle z(x).\mathbf{0}\cdot\{z,\overline{s}\}\cdot z\right\rangle_{\kappa_2:\overline{s}} \parallel \overline{s}\lfloor S^1\cdot v_1\rfloor \\
& \parallel \left\langle \mathbf{0}\cdot\{w,\overline{s}\},\{x,\mathtt{f}(v_1)\}\cdot w\right\rangle_{\kappa_2:\overline{s}} \parallel \overline{s}\lfloor R\hat{}\cdot x\rfloor \parallel \left\langle Q_2\cdot\{w,s\}\cdot\epsilon\right\rangle_{\kappa_1:s} \parallel s\lfloor\hat{}K\cdot\epsilon\rfloor = M_2 \\
\twoheadrightarrow\ & \left\langle \mathbf{0}\cdot\{z,s\},\{x,v_1\}\cdot z,z\right\rangle_{\kappa_1:s} \parallel s\lfloor T\hat{}\cdot x,\mathtt{f}(x)\rfloor \parallel \left\langle \mathbf{0}\cdot\{z,\overline{s}\}\cdot z,z\right\rangle_{\kappa_2:\overline{s}} \parallel \overline{s}\lfloor S\hat{}\cdot v_1,x\rfloor \\
& \parallel \left\langle \mathbf{0}\cdot\{w,\overline{s}\},\{x,\mathtt{f}(v_1)\}\cdot w\right\rangle_{\kappa_2:\overline{s}} \parallel \overline{s}\lfloor R\hat{}\cdot x\rfloor \parallel \left\langle \mathbf{0}\cdot\{w,s\}\cdot w\right\rangle_{\kappa_1:s} \parallel s\lfloor K\hat{}\cdot\mathtt{dummy}\rfloor = M_3
\end{aligned}$$

The Loop Lemma holds for each of the forward reductions above, for they can be undone: we have $M \twoheadrightarrow M_1 \rightsquigarrow M$, $M_1 \twoheadrightarrow M_2 \rightsquigarrow M_1$, and $M_2 \twoheadrightarrow M_3 \rightsquigarrow M_2$. Let us now consider a backward step $M_3 \rightsquigarrow M_4$ with

$$\begin{aligned}
M_4 = \ & \left\langle z\langle \mathtt{f}(x)\rangle.\mathbf{0}\cdot\{z,s\},\{x,v_1\}\cdot z\right\rangle_{\kappa_1:s} \parallel s\lfloor T^1\cdot x\rfloor \parallel \left\langle z(x).\mathbf{0}\cdot\{z,\overline{s}\}\cdot z\right\rangle_{\kappa_2:\overline{s}} \parallel \overline{s}\lfloor S^1\cdot v_1\rfloor \\
& \parallel \left\langle \mathbf{0}\cdot\{w,\overline{s}\},\{x,\mathtt{f}(v_1)\}\cdot w\right\rangle_{\kappa_2:\overline{s}} \parallel \overline{s}\lfloor R\hat{}\cdot x\rfloor \parallel \left\langle \mathbf{0}\cdot\{w,s\}\cdot w\right\rangle_{\kappa_1:s} \parallel s\lfloor K\hat{}\cdot\mathtt{dummy}\rfloor
\end{aligned}$$

All the forward reductions above have the same stamp: $\eta = \{\kappa_1 : s, \kappa_2 : \overline{s}\}$. Hence $M \xrightarrow{\rho} M_2 \xrightarrow{\eta} M_3$ and $M \xrightarrow{\rho_1} M_4$, with $\rho = \eta;\eta$ and $\rho_1 = \rho;\eta;\eta_\bullet$. Now, by definition of $\asymp$ (Definition 4.8) we have that $\rho \asymp \rho_1$; thus, by Theorem 4.1 these two traces must be coinitial and cofinal. However, although ρ and ρ_1 are indeed coinitial, they are not cofinal—a contradiction to the causal consistency theorem.

5. Controlled Reversibility

A natural issue when defining reversible semantics is how to control reversible actions. Indeed, we are typically interested in processes in which some actions (but not all) are reversible, and/or in which reversibility capabilities are limited. This observation is also relevant when considering session-based protocols, in which communication actions are subject to conditions such as linearity (i.e., each action must be performed exactly once) in order to ensure protocol correctness and avoid mismatches. In operational frameworks such as the one in Section 3, however, reversibility is *uncontrolled*: we could clearly perform and reverse communication steps *ad infinitum*, which may be in contrast with the disciplined behavior expected of a session once it is established.

Here we explore an extension of the model in Section 3 intended to control reversible actions. Following the same spirit than in Section 3, rather than modifying the process syntax in order to limit reversibility via dedicated primitive constructs (as in, e.g., [29]), we enrich session types in monitors with *reversibility modes* that describe the reversibility capabilities of the processes governed by such types. By controlling reversibility via monitors, we obtain a simple generalization of the framework in Section 3. We describe this extension, and discuss how the main relevant properties (namely, Loop Lemma and Causal Consistency) carry over to this extended framework.

$$
\begin{array}{lrcl}
\text{(\textit{reversibility modes})} & \mathsf{r} & ::= & 0 \mid 1 \mid \infty \\
\text{(\textit{actions})} & \alpha, \beta & ::= & !^{\mathsf{r}}U \mid ?^{\mathsf{r}}U \\
\text{(\textit{session types})} & S, T & ::= & \mathtt{end} \mid \alpha.S \mid \oplus\{l_1^{\mathsf{r}_1}{:}S_1\,,\, l_2^{\mathsf{r}_2}{:}S_2\} \mid \&\{l_1^{\mathsf{r}_1}{:}S_1\,,\, l_2^{\mathsf{r}_2}{:}S_2\} \\
\text{(\textit{history types})} & H, K & ::= & \hat{}\,S \mid S\hat{} \mid \alpha_1.\cdots.\alpha_n.\hat{}\,S \mid \oplus\{l_i^{\mathsf{r}_i}{:}S_i\,,\, l_j^{\mathsf{r}_j}{:}H_j\} \mid \&\{l_i^{\mathsf{r}_i}{:}S_i\,,\, l_j^{\mathsf{r}_j}{:}H_j\}
\end{array}
$$

Figure 6: Session types with reversibility modes. All other elements (processes, configurations) are as in Figure 1.

5.1. Reversibility Modes

In order to control reversibility via types, we annotate session types with reversibility modes, ranged over $\mathsf{r}, \mathsf{r}', \ldots$. Attached to individual communication actions described by types, reversibility modes describe reversibility capabilities:

- $\mathsf{r} = 0$ is the reversibility mode of an action that cannot be reversed (an *irreversible action*);
- $\mathsf{r} = 1$ is the reversibility mode of an action that can be reversed at most once;
- $\mathsf{r} = \infty$ is the reversibility mode of an action which can be arbitrarily reversed.

Figure 6 describes required modifications in the syntax; all other elements are as in Figure 1. In particular, we retain the syntax of processes given before: we stress that our interest is in controlling reversibility in an orthogonal way, keeping the same language of processes.

The rules of the extended forward reduction semantics, denoted $\twoheadrightarrow_{\mathsf{m}}^{\mathsf{r}}$ (where r is a reversibility mode), are given in Figure 7. The rules rely on the following (simple) extension of duality (Definition 3.1) to session types with modes:

$$
\begin{array}{l}
\overline{!U^{\mathsf{r}}.S} = ?U^{\mathsf{r}}.\overline{S} \qquad \overline{?U^{\mathsf{r}}.S} = !U^{\mathsf{r}}.\overline{S} \qquad \overline{\mathtt{end}} = \mathtt{end} \\
\overline{\oplus\{l_1^{\mathsf{r}_1} : S_1\,,\, l_2^{\mathsf{r}_2} : S_2\}} = \&\{l_1^{\mathsf{r}_1} : \overline{S_1}\,,\, l_2^{\mathsf{r}_2} : \overline{S_2}\} \\
\overline{\&\{l_1^{\mathsf{r}_1} : S_1\,,\, l_2^{\mathsf{r}_2} : S_2\}} = \oplus\{l_1^{\mathsf{r}_1} : \overline{S_1}\,,\, l_2^{\mathsf{r}_2} : \overline{S_2}\}
\end{array}
$$

That is, we assume that dual actions in types should have the same reversibility mode; an alternative formulation, with different modes in dual actions, is discussed below.

We also assume the expected extension of type and choice contexts (Definitions 3.3 and 3.4). The rules in Figure 7 essentially enrich those in Figure 3 with reversibility modes. Notice that the reversibility modes of the types involved in a forward synchronization action need to match; this choice is for simplicity: we leave for future work the definition of more flexible forms of type compatibility in which non identical reversibility modes are admitted.

Intuitively, by writing $M \twoheadrightarrow_{\mathsf{m}}^{\mathsf{r}} N$ we express that the (forward) action from M that results into N can be reversed depending on mode r, which is recorded together with its associated session type

with cursor. Observe that session establishment can always be reversed; its associated reversibility mode is ∞. Indeed, we decree that the control of reversibility is enforced *within a session*, i.e., once the session is established. This decision is in line with the intrinsically non-deterministic character of session establishment in session-based concurrency (as opposed to the linear behavior that is to be enforced once the session is established).

The most interesting part of the extension is captured by the extended rules for backward reductions, denoted $\rightsquigarrow_{\mathsf{m}}$, which are given in Figure 8. Following the previous discussion, a session establishment can always be reversed: Rule (MOPEN)* is essentially as Rule (OPEN)* in Figure 4. The purpose and effect of reversibility modes is captured by Rules (MCOM)* and (MCHOICE)*, which are enabled only when the reversibility mode of the last action in the monitor is different from 0: condition $(\mathsf{r} = \infty \Rightarrow \mathsf{r}' = \infty \wedge \mathsf{r} = 1 \Rightarrow \mathsf{r}' = 0)$, present in both rules, enables to "consume" the reversibility mode, and to obtain the new mode r' that will hold after the action has been reversed. And this condition is the only difference with respect rules of Figure 4. This is how performing a reversible action may reduce the potential for future actions.

It is instructive to briefly discuss the possibility of more general definitions of duality in the presence of reversibility modes. As a simple illustration, consider the session types:

$$S_1 = \, !{U_1}^{\mathsf{r}_1}.?{U_2}^{\mathsf{r}_3}.\mathtt{end} \qquad S_2 = \, ?{U_1}^{\mathsf{r}_2}.!{U_2}^{\mathsf{r}_4}.\mathtt{end}$$

As already discussed, our definition of duality relates S_1 and S_2 only when $\mathsf{r}_1 = \mathsf{r}_2$ and $\mathsf{r}_3 = \mathsf{r}_4$. A more flexible definition would also validate cases in which $\mathsf{r}_1 \neq \mathsf{r}_2$ and/or $\mathsf{r}_3 \neq \mathsf{r}_4$; therefore, types such as the following would be related by a more flexible duality:

$$S_1' = \, !{U_1}^{0}.?{U_2}^{\infty}.\mathtt{end} \qquad S_2' = \, ?{U_1}^{1}.!{U_2}^{1}.\mathtt{end}$$

This additional flexibility should be implemented in the premises of the backward reduction rules, which would need to account for the *less permissive* of the two reversibility modes. To see this, consider the interaction of processes implementing S_1' and S_2': a forward reduction corresponding to the exchange of the value of type U_1 cannot be reversed, because the sending action in S_1' cannot be reversed (this is consistent with $?{U_1}^1$ in S_2', because it can be reversed *at most* once). Similarly, the forward reduction corresponding to the passing of value of type U_2 can be reversed at most once, even if the reversibility mode of the receiving action in S_1' is ∞.

5.2. Example

One advantage of using types for controlling reversibility is that the same process specification can exhibit different behaviors by *programming* its reversibility modes in a different way. To illustrate this aspect, we revisit the example of Section 3.3, now enriched with a second seller, dubbed SellerB. Processes SellerB and Seller below have the same specification as the one in Section 3.3, but their type R_a differs in its reversibility modes. To lighten up notations, we omit

(MOPEN)

$$\frac{\mathsf{dual}(S,T) \qquad \overline{s} \notin \delta \qquad s \notin \delta' \qquad \sigma_1(u) = \sigma_2(u')}{\begin{array}{c}\left\langle \mathbb{K}[u\langle x:S\rangle.P]\cdot\sigma_1\cdot\widetilde{u}_1\right\rangle_{\delta} \parallel \left\langle \mathbb{H}[u'(y:T).Q]\cdot\sigma_2\cdot\widetilde{u}_2\right\rangle_{\delta'} \quad \twoheadrightarrow^{\infty}_{\mathsf{m}} \\ (\nu s,\overline{s}).\left(\left\langle \mathbb{K}[P]\cdot\sigma_1[x\mapsto\overline{s}]\cdot\widetilde{u}_1, u\right\rangle_{\delta,\overline{s}} \parallel \overline{s}\lfloor\hat{}\,S\cdot x\rfloor \parallel \left\langle \mathbb{H}[Q]\cdot\sigma_2[y\mapsto s]\cdot\widetilde{u}_2, u'\right\rangle_{\delta',s} \parallel s\lfloor\hat{}\,T\cdot y\rfloor\right)\end{array}}$$

(MCOM)

$$\frac{\sigma_1(k) = s \qquad s \in \delta \qquad \sigma_2(k') = \overline{s} \qquad \overline{s} \in \delta' \qquad \sigma_2(e) = v}{\begin{array}{c}\left\langle \mathbb{K}[k(x).P]\cdot\sigma_1\cdot\widetilde{n}\right\rangle_{\delta} \parallel \left\langle \mathbb{H}[k'\langle e\rangle.Q]\cdot\sigma_2\cdot\widetilde{m}\right\rangle_{\delta'} \parallel s\lfloor\mathbb{T}[\hat{}\,?^{\mathsf{r}}U.S]\cdot\widetilde{e}_1\rfloor \parallel \overline{s}\lfloor\mathbb{S}[\hat{}\,!^{\mathsf{r}}U.T]\cdot\widetilde{e}_2\rfloor \\ \twoheadrightarrow^{\mathsf{r}}_{\mathsf{m}} \\ \left\langle \mathbb{K}[P]\cdot\sigma_1[x\mapsto v]\cdot\widetilde{n}, k\right\rangle_{\delta} \parallel \left\langle \mathbb{H}[Q]\cdot\sigma_2\cdot\widetilde{m}, k'\right\rangle_{\delta'} \parallel s\lfloor\mathbb{T}[?^{\mathsf{r}}U.\hat{}\,S]\cdot\widetilde{e}_1, x\rfloor \parallel \overline{s}\lfloor\mathbb{S}[!^{\mathsf{r}}U.\hat{}\,T]\cdot\widetilde{e}_2, e\rfloor\end{array}}$$

(MCHOICE)

$$\frac{\sigma_1(k) = s \qquad s \in \delta \qquad \sigma_2(k') = \overline{s} \qquad \overline{s} \in \delta' \qquad \mathbb{H}'[\bullet] = \mathbb{H}[k' \triangleright \{l_j^{\mathsf{r}_j} : \langle Q_j\rangle\,,\, l_i^{\mathsf{r}_i} : \bullet\}]}{\begin{array}{c}\left\langle \mathbb{K}[k \triangleleft l_i.P]\cdot\sigma_1\cdot\widetilde{n}\right\rangle_{\delta} \parallel \left\langle \mathbb{H}[k' \triangleright \{l_j : Q_j\,,\, l_i : Q_i\}]\cdot\sigma_2\cdot\widetilde{m}\right\rangle_{\delta'} \parallel \\ s\lfloor\mathbb{T}[\hat{}\,\oplus\{l_j^{\mathsf{r}_j}{:}S_j\,,\, l_i^{\mathsf{r}_i}{:}S_i\}]\cdot\widetilde{e}_1\rfloor \parallel \overline{s}\lfloor\mathbb{S}[\hat{}\,\&\{l_j^{\mathsf{r}_j}{:}T_j\,,\, l_i^{\mathsf{r}_i}{:}T_i\}]\cdot\widetilde{e}_2\rfloor \quad \twoheadrightarrow^{\mathsf{r}_i}_{\mathsf{m}} \\ \left\langle \mathbb{K}[P]\cdot\sigma_1\cdot\widetilde{n}, k\right\rangle_{\delta} \parallel \left\langle \mathbb{H}'[Q_i]\cdot\sigma_2\cdot\widetilde{m}, k'\right\rangle_{\delta'} \parallel \\ s\lfloor\mathbb{T}[\oplus\{l_j^{\mathsf{r}_j}{:}S_j, l_i^{\mathsf{r}_i}{:}\hat{}\,S_i\}]\cdot\widetilde{e}_1, l_i\rfloor \parallel \overline{s}\lfloor\mathbb{S}[\&\{l_j^{\mathsf{r}_j}{:}T_j, l_i^{\mathsf{r}_i}{:}\hat{}\,T_i\}]\cdot\widetilde{e}_2\rfloor\end{array}}$$

Figure 7: Operational Semantics: Forward Reduction Semantics with Modalities ($\twoheadrightarrow^{\mathsf{r}}_{\mathsf{m}}$).

the mode decorations when $\mathsf{r} = \infty$:

$$\begin{aligned} &R_a : ?\mathbf{str}.!\mathbf{int}.\&\{ok^0{:}?\mathbf{str}.?\mathbf{int}.!\mathbf{cal}.\mathtt{end}\,,\, quit{:}\mathtt{end}\} \\ \textsc{SellerB} \triangleq\ &a(z:R_a).z(title).z\langle\mathtt{quote}(title)\rangle. \\ &\quad z \triangleright \{ok{:}z(addr).z(payment).z\langle\mathtt{date}(addr)\rangle.\mathbf{0}\,,\, quit{:}\mathbf{0}\} \end{aligned}$$

In this example, we assume Buyer is a service with type T'_a, where:

$$T'_a : !\mathbf{str}.?\mathbf{int}.\oplus\{ok^0{:}!\mathbf{str}.!\mathbf{int}.?\mathbf{cal}.\mathtt{end}\,,\, quit{:}\mathtt{end}\}$$

That is, T'_a is as T_a in Section 3.3 except for the reversibility mode, which is adjusted to be dual to R_a. Therefore, the difference with the protocol in Section 3.3 is that now Buyer, once he decides he is fine with the provided price he can no longer revert his choice. That is, choosing ok can be seen as a *commitment* between Buyer and Seller (or SellerB). If Buyer does not like the price and chooses the "quit" branch, then he can always go back to the state prior to the choice and ask for a better quote from another seller. More formally, we extend the system specification with a process named SellerB, with both Seller and SellerB having type R_a:

$$\textsc{System1} \triangleq \left\langle \textsc{Buyer}\cdot\epsilon\cdot\epsilon\right\rangle_{\kappa_1:\epsilon} \parallel \left\langle \textsc{Seller}\cdot\epsilon\cdot\epsilon\right\rangle_{\kappa_2:\epsilon} \parallel \left\langle \textsc{Friend}\cdot\epsilon\cdot\epsilon\right\rangle_{\kappa_3:\epsilon} \parallel \left\langle \textsc{SellerB}\cdot\epsilon\cdot\epsilon\right\rangle_{\kappa_4:\epsilon}$$

(MOPEN*)

$$\frac{\mathsf{dual}(S,T) \qquad \sigma_1(x)=\overline{s} \qquad \sigma_2(y)=s \qquad \sigma_1(u)=\sigma_2(u')}{(\nu s,\overline{s}).\left(\left\langle \mathbb{K}_1[P]\cdot\sigma_1\cdot\widetilde{u}_1,u\right\rangle_{\delta,\overline{s}} \parallel \overline{s}\lfloor\hat{}\,S\cdot x\rfloor \parallel \left\langle \mathbb{K}_2[Q]\cdot\sigma_2\cdot\widetilde{u}_2,u'\right\rangle_{\delta',s} \parallel s\lfloor\hat{}\,T\cdot y\rfloor\right) \rightsquigarrow_{\mathsf{m}} \left\langle \mathbb{K}_1[u\langle x:S\rangle.P]\cdot\sigma_1\setminus x\cdot\widetilde{u}_1\right\rangle_{\delta} \parallel \left\langle \mathbb{K}_2[u'(y:T).Q]\cdot\sigma_2\setminus y\cdot\widetilde{u}_2\right\rangle_{\delta'}}$$

(MCOM*)

$$\frac{\begin{array}{c}\sigma_1(k)=s \qquad s\in\delta \qquad \sigma_2(k')=\overline{s} \qquad \overline{s}\in\delta' \\ \mathsf{r}>0 \qquad (\mathsf{r}=\infty\Rightarrow\mathsf{r}'=\infty\wedge\mathsf{r}=1\Rightarrow\mathsf{r}'=0)\end{array}}{\begin{array}{c}\left\langle \mathbb{K}_1[P]\cdot\sigma_1\cdot\widetilde{n},k\right\rangle_{\delta} \parallel \left\langle \mathbb{K}_2[Q]\cdot\sigma_2\cdot\widetilde{m},k'\right\rangle_{\delta'} \parallel s\lfloor\mathbb{T}_1[?^{\mathsf{r}}U.\hat{}\,S_1]\cdot\widetilde{e}_1,x\rfloor \parallel \overline{s}\lfloor\mathbb{T}_2[!^{\mathsf{r}}U.\hat{}\,S_2]\cdot\widetilde{e}_2,e\rfloor \\ \rightsquigarrow_{\mathsf{m}} \\ \left\langle \mathbb{K}_1[k(x).P]\cdot\sigma_1\setminus x\cdot\widetilde{n}\right\rangle_{\delta} \parallel \left\langle \mathbb{K}_2[k\langle e\rangle.Q]\cdot\sigma_2\cdot\widetilde{m}\right\rangle_{\delta'} \parallel s\lfloor\mathbb{T}_1[\hat{}\,?^{\mathsf{r}'}U.S_1]\cdot\widetilde{e}_1\rfloor \parallel \overline{s}\lfloor\mathbb{T}_2[\hat{}\,!^{\mathsf{r}'}U.S_2]\cdot\widetilde{e}_2\rfloor\end{array}}$$

(MCHOICE*)

$$\frac{\begin{array}{c}\sigma_1(k)=s \qquad s\in\delta \qquad \sigma_2(k')=\overline{s} \qquad \overline{s}\in\delta' \qquad \mathbb{H}'[\bullet]=\mathbb{H}[k'\triangleright\{l_j^{\mathsf{r}_j}:\langle Q_j\rangle\,,\,l_i^{\mathsf{r}_i}:\bullet\}] \\ \mathsf{r}_i>0 \qquad (\mathsf{r}_i=\infty\Rightarrow\mathsf{r}_i'=\infty\wedge\mathsf{r}_i=1\Rightarrow\mathsf{r}_i'=0)\end{array}}{\begin{array}{c}\left\langle \mathbb{K}[P]\cdot\sigma_1\cdot\widetilde{n},k\right\rangle_{\delta} \parallel \left\langle \mathbb{H}'[Q_i]\cdot\sigma_2\cdot\widetilde{m}\right\rangle_{\delta'} \parallel \\ s\lfloor\mathbb{T}[\oplus\{l_j^{\mathsf{r}_j}:S_j,l_i^{\mathsf{r}_i}:\hat{}\,S_i\}]\cdot\widetilde{e}_1,l_i\rfloor \parallel \overline{s}\lfloor\mathbb{S}[\&\{l_j^{\mathsf{r}_j}:T_j,l_i^{\mathsf{r}_i}:\hat{}\,T_i\}]\cdot\widetilde{e}_2\rfloor \qquad \rightsquigarrow_{\mathsf{m}} \\ \left\langle \mathbb{K}[k\triangleleft l_i.P]\cdot\sigma_1\cdot\widetilde{n}\right\rangle_{\delta} \parallel \left\langle \mathbb{H}[k'\triangleright\{l_j:Q_j\,,\,l_i:Q_i\}]\cdot\sigma_2\cdot\widetilde{m}\right\rangle_{\delta'} \parallel \\ s\lfloor\mathbb{T}[\hat{}\,\oplus\{l_j^{\mathsf{r}_j}:S_j\,,\,l_i^{\mathsf{r}_i'}:S_i\}]\cdot\widetilde{e}_1\rfloor \parallel \overline{s}\lfloor\mathbb{S}[\hat{}\,\&\{l_j^{\mathsf{r}_j}:T_j\,,\,l_i^{\mathsf{r}_i'}:T_i\}]\cdot\widetilde{e}_2\rfloor\end{array}}$$

Figure 8: Operational Semantics: Backward Reduction Semantics with modalities ($\rightsquigarrow_{\mathsf{m}}$).

where Seller, Buyer, and Friend are as in Section 3.3. We now illustrate the behavior that results from SYSTEM1: as before, we indicate with T_a the dual types of T_b. Moreover, we have that R_a and T_a are dual.

After having established the session with Seller, Buyer receives the quote for the requested book:

$$\begin{aligned}\text{SYSTEM1} \twoheadrightarrow_m^{\infty^*} (\nu s,\overline{s}).\Big(&\left\langle \text{BUYER}_3\cdot(\{z,s\},\{prc,10\})\cdot a,z,z\right\rangle_{\kappa 1:s} \parallel \\ &s\lfloor?\mathtt{int}\hat{}\,T'_{a_2}\cdot z,\text{“}dune\text{”},prc\rfloor \parallel \\ &\left\langle \text{SELLER}_3\cdot(\{z,\overline{s}\},\{title,\text{“}dune\text{”}\})\cdot a,z,z\right\rangle_{\kappa 2:\overline{s}} \parallel \\ &\overline{s}\lfloor?\mathtt{str}\hat{}\,R_{a_2}\cdot z,title,\mathtt{quote}(title)\rfloor \parallel \left\langle \text{FRIEND}\cdot\epsilon\cdot\epsilon\right\rangle_{\kappa 3:\epsilon} \parallel \\ &\left\langle \text{SELLERB}\cdot\epsilon\cdot\epsilon\right\rangle_{\kappa 4:\epsilon}\Big) = M \end{aligned} \tag{12}$$

Buyer now can revert the entire session (e.g., if he wants a better quote for the book) and get back

to the initial configuration. From there he can now interact with SellerB hoping for a better price:

$$M \rightsquigarrow_m^* \twoheadrightarrow_m^{\infty^*} (\nu t, \bar{t}).\Big(\big\langle \textsc{Buyer}_3 \cdot (\{z,t\},\{prc,8\}) \cdot a, z, z\big\rangle_{\kappa_1:t} \parallel t\lfloor ?\mathtt{int}\,\hat{}\,T'_{a_2} \cdot z, \text{“}dune\text{”}, prc\rfloor \parallel \big\langle \textsc{SellerB}_3 \cdot (\{z,\bar{t}\},\{title, \text{“}dune\text{”}\}) \cdot a, z, z\big\rangle_{\kappa_4:\bar{t}} \parallel \bar{t}\lfloor ?\mathtt{str}\,\hat{}\,R_{a_2} \cdot z, title, \mathtt{quote}(title)\rfloor \parallel \big\langle \textsc{Friend} \cdot \epsilon \cdot \epsilon\big\rangle_{\kappa_3:\epsilon} \parallel \big\langle \textsc{Seller} \cdot \epsilon \cdot \epsilon\big\rangle_{\kappa_2:\epsilon}\Big) = M_1 \tag{13}$$

From M_1, the system can always reach M (cf. (12)), and vice versa. At this point, Buyer may decide that the price is fair and then proceed along the ok branch, therefore performing an irreversible action:

$$M \twoheadrightarrow_m^0 (\nu t, \bar{t}).\Big(\big\langle \mathbb{H}[\textsc{Buyer}_{ok}] \cdot (\{z,t\},\{prc,8\}) \cdot a, z, z, z\big\rangle_{\kappa_1:t} \parallel t\lfloor \mathbb{T}[\oplus\{ok^0\!:\hat{}\,T'^{ok}_a\,,\ quit\!:\!\mathtt{end}\}] \cdot z, \text{“}dune\text{”}, prc, ok\rfloor \parallel \big\langle \mathbb{K}[\textsc{SellerB}_{ok}] \cdot (\{z,\bar{t}\},\{title, \text{“}dune\text{”}\}) \cdot a, z, z\big\rangle_{\kappa_4:\bar{t}} \parallel \bar{t}\lfloor \mathbb{S}[\&\{ok^0\!:\hat{}\,R^{ok}_a\,,\ quit\!:\!\mathtt{end}\}] \cdot z, title, \mathtt{quote}(title)\rfloor \parallel \big\langle \textsc{Friend} \cdot \epsilon \cdot \epsilon\big\rangle_{\kappa_3:\epsilon} \parallel \big\langle \textsc{Seller} \cdot \epsilon \cdot \epsilon\big\rangle_{\kappa_2:\epsilon}\Big) = M_2 \tag{14}$$

where $\textsc{Buyer}_{ok}$, $\textsc{SellerB}_{ok}$, T'^{ok}_a and R^{ok}_a are respectively the Buyer and the SellerB processes after choosing ok and their corresponding types. Moreover:

$$\mathbb{H}[\bullet] = \mathbf{if}(\mathtt{decide}(prc))\ \mathbf{then}\ \bullet\ \mathbf{else}\ \langle z \triangleleft quit\rangle \qquad \mathbb{T}[\bullet] = !\mathbf{str}.?\mathbf{int}.\bullet$$
$$\mathbb{K}[\bullet] = z \triangleright \{ok\!:\!\bullet\,,\ quit\!:\!\langle \mathbf{0}\rangle\} \qquad \mathbb{S}[\bullet] = ?\mathbf{str}.!\mathbf{int}.\bullet$$

Because of the involved reversibility mode, starting from M_2 Buyer cannot revert his decision of accepting the price offered by Seller.

5.3. Properties

We now revisit the main properties (Loop Lemma and Causal Consistency) in the light of the extended semantics with controlled reversibility. Recall that while the Loop Lemma is a local property of single reductions (forward and backward), causal consistency is a global property that pertains to sequences of reductions. Consequently, the main effect of controlling reversibility via modes is in the Loop Lemma. To formally state these effects, we require the following auxiliary definition, that erases reversibility modes from configurations:

Definition 5.1 (Erasure)**.** Let $(\cdot)^\dagger$ be the mapping on configurations and types defined as in Figure 9.

$$
\begin{array}{lll}
(\mathbf{0})^\dagger = \mathbf{0} & (!^{\mathsf{r}}U)^\dagger = !U & (\oplus\{l_1^{\mathsf{r}_1}{:}S_1\,,\, l_2^{\mathsf{r}_2}{:}S_2\})^\dagger = \oplus\{l_1{:}(S_1)^\dagger\,,\, l_2{:}(S_2)^\dagger\} \\
(\langle P \cdot \sigma \cdot \widetilde{u} \rangle_\delta)^\dagger = \langle P \cdot \sigma \cdot \widetilde{u} \rangle_\delta & (?^{\mathsf{r}}U)^\dagger = ?U & (\&\{l_1^{\mathsf{r}_1}{:}S_1\,,\, l_2^{\mathsf{r}_2}{:}S_2\})^\dagger = \&\{l_1{:}(S_1)^\dagger\,,\, l_2{:}(S_2)^\dagger\} \\
(s\lfloor H \cdot \widetilde{e} \rfloor)^\dagger = s\lfloor (H)^\dagger \cdot \widetilde{e} \rfloor & (\mathtt{end})^\dagger = \mathtt{end} & (\alpha_1.\cdots.\alpha_n.\hat{}\,S)^\dagger = (\alpha_1)^\dagger.\cdots.(\alpha_n)^\dagger.\hat{}\,(S)^\dagger \\
(\nu n.M)^\dagger = \nu n.(M)^\dagger & (\alpha.S)^\dagger = (\alpha)^\dagger.(S)^\dagger & (\oplus\{l_i^{\mathsf{r}_i}{:}S_i\,,\, l_j^{\mathsf{r}_j}{:}H_j\})^\dagger = \oplus\{l_i{:}(S_i)^\dagger\,,\, l_j{:}(H_j)^\dagger\} \\
(M \parallel N)^\dagger = (M)^\dagger \parallel (N)^\dagger & (\hat{}\,S)^\dagger = \hat{}\,(S)^\dagger & (\&\{l_i^{\mathsf{r}_i}{:}S_i\,,\, l_j^{\mathsf{r}_j}{:}H_j\})^\dagger = \&\{l_i{:}(S_i)^\dagger\,,\, l_j{:}(H_j)^\dagger\} \\
 & (S\hat{}\,)^\dagger = (S)^\dagger\hat{} &
\end{array}
$$

Figure 9: Erasure on configurations and types.

We now have a more detailed presentation of the Loop Lemma, as now we have irreversible actions as well as actions that *may become* irreversible. We assume expected extensions of notions related to configurations (initial and reachable configurations, normal forms for configurations, well-formedness conditions).

Lemma 5.1 (Controlled Loop Lemma)**.** *For any reachable configuration with reversibility modes M, N, we have:*

1. *If $M \twoheadrightarrow_{\mathsf{m}}^{\mathsf{r}} N$ and $\mathsf{r} > 0$ then $N \rightsquigarrow_{\mathsf{m}} M'$ and $(M)^\dagger = (M')^\dagger$.*
2. *If $N \rightsquigarrow_{\mathsf{m}} M$ and $M \twoheadrightarrow_{\mathsf{m}}^{\mathsf{r}} N'$ then $(N)^\dagger = (N')^\dagger$.*

The Controlled Loop Lemma allows us to observe that the extended framework indeed generalizes that in Section 3, as two particular instances arise naturally:

- When all reversibility modes in all session types in a configuration are ∞, then we recover the framework of Section 3 (uncontrolled reversibility)
- When all reversibility modes in all session types in a configuration are 0, then we recover a monitor-based semantics for ordinary session types (without reversibility)

We further notice that causal consistency and its associated properties (Square, Rearranging, and Shortening Lemmas) hold also in the framework with reversibility modes. This is because causal consistency is a property of traces, i.e., already executed sequences of reductions. Clearly, causal consistency in the extended framework developed here considers a reduced space for processes with (controlled) reversible actions. This is a pleasant and concrete advantage of exercising reversibility control using a monitored semantics, and of keeping process implementations untouched.

6. Extensions

We briefly discuss some extensions to the framework introduced here. Rather than describing them in full detail, we informally explain how our basic framework would need to be adapted to account for each of them.

Delegation. Our framework does not support delegation, i.e., the exchange of session names as communicated messages. This to simplify the presentation and to highlight the merits of our novel approach to reversible semantics using a core model. Nevertheless, delegation can be added without technical difficulties, at the price of additional forward and backward rules in the reduction semantics. We sketch the required additional rules below, omitting choice and type contexts:

(DEL)

$$\frac{\sigma_1(k) = s \qquad s \in \delta \qquad \sigma_2(k') = \overline{s} \qquad \overline{s} \in \delta' \qquad \sigma_2(e) = s_1}{\begin{array}{l}\left\langle k(x).P \cdot \sigma_1 \cdot \widetilde{u}_1 \right\rangle_{\delta} \parallel \left\langle k'\langle e\rangle.Q \cdot \sigma_2 \cdot \widetilde{u}_2 \right\rangle_{\delta'} \parallel s\lfloor \mathbb{T}[\hat{}\,?T.S_1] \cdot \widetilde{e}_1 \rfloor \parallel \overline{s}\lfloor \hat{}\,!T.S_2 \cdot \widetilde{e}_2 \rfloor \quad \twoheadrightarrow \\ \left\langle P \cdot \sigma_1[x \mapsto s_1] \cdot \widetilde{u}_1, k \right\rangle_{\delta,\, s_1} \parallel \left\langle Q \cdot \sigma_2 \cdot \widetilde{u}_2, k' \right\rangle_{\delta' \setminus s_1} \parallel s\lfloor ?T.\hat{}\,S_1 \cdot \widetilde{e}_1, x \rfloor \parallel \overline{s}\lfloor !T.\hat{}\,S_2 \cdot \widetilde{e}_2, e \rfloor\end{array}}$$

(DEL*)

$$\frac{\sigma_1(k) = s \qquad s \in \delta \qquad \sigma_2(k') = \overline{s} \qquad \overline{s} \in \delta'}{\begin{array}{l}\left\langle P \cdot \sigma_1 \cdot \widetilde{u}_1, k \right\rangle_{\delta} \parallel \left\langle Q \cdot \sigma_2 \cdot \widetilde{u}_2, k' \right\rangle_{\delta'} \parallel s\lfloor ?T.\hat{}\,S_1 \cdot \widetilde{e}_1, x \rfloor \parallel \overline{s}\lfloor !T.\hat{}\,S_2 \cdot \widetilde{e}_2, e \rfloor \quad \rightsquigarrow \\ \left\langle k(x).P \cdot \sigma_1 \setminus x \cdot \widetilde{u}_1 \right\rangle_{\delta \setminus s_1} \parallel \left\langle k\langle e\rangle.Q \cdot \sigma_2 \cdot \widetilde{u}_2 \right\rangle_{\delta',\, s_1} \parallel s\lfloor \hat{}\,?T.S_1 \cdot \widetilde{e}_1 \rfloor \parallel \overline{s}\lfloor \hat{}\,!T.S_2 \cdot \widetilde{e}_2 \rfloor\end{array}}$$

While similar to Rules COM and COM* (cf. Figures 3 and 4), the key difference in the rules for delegation is that sequences δ and δ' should be modified to reflect that a session name contained in them (s_1 in the rules) is being transferred between their respective running processes. The session types involved are also different: rather than U the communicated value is of type T.

Asynchronous Communication. Although our process model includes queues, our model considers a synchronous communication mechanism, i.e., sender and receiver should synchronize in order to communicate. In session types, asynchronous communication is usually handled using queues: the sender adds a message to the receiver's queue; later on, the receiver consumes the first message un its queue. Our semantics can be modified in order to have monitors with message queues, using *uncoordinated* session types with cursors: sender and receiver monitor types do not need to move at the same time. As a result, forward and backward communications become more fine-grained. Notice that both message ordering within a queue and the use of the subject list of a process would guarantee that the order of messages is preserved while reversing actions.

Increased Parallelism Within Configurations. Without loss of expressiveness, our model admits the parallel interaction of configurations (which include running processes) rather than that of processes themselves, i.e., individual configurations contain sequential processes. This decision simplifies the reduction semantics and the reduction stamps required to prove causal consistency. An extension with parallelism at the level of processes within configurations is certainly possible, but it would entail notational burden, and would not add expressiveness to the model.

Infinite Behavior. Our framework considers finite processes and types. Adding recursion or parameterized definitions is straightforward; we have refrained from adding infinite constructs to our model, for it obscures the required notations and essential approach. To consider recursive behaviors, we need to admit processes $\mu\, X.P$ and process variables X in the syntax of processes (Figure 1). Accordingly, the syntax of session types T would include recursive types $\mu\, Y.T$ and type variables Y. As customary, we would also require that all process and type variables are guarded by an interaction construct, and that all variables are bound. Following the approach in [30], recursive behavior can be treated at the semantic level via structural congruences at the level of processes and monitors/configurations, so as to identify a recursive definition with its unfolding (equi-recursive types). Precisely, we would need to extend $\equiv$ (Figure 2) with the following axioms:

$$\text{(E.RecP)}\ \mu\, X.P \equiv P\{{}^{\mu\, X.P}/_{X}\} \qquad \text{(E.RecT)}\ s\lfloor \mathbb{T}[\hat{}\,\mu\, Y.T] \cdot \widetilde{e}\rfloor \equiv s\lfloor \mathbb{T}[\hat{}\, T\{{}^{\mu\, Y.T}/_{Y}\}] \cdot \widetilde{e}\rfloor$$

Notice that in (E.RecT) unfolding is *guided* by the cursor $\hat{}$: it is the cursor (and consequently the computation) that indicates what is the next recursive definition to unfold. Indeed, by identifying a recursive definition with its unfolding, the syntax of history types can be kept unchanged.

7. Related Work

Our work integrates concepts and formalisms from two seemingly unrelated areas: *reversibility in concurrency* and *models of behavioral types and contracts*. Accordingly, we compare our approach and results with respect to previous works in these two research strands.

Reversibility in Concurrency. One simple but central idea in our approach is to use session types with a "cursor" (here denoted $\hat{}$) to record the current state of the protocol by "marking" its associated session type. To our knowledge, the idea of using a cursor/marking was first proposed by Boudol and Castellani [31] when defining *event transition systems* for CCS. In the context of reversible calculi, Phillips and Ulidowski [32] use a cursor to mark the past actions of CCS processes; this marking avoids resorting to extra memory information, as in the approach to reversibility advocated by Danos and Krivine in RCCS [13]. Medic and Mezzina [33] have recently shown that the approach used in RCCS is at least as expressive as the one of Phillips and Ulidowski. Cardelli and Laneve also use a cursor to formalize their *reversible structures* [34].

In [14] Phillips and Ulidowski develop a general approach to reverse a process calculus given in a particular SOS format (the *path* format); two of the main ideas of [14] are to uniquely identify each event that occurs in the system, and to make all the operators in the calculus *static*. An operator is *static* if it is preserved in the process term after an associated reduction/transition. In contrast, the operator is said to be *dynamic* if it disappears after the reduction/transition (a similar terminology is used in [31]). This way, e.g., in CCS, parallel composition is a static operator, while prefix and choice are dynamic operators. Hence, dynamic operators are more "forgetful" than static operators. Our treatment of labeled choices is inspired by [14]: we consider labelled choice as a static operator by using contexts for processes and types. One limitation of the approach in [14] is that it supports CCS-like calculi, and so there is no handling of binders. Lanese et al. [15, 35]

overcome this limitation by using memories, extending the approach of Danos and Krivine [13] in their development of the first reversible variant of the (higher-order) π-calculus. This approach has been used to give reversible semantics for programming languages [26] and more complex calculi [36]. Cristescu et al. [27] develop a reversible semantics for the π-calculus based on a labeled transition system rather than as a reduction semantics (as in [15]).

One key insight of our work is that by addressing session-based concurrency, the formal machinery required to support reversibility is simpler than in previous (untyped) reversible calculi [13, 15, 14, 27, 35] and reversible programming languages [26, 36]. This simplicity is particularly evident (and convenient) in proofs of causal consistency; it emerges clearly in the kind of memory stamps that we rely on. Indeed, since we need to keep less information on the nature and actions of session processes, our stamps are rather compact, certainly lighter than in previous approaches (for instance [15]). This simplicity is due to the fact that session linearity ensures the existence of exactly one process per endpoint; this relieves us from the need of splitting an identifier (or memory) among parallel processes, as done in several previous works. This also implies that events in the system (e.g., communications / choices / session establishments) need not be tagged with fresh identifiers. Another source of simplicity is the kind of (labeled) choice used in session-based concurrency. Rather than non deterministic choice, session processes rely on deterministic, labeled choices; divided into constructs for external and internal choice (representing selection and branching, using pairwise distinct labels), the branches of a labeled choice are in conflict by definition: two running processes cannot perform the same selection, and a given running process cannot select more than one option of a complementary offer. As a result, we are able to prove causal consistency with rather simple reduction stamps; we only need to univocally identify all the processes in the system as well as the session on which they perform their communication actions.

Controlled reversibility has been first studied by Danos and Krivine in [37], who introduce irreversible actions in RCCS to model transactions. Lanese et al. [29] control reversibility in the higher-order π-calculus via an explicit rollback operator that reverts the computation up to a particular point. Other approaches to controlled reversibility have been developed by Bacci et al. [38], using energy parameters to drive the evolution of the process; by Phillips et al. [39] who use a non-reversible controller to guide the execution of a reversible process; and, more recently, by Kuhn and Ulidowski [40], who address controlled reversibility that is local to a prefix action (as opposed to mechanisms defined externally to the process).

Models of Behavioral Types and Contracts. The interplay between reversibility and models of behavioral types and contracts has been studied in [12, 41, 42].

As already mentioned, Tiezzi and Yoshida [10, 12] were the first to investigate a reversible semantics for session-based concurrency, by adapting the approach of [15] into ReSπ, a π-calculus with binary sessions. The calculus ReSπ is equipped with three different kinds of memories (action, choice, fork); its semantics is given in terms of a reduction relation. The key properties of this semantics (Loop lemma and causal consistency) are established in this untyped setting. Then, *a posteriori*, it is shown that a standard type system for binary sessions can be reused for ReSπ, and associated subject reduction and type safety results are established. An extension of ReSπ

with committable sessions, expressed by a dedicated construct at the level of processes, is also developed. The main difference between our work and [12] is that we critically rely on session type information to enact reversibility. While untyped, processes in our setting are governed by session types included in the monitor associated to each session endpoint; our monitors can thus be seen as a uniform kind of memory. Since session types directly guide the forward and backward semantics of processes in our model, our analysis of reversibility properties (most notably, causal consistency) in explicitly accounts for the protocols specified by session types. We further show that forms of controlled reversibility can be programmed by equipping the session types in monitors with reversibility modes attached to individual communication actions.

The recent work [42] complements [12] by analyzing the cost of enforcing different forms of reversibility in single (sequential) sessions; the study covers calculi for both binary and multiparty sessions. As in [12], the analysis carried out in [42] is operational and does not consider session type information guiding the behavior of processes. It should be interesting to assess the merits of our monitor-based approach to session reversibility following the systematic study detailed in [42]. In another recent work, Dezani-Ciancaglini and Giannini develop a model of multiparty sessions with checkpoints, named points specified in a global protocol (before internal and external choices) to which computation may return [43]. This reversibility mechanism is different from ours, for rollback operations should specify the name of the checkpoint to which computation should revert.

Although intuitively similar, theories of behavioral types (such as session types) and contracts differ in aspects such as the treatment of choices and subtyping [44] (see also the survey [4]). Barbanera et al. have studied reversible/retractable contracts from different perspectives [11, 41, 45]. In [11], they study the impact of a disciplined form of backtracking in the compliance relation for language of contracts with checkpoints, and obtain alternative characterizations for it. In [41] the interplay between contracts and rollback is studied, and a decidable compliance relation is obtained. The key idea is to store the discarded branches of a choice in a log, which is used when a process is stuck and willing to rollback to its last decision point. In this way, if a branch of a choice fails, execution may continue by following one of the stored branches. Disciplining contract rollback is simpler than in [29], since contracts (as session types) describe sequential interactions. Controlling backward actions using reversibility modes is somewhat similar to the rollbacks in [41], although our framework cannot disable a branch once it has been reverted: rather than narrowing forward actions, the reversibility modes considered here concern reversible actions—they say how many times an action can be reverted. We plan to explore alternative formulations for (reversibility) modes, so as to control also forward actions and encode rollbacks as in [41]. The recent work [45] gives a game-theoretical interpretation of compliance in retractable contracts via three-party games involving a client, a server, and an orchestrator.

Our reduction semantics for session-based concurrency bears similarities with the operational semantics usually given to session calculi with asynchronous, monitored behavior. Indeed, our model builds upon a style of process semantics in which monitors (which include session types) enable and guide process behavior, formally expressed via reduction steps. To our knowledge, the first formulation of this kind was introduced by Kouzapas [16]; it has been later used for different purposes/motivations, including eventful sessions [17], asynchronous session semantics [18], and

adaptation [19, 20]. To our knowledge this is the first time that this formulation is used to support reversibility. Our semantics is also similar to that present in session frameworks with run-time verification capabilities, such as those presented in, e.g., [23, 6]. Since these works do not support reversibility, our approach may suggest enhancements for their dynamic verification capabilities.

Finally, as already hinted at in the introduction, reversible semantics for session-based concurrency can be seen as loosely related to existing process models with specific constructs for exceptions, compensations, adaptation, and transactions (see, e.g., [46, 47, 48, 8]). In a way, such constructs implement particular strategies for reversing the behavior of a process/program, usually exploiting the granularity and information given by some explicit delimiter (such as try/catch blocks or adaptation scopes). A main difference between reversible semantics and constructs such as compensation handlers is that the latter define ad-hoc pieces of code that lead the system back to a consistent state, possibly different from the ones that the process/program has already gone through; in contrast, reversible steps correspond to previously executed actions. Also, while constructs such as exceptions typically act triggered by some external stimulus (e.g., a raised exception or an adaptation request), reversibility steps are embedded in (and automatically enacted by) a program's operational semantics. Despite these conceptual differences, Lanese et al. have shown that forms of compensations and (controlled) reversibility can be fruitfully combined [49, 50].

8. Concluding Remarks and Future Work

We have proposed a fresh approach to reversible semantics for session-based concurrency. Our approach builds upon a style of process semantics in which run-time monitors include session types and enable process reductions. Even if this style of process semantics is not new—it was introduced in [16] and later used in [17, 18, 19, 20]—to our knowledge this is the first time that this formulation is used to support a reversible semantics that is causally consistent.

We rely on monitors which contain session types enhanced with descriptions of past and future structured interactions; these types offer a uniform form of memories for supporting forward and backward semantics. We motivated our approach by introducing a simple process framework with session establishment, input-output communication, and labeled choices; extensions with other usual session constructs, such as recursion, are straightforward. To highlight the simplicity of our approach, we considered binary session types [1]. Our approach should scale to account also for multiparty structured communications [2]; in such a setting, monitors would be generated after multiparty session establishment, and would be equipped with local projections of global types, as in [23, 51]. A multiparty, asynchronous semantics may need to consider forms of *coordinated* reversibility among different partners; we plan to address these challenges in future work.

Most models of reversible processes (cf. [13]) do not consider (behavioral) types, and so their reversible semantics must account for arbitrarily complex forms of concurrent behavior. In reversing the untyped π-calculus, substitutions and scope extrusion are known to be challenging issues [27, 35]. Reversing session processes is a seemingly simpler problem, as behavior is disciplined by types: once a session is established, concurrency interactions proceed in a deterministic, confluent manner. Also, in session π-calculi scope extrusion is limited. Our developments show

that explicitly considering types in the reduction semantics for session processes leads to significant simplifications in proofs of key correctness properties, such as causal consistency. Intuitively, these simplifications are due to the less amount of information related to configurations that needs to be recorded in order to establish auxiliary technical properties.

As already discussed, the work [12] is the first to address reversibility for a synchronous π-calculus with binary session types. A key difference between our work and [12] is the role that session types play in the reversible semantics. While we used session types to define forward and backward semantics, the reversible semantics in [12] establishes key results for reversibility (most notably, causal consistency) using an untyped reduction semantics. Hence, the influence of types on the reversible semantics of [12] is indirect at best.

As further topics for future work, inspired by [42] we plan to establish the precise savings involved from moving from (i) an untyped reversible semantics to (ii) a monitored reversible semantics with types, as proposed here. We also plan to compare the (untyped) reversible higher-order processes in [15] and the core higher-order session calculus in [52], which may precisely encode the first-order session π-calculus. Moreover, it should be useful to develop alternative formulations of controlled reversibility via type annotations. We believe that type annotations, in the form of advanced reversibility modes (cf. Section 5), could encode the rollback facility of [41].

Acknowledgments. We are grateful to Ilaria Castellani, Ivan Lanese, and Dimitris Kouzapas for useful exchanges and constructive criticism. We would also like to thank the anonymous reviewers and attendees of PLACES 2016 for their suggestions and feedback. Furthermore, we are grateful to Mariangiola Dezani-Ciancaglini and to the anonymous reviewers of the present paper for their useful remarks and suggestions, which led to substantial improvements.

This work was partially supported by COST Actions IC1201 (Behavioral Types for Reliable Large-Scale Software Systems), IC1402 (Runtime Verification beyond Monitoring), and IC1405 (Reversible Computation - Extending Horizons of Computing). Pérez has been partially supported by CNRS PICS project 07313 (SuCCeSS); he is also affiliated to the NOVA Laboratory for Computer Science and Informatics (NOVA LINCS - PEst/UID/CEC/04516/2013), Universidade Nova de Lisboa, Portugal.

References

[1] K. Honda, V. T. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programming, in: C. Hankin (Ed.), ESOP'98, Vol. 1381 of Lecture Notes in Computer Science, Springer, 1998, pp. 122–138. doi:10.1007/BFb0053567.

[2] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: G. C. Necula, P. Wadler (Eds.), POPL 2008, ACM, 2008, pp. 273–284. doi:10.1145/1328438.1328472.

[3] L. Caires, H. T. Vieira, Conversation types, Theor. Comput. Sci. 411 (51-52) (2010) 4399–4440. doi:10.1016/j.tcs.2010.09.010.

[4] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, G. Zavattaro, Foundations of session types and behavioural contracts, ACM Comput. Surv. 49 (1) (2016) 3:1–3:36. doi:10.1145/2873052.
URL http://doi.acm.org/10.1145/2873052

[5] L. Caires, C. Ferreira, H. T. Vieira, A process calculus analysis of compensations, in: C. Kaklamanis, F. Nielson (Eds.), TGC 2008, Vol. 5474 of Lecture Notes in Computer Science, Springer, 2009, pp. 87–103. doi:10.1007/978-3-642-00945-7_6.

[6] R. Demangeon, K. Honda, R. Hu, R. Neykova, N. Yoshida, Practical interruptible conversations: distributed dynamic verification with multiparty session types and python, Formal Methods in System Design 46 (3) (2015) 197–225. doi:10.1007/s10703-014-0218-8.

[7] S. Capecchi, E. Giachino, N. Yoshida, Global escape in multiparty sessions, Mathematical Structures in Computer Science 26 (2) (2016) 156–205. doi:10.1017/S0960129514000164.

[8] C. D. Giusto, J. A. Pérez, Disciplined structured communications with disciplined runtime adaptation, Sci. Comput. Program. 97 (2015) 235–265. doi:10.1016/j.scico.2014.04.017.

[9] L. Jia, H. Gommerstadt, F. Pfenning, Monitors and blame assignment for higher-order session types, in: POPL 2016, ACM, 2016, pp. 582–594. doi:10.1145/2837614.2837662.

[10] F. Tiezzi, N. Yoshida, Towards reversible sessions, in: A. F. Donaldson, V. T. Vasconcelos (Eds.), Proceedings 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2014, Grenoble, France, 12 April 2014., Vol. 155 of EPTCS, 2014, pp. 17–24. doi:10.4204/EPTCS.155.3.
URL http://dx.doi.org/10.4204/EPTCS.155.3

[11] F. Barbanera, M. Dezani-Ciancaglini, U. de'Liguoro, Compliance for reversible client/server interactions, in: M. Carbone (Ed.), Proceedings Third Workshop on Behavioural Types, BEAT 2014, Rome, Italy, 1st September 2014., Vol. 162 of EPTCS, 2014, pp. 35–42. doi:10.4204/EPTCS.162.5.
URL http://dx.doi.org/10.4204/EPTCS.162.5

[12] F. Tiezzi, N. Yoshida, Reversible session-based pi-calculus, J. Log. Algebr. Meth. Program. 84 (5) (2015) 684–707. doi:10.1016/j.jlamp.2015.03.004.

[13] V. Danos, J. Krivine, Reversible communicating systems, in: P. Gardner, N. Yoshida (Eds.), Proc. of CONCUR 2004, Lecture Notes in Computer Science, Springer, 2004, pp. 292–307. doi:10.1007/978-3-540-28644-8_19.

[14] I. C. C. Phillips, I. Ulidowski, Reversing algebraic process calculi, J. Log. Algebr. Program. 73 (1-2) (2007) 70–96. doi:10.1016/j.jlap.2006.11.002.
URL http://dx.doi.org/10.1016/j.jlap.2006.11.002

[15] I. Lanese, C. A. Mezzina, J.-B. Stefani, Reversing higher-order pi, in: P. Gastin, F. Laroussinie (Eds.), Proc. of CONCUR 2010, Lecture Notes in Computer Science, Springer, 2010, pp. 478–493. doi:10.1007/978-3-642-15375-4_33.

[16] D. Kouzapas, A Session Type Discipline for Event Driven Programming Models, Master's thesis, Imperial College London (September 2009).
URL http://www.doc.ic.ac.uk/teaching/distinguished-projects/2009/d.kouzapas.pdf

[17] R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, K. Honda, Type-safe eventful sessions in java, in: T. D'Hondt (Ed.), Proc. of ECOOP 2010, Vol. 6183 of Lecture Notes in Computer Science, Springer, 2010, pp. 329–353. doi:10.1007/978-3-642-14107-2_16.

[18] D. Kouzapas, N. Yoshida, K. Honda, On asynchronous session semantics, in: Proc. of FMOODS 2011 and FORTE 2011, Vol. 6722 of Lecture Notes in Computer Science, Springer, 2011, pp. 228–243. doi:10.1007/978-3-642-21461-5_15.

[19] C. D. Giusto, J. A. Pérez, An event-based approach to runtime adaptation in communication-centric systems, in: T. T. Hildebrandt, A. Ravara, J. M. van der Werf, M. Weidlich (Eds.), Web Services, Formal Methods, and Behavioral Types - 11th International Workshop, WS-FM 2014 and 12th International Workshop, WS-FM/BEAT 2015. Revised Selected Papers, Vol. 9421 of Lecture Notes in Computer Science, Springer, 2015, pp. 67–85. doi:10.1007/978-3-319-33612-1_5.
URL http://dx.doi.org/10.1007/978-3-319-33612-1_5

[20] M. Coppo, M. Dezani-Ciancaglini, B. Venneri, Self-adaptive multiparty sessions, Service Oriented Computing and Applications 9 (3-4) (2015) 249–268. doi:10.1007/s11761-014-0171-9.

[21] S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, Information flow safety in multiparty sessions, in: B. Luttik, F. Valencia (Eds.), Proc. of EXPRESS 2011, Vol. 64 of EPTCS, 2011, pp. 16–30. doi:10.4204/EPTCS.64.2.

[22] I. Castellani, M. Dezani-Ciancaglini, J. A. Pérez, Self-adaptation and secure information flow in multiparty communications, Formal Aspects of Computing 28 (4) (2016) 669–696. doi:10.1007/s00165-016-0381-3.
URL http://dx.doi.org/10.1007/s00165-016-0381-3

[23] L. Bocchi, T. Chen, R. Demangeon, K. Honda, N. Yoshida, Monitoring networks through multiparty session types, in: D. Beyer, M. Boreale (Eds.), Proc. of FMOODS/FORTE 2013,

Vol. 7892 of Lecture Notes in Computer Science, Springer, 2013, pp. 50–65. doi:10.1007/978-3-642-38592-6_5.

[24] C. A. Mezzina, J. A. Pérez, Reversible sessions using monitors, in: D. A. Orchard, N. Yoshida (Eds.), Proceedings of the Ninth workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2016, Vol. 211 of EPTCS, 2016, pp. 56–64. doi:10.4204/EPTCS.211.6.
URL http://dx.doi.org/10.4204/EPTCS.211.6

[25] C. A. Mezzina, J. A. Pérez, Reversible semantics in session-based concurrency, in: V. Bilò, A. Caruso (Eds.), Proceedings of the 17th Italian Conference on Theoretical Computer Science, Lecce, Italy, September 7-9, 2016., Vol. 1720 of CEUR Workshop Proceedings, CEUR-WS.org, 2016, pp. 221–226.
URL http://ceur-ws.org/Vol-1720

[26] M. Lienhardt, I. Lanese, C. A. Mezzina, J.-B. Stefani, A reversible abstract machine and its space overhead, in: H. Giese, G. Rosu (Eds.), Proc. of FMOODS/FORTE 2012, Lecture Notes in Computer Science, Springer, 2012, pp. 1–17. doi:10.1007/978-3-642-30793-5_1.

[27] I. Cristescu, J. Krivine, D. Varacca, A compositional semantics for the reversible p-calculus, in: Proc. of LICS2013, IEEE Computer Society, 2013, pp. 388–397. doi:10.1109/LICS.2013.45.

[28] J. Lévy, An algebraic interpretation of the *lambda beta* k-calculus; and an application of a labelled *lambda* -calculus, Theor. Comput. Sci. 2 (1) (1976) 97–114. doi:10.1016/0304-3975(76)90009-8.
URL http://dx.doi.org/10.1016/0304-3975(76)90009-8

[29] I. Lanese, C. A. Mezzina, A. Schmitt, J.-B. Stefani, Controlling reversibility in higher-order pi, in: J. Katoen, B. König (Eds.), Proc. of CONCUR 2011, Lecture Notes in Computer Science, Springer, 2011, pp. 297–311. doi:10.1007/978-3-642-23217-6_20.

[30] D. Kouzapas, N. Yoshida, R. Hu, K. Honda, On asynchronous eventful session semantics, Mathematical Structures in Computer Science 26 (2) (2016) 303–364. doi:10.1017/S096012951400019X.
URL http://dx.doi.org/10.1017/S096012951400019X

[31] G. Boudol, I. Castellani, Flow models of distributed computations: Three equivalent semantics for CCS, Inf. Comput. 114 (2) (1994) 247–314. doi:10.1006/inco.1994.1088.
URL http://dx.doi.org/10.1006/inco.1994.1088

[32] I. C. C. Phillips, I. Ulidowski, Operational semantics of reversibility in process algebra, Electr. Notes Theor. Comput. Sci. 162 (2006) 281–286. doi:10.1016/j.entcs.2005.12.

095.
URL http://dx.doi.org/10.1016/j.entcs.2005.12.095

[33] D. Medic, C. A. Mezzina, Static VS dynamic reversibility in CCS, in: S. J. Devitt, I. Lanese (Eds.), Reversible Computation - 8th International Conference, RC 2016, Vol. 9720 of Lecture Notes in Computer Science, Springer, 2016, pp. 36–51. doi:10.1007/978-3-319-40578-0_3.

[34] L. Cardelli, C. Laneve, Reversible structures, in: F. Fages (Ed.), Proc. of CMSB 2011, ACM, 2011, pp. 131–140. doi:10.1145/2037509.2037529.

[35] I. Lanese, C. A. Mezzina, J. Stefani, Reversibility in the higher-order π-calculus, Theor. Comput. Sci. 625 (2016) 25–84. doi:10.1016/j.tcs.2016.02.019.
URL http://dx.doi.org/10.1016/j.tcs.2016.02.019

[36] E. Giachino, I. Lanese, C. A. Mezzina, F. Tiezzi, Causal-consistent rollback in a tuple-based language, Journal of Logical and Algebraic Methods in Programming (2016) 1–22doi:http://dx.doi.org/10.1016/j.jlamp.2016.09.003.
URL http://www.sciencedirect.com/science/article/pii/S2352220816301109

[37] V. Danos, J. Krivine, Transactions in RCCS, in: M. Abadi, L. de Alfaro (Eds.), Proc of CONCUR 2005, 2005, pp. 398–412. doi:10.1007/11539452_31.
URL http://dx.doi.org/10.1007/11539452_31

[38] G. Bacci, V. Danos, O. Kammar, On the statistical thermodynamics of reversible communicating processes, in: A. Corradini, B. Klin, C. Cîrstea (Eds.), Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, 2011, pp. 1–18. doi:10.1007/978-3-642-22944-2_1.
URL http://dx.doi.org/10.1007/978-3-642-22944-2_1

[39] I. Phillips, I. Ulidowski, S. Yuen, A reversible process calculus and the modelling of the ERK signalling pathway, in: R. Glück, T. Yokoyama (Eds.), Reversible Computation, 4th International Workshop, RC 2012. Revised Papers, 2012, pp. 218–232. doi:10.1007/978-3-642-36315-3_18.
URL http://dx.doi.org/10.1007/978-3-642-36315-3_18

[40] S. Kuhn, I. Ulidowski, A calculus for local reversibility, in: S. J. Devitt, I. Lanese (Eds.), Reversible Computation - 8th International Conference, RC 2016,, Vol. 9720 of Lecture Notes in Computer Science, Springer, 2016, pp. 20–35.

[41] F. Barbanera, M. Dezani-Ciancaglini, I. Lanese, U. de'Liguoro, Retractable contracts, in: S. Gay, J. Alglave (Eds.), PLACES 2015, Vol. 203 of Electronic Proceedings in Theoretical Computer Science, Open Publishing Association, 2016, pp. 61–72. doi:10.4204/EPTCS.203.5.

[42] F. Tiezzi, N. Yoshida, Reversing single sessions, in: S. J. Devitt, I. Lanese (Eds.), Reversible Computation - 8th International Conference, RC 2016, Vol. 9720 of Lecture Notes in Computer Science, Springer, 2016, pp. 52–69.

[43] M. Dezani-Ciancaglini, P. Giannini, Reversible multiparty sessions with checkpoints, in: D. Gebler, K. Peters (Eds.), Proceedings Combined 23rd International Workshop on Expressiveness in Concurrency and 13th Workshop on Structural Operational Semantics, EXPRESS/SOS 2016, Québec City, Canada, 22nd August 2016., Vol. 222 of EPTCS, 2016, pp. 60–74. doi:10.4204/EPTCS.222.5.
URL http://dx.doi.org/10.4204/EPTCS.222.5

[44] C. Laneve, L. Padovani, The pairing of contracts and session types, in: P. Degano, R. D. Nicola, J. Meseguer (Eds.), Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday, Vol. 5065 of Lecture Notes in Computer Science, Springer, 2008, pp. 681–700. doi:10.1007/978-3-540-68679-8_42.
URL http://dx.doi.org/10.1007/978-3-540-68679-8_42

[45] F. Barbanera, U. de'Liguoro, A game interpretation of retractable contracts, in: A. Lluch-Lafuente, J. Proença (Eds.), Coordination Models and Languages - COORDINATION 2016, Vol. 9686 of Lecture Notes in Computer Science, Springer, 2016, pp. 18–34. doi:10.1007/978-3-319-39519-7_2.
URL http://dx.doi.org/10.1007/978-3-319-39519-7_2

[46] M. Carbone, K. Honda, N. Yoshida, Structured interactional exceptions in session types, in: F. van Breugel, M. Chechik (Eds.), Proc of CONCUR 2008, 2008, pp. 402–417. doi:10.1007/978-3-540-85361-9_32.
URL http://dx.doi.org/10.1007/978-3-540-85361-9_32

[47] E. de Vries, V. Koutavas, M. Hennessy, Communicating transactions - (extended abstract), in: P. Gastin, F. Laroussinie (Eds.), Proc. of CONCUR 2010, 2010, pp. 569–583. doi:10.1007/978-3-642-15375-4_39.
URL http://dx.doi.org/10.1007/978-3-642-15375-4_39

[48] C. Ferreira, I. Lanese, A. Ravara, H. T. Vieira, G. Zavattaro, Advanced mechanisms for service combination and transactions, in: M. Wirsing, M. M. Hölzl (Eds.), Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing, Vol. 6582 of Lecture Notes in Computer Science, Springer, 2011, pp. 302–325. doi:10.1007/978-3-642-20401-2_14.
URL http://dx.doi.org/10.1007/978-3-642-20401-2_14

[49] I. Lanese, C. A. Mezzina, J.-B. Stefani, Controlled reversibility and compensations, in: R. Glück, T. Yokoyama (Eds.), Reversible Computation, 4th International Workshop, RC 2012. Revised Papers, Vol. 7581 of Lecture Notes in Computer Science, Springer, 2013, pp. 233–240.

[50] I. Lanese, M. Lienhardt, C. A. Mezzina, A. Schmitt, J.-B. Stefani, Concurrent flexible reversibility, in: M. Felleisen, P. Gardner (Eds.), Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Vol. 7792 of Lecture Notes in Computer Science, Springer, 2013, pp. 370–390. doi:10.1007/978-3-642-37036-6.

[51] I. Castellani, M. Dezani-Ciancaglini, J. A. Pérez, Self-adaptation and secure information flow in multiparty structured communications: A unified perspective, in: BEAT 2014, Vol. 162 of EPTCS, 2014, pp. 9–18. doi:10.4204/EPTCS.162.2.

[52] D. Kouzapas, J. A. Pérez, N. Yoshida, On the relative expressiveness of higher-order session processes, in: P. Thiemann (Ed.), Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, 2016, pp. 446–475. doi:10.1007/978-3-662-49498-1_18.
URL http://dx.doi.org/10.1007/978-3-662-49498-1_18

Appendix A. Omitted Proofs

Appendix A.1. Proof of Proposition 3.2

Proposition 3.2. Any reachable configuration M is well formed.

Proof. By Definition 3.6, M is a reachable configuration if $M_0 \longrightarrow^* M$, for some initial configuration M_0. The proof is then by induction on n, the length of the reduction sequence $M_0 \longrightarrow^* M$.

In the base case $(n = 0)$ we have to show that initial configurations are well formed. We proceed to check the conditions of Definition 3.7: Condition (1) holds since in an initial configuration all running process identifiers are unique. Conditions (2), (3), (4a) and (4b) trivially hold since an initial configuration contains no monitors.

In the inductive case $(n > 0)$ we have that $M_0 \longrightarrow^* M_n \longrightarrow M$. The proof then proceeds by case analysis on the reduction $M_n \longrightarrow M$. Let us note that all the conditions in Definition 3.7 are on the running process identifiers and on the monitor (session) names. Moreover, the only rules that modify these identifiers/names are OPEN and its inverse OPEN*, so we detail the analysis for these two rules. For the remaining rules all the properties trivially hold by inductive hypothesis.

OPEN By Lemma 3.1 we have that:

$$M_n \equiv \nu\widetilde{a}.\left(\prod_{i\in I}\left\langle \mathbb{K}_i[P_i]\cdot\sigma_i\cdot\widetilde{u}_i\right\rangle_{\delta_i} \parallel \prod_{j\in J} s_j\lfloor H_j\cdot\widetilde{e}_j\rfloor\right)$$

Since $M_n \longrightarrow M$ via Rule OPEN, this implies that there exist $z, w \in I$ such that

$$P_w = u\langle x:S\rangle.P_w \qquad P_z = u'(y:T).P_z \qquad \sigma_w(u) = \sigma_z(u') \qquad \mathsf{dual}(S,T)$$
$$\overline{s}\notin\delta_w \qquad s\notin\delta_z$$

Moreover thanks to Barendregt's Variable Convention, we can assume also

$$s,\overline{s}\notin \mathtt{fn}(\mathbb{K}_w)\cup\mathtt{fn}(\mathbb{K}_z)$$

Let N stand for $\prod_{i\in I\setminus\{z,w\}}\left\langle \mathbb{K}_i[P_i]\cdot\sigma_i\cdot\widetilde{u}_i\right\rangle_{\delta_i} \parallel \prod_{j\in J} s_j\lfloor H_j\cdot\widetilde{e}_j\rfloor$. We then have:

$$M_n \equiv \nu\widetilde{a}.\Big(N \parallel \left\langle \mathbb{K}_w[u\langle x:S\rangle.P_w]\cdot\sigma_w\cdot\widetilde{u}_w\right\rangle_{\delta_w} \parallel \left\langle \mathbb{K}_z[u'(y:T).P_z]\cdot\sigma_z\cdot\widetilde{u}_z\right\rangle_{\delta_z}\Big)$$
$$\twoheadrightarrow \nu(s,\overline{s}).\nu\widetilde{a}.\Big(N \parallel \left\langle \mathbb{K}_w[P_w]\cdot\sigma_w[x\mapsto\overline{s}]\cdot\widetilde{u}_w, u\right\rangle_{\delta_w,\overline{s}} \parallel \overline{s}\lfloor\hat{}S\cdot x\rfloor$$
$$\parallel \left\langle \mathbb{K}_z[P_z]\cdot\sigma_z[y\mapsto s]\cdot\widetilde{u}_z, u'\right\rangle_{\delta_z,s} \parallel s\lfloor\hat{}T\cdot y\rfloor\Big) \equiv M$$

By inductive hypothesis Condition (1) holds for all the running processes indexed by $I\setminus\{z,w\}$ and for δ_z and δ_w. Moreover, since s and $\overline{s}$ are fresh, this condition also holds for $\delta_z, \overline{s}$

and δ_w, s. By inductive hypothesis Conditions (2) and (3) hold for all the monitors whose session endpoint index is in J. Since s and $\overline{s}$ are fresh then these conditions also hold for the newly created monitors. Similarly, by inductive hypothesis we have that Conditions (4a) and (4b) hold for all the monitored processes whose identifier is indexed by $I \setminus \{z, w\}$ and for δ_z and δ_w. Since two new monitors identified by endpoints s and $\overline{s}$ are created, then Condition (4a) is satisfied by running process identifiers $\delta_z, \overline{s}$ and δ_w, s, and since the two endpoints are fresh there is no clash with other existing ones. Hence, Condition (4b) holds.

OPEN* This case is similar to the previous one. The difference is that two endpoints (and related monitors) of a session are removed by the reduction.

□

Appendix A.2. Proof of Lemma 3.2

Lemma 3.2. Let M and N be reachable configurations. Then: $M \twoheadrightarrow N \iff N \rightsquigarrow M$.

Proof. By induction on the derivation of $M \twoheadrightarrow N$ for the if direction, and on the derivation of $N \rightsquigarrow M$ for the converse. We examine a few cases in both directions; the rest is similar.

- Rule OPEN. By Lemma 3.1 we have that:

$$M \equiv \nu \widetilde{a}.\left(\prod_{i\in I}\left\langle \mathbb{K}_i[P_i]\cdot\sigma_i\cdot\widetilde{u}_i\right\rangle_{\delta_i} \parallel \prod_{j\in J} s_j\lfloor H_j\cdot\widetilde{e}_j\rfloor\right)$$

  and since Rule OPEN is applied, then there exist indexes $w, z \in I$ such that

$$P_w = u\langle x:S\rangle.P_w \qquad P_z = u'(y:T).P_z \qquad \sigma_w(u) = \sigma_z(u') \qquad \mathsf{dual}(S,T)$$
$$\overline{s}\notin\delta_w \qquad s\notin\delta_z \qquad s,\overline{s}\notin \mathtt{fn}(\mathbb{K}_w)\cup\mathtt{fn}(\mathbb{K}_z)$$

  Let M_1 stand for $\prod_{i\in I\setminus\{w,z\}}\left\langle \mathbb{K}_i[P_i]\cdot\sigma_i\cdot\widetilde{u}_i\right\rangle_{\delta_i} \parallel \prod_{j\in J} s_j\lfloor H_j\cdot\widetilde{e}_j\rfloor$. We then have:

$$M \twoheadrightarrow \nu\widetilde{a}, s, \overline{s}.\big(\ M_1 \parallel \left\langle \mathbb{K}_w[P_w]\cdot\sigma_w[x\mapsto\overline{s}]\cdot\widetilde{u}_w, u\right\rangle_{\delta_w,\overline{s}} \parallel \left\langle \mathbb{K}_z[P_z]\cdot\sigma_z[y\mapsto s]\cdot\widetilde{u}_z, u'\right\rangle_{\delta_z,s}$$
$$\parallel s_w\lfloor \hat{}\,T\cdot x\rfloor \parallel s_z\lfloor \hat{}\,S\cdot y\rfloor\big) = M'$$

  It is easy to see that by applying Rule OPEN* from M' we get back to M, as desired.

- Rule CHOICE. By Lemma 3.1 we have that:

$$M \equiv \nu \widetilde{a}.\left(\prod_{i\in I}\left\langle \mathbb{K}_i[P_i]\cdot\sigma_i\cdot\widetilde{u}_i\right\rangle_{\delta_i} \parallel \prod_{j\in J} s_j\lfloor H_j\cdot\widetilde{e}_j\rfloor\right)$$

and since Rule CHOICE is applied, then there exist indexes $w, z \in I$ and $m, n \in J$ such that

$$P_w = k \triangleleft l_b.P \qquad P_z = k' \triangleright \{l_a : Q_a \, , \, l_b : Q_b\}$$
$$H_m = \mathbb{T}[\,\hat{}\, \oplus \{l_a{:}S_a, l_b{:}S_b\}] \qquad H_n = \mathbb{S}[\,\hat{}\, \&\{l_a{:}T_a, l_b{:}T_b\}]$$

with $\sigma_w(k) = s$, $s \in \delta_w$, $\sigma_z(k') = \overline{s}$ and $\overline{s} \in \delta_z$. Let

$$M_1 = \prod_{i \in I \setminus \{z,w\}} \left\langle \mathbb{K}_i[P_i] \cdot \sigma_i \cdot \widetilde{u}_i \right\rangle_{\delta_i} \parallel \prod_{j \in J \setminus \{m,n\}} s_j \lfloor H_j \cdot \widetilde{e}_j \rfloor$$
$$\mathbb{H}'[\bullet] = \mathbb{H}[k' \triangleright \{l_a : \langle Q_a \rangle \, , \, l_b : \bullet\}]$$

Then we have:

$$M \twoheadrightarrow \nu \widetilde{a}.\Big(M_1 \parallel \left\langle \mathbb{K}[P] \cdot \sigma_w \cdot \widetilde{u}_w, k \right\rangle_{\delta_w} \parallel \left\langle \mathbb{H}'[Q_b] \cdot \sigma_z \cdot \widetilde{u}_z \right\rangle_{\delta_z} \parallel$$
$$s \lfloor \mathbb{T}[\oplus\{l_a{:}S_a, l_b{:}\,\hat{}\, S_b\}] \cdot \widetilde{e}_1, l_b \rfloor \parallel \overline{s} \lfloor \mathbb{S}[\&\{l_a{:}T_a, l_b{:}\,\hat{}\, T_b\}] \cdot \widetilde{e}_2 \rfloor \Big) = M'$$

It is easy to see that by applying Rule CHOICE* from M' we get back to M, as desired.

- Rule COM*. By Lemma 3.1 we have that:

$$M \equiv \nu \widetilde{a}.\left(\prod_{i \in I} \left\langle \mathbb{K}_i[P_i] \cdot \sigma_i \cdot \widetilde{u}_i \right\rangle_{\delta_i} \parallel \prod_{j \in J} s_j \lfloor H_j \cdot \widetilde{e}_j \rfloor \right)$$

and since Rule COM* is applied, then there exist indexes $w, z \in I$ and $m, n \in J$ such that

$$\sigma_w(k_w) = s \quad \overline{s} \in \delta_w \quad \widetilde{u}_w = \widetilde{u}'_w, k_w \quad \widetilde{u}_z = \widetilde{u}'_z, k_z \quad \sigma_z(k_z) = s \quad \overline{s} \in \delta_z$$
$$H_m = \mathbb{T}_1[?U.\,\hat{}\, S_h] \quad H_n = \mathbb{T}_2[!U.\,\hat{}\, S_k]$$

Let

$$M_1 = \prod_{i \in I \setminus \{w,z\}} \left\langle \mathbb{K}_i[P_i] \cdot \sigma_i \cdot \widetilde{u}_i \right\rangle_{\delta_i} \parallel \prod_{j \in J \setminus \{m,n\}} s_j \lfloor H_j \cdot \widetilde{e}_j \rfloor$$

Then we have:

$$M \equiv \nu \widetilde{a}.\Big(M_1 \parallel \left\langle \mathbb{K}_w[P_w] \cdot \sigma_w \cdot \widetilde{u}'_w, k_w \right\rangle_{\delta_w} \parallel \left\langle \mathbb{K}_z[P_z] \cdot \sigma_z \cdot \widetilde{u}'_z, k_z \right\rangle_{\delta_z} \parallel$$
$$\overline{s} \lfloor \mathbb{T}_1[?U.\,\hat{}\, S_m] \cdot \widetilde{e}_m, x \rfloor \parallel s \lfloor \mathbb{T}_2[!U.\,\hat{}\, S_n] \cdot \widetilde{e}_n, e \rfloor \Big)$$

By applying Rule COM* we have that:

$$M \rightsquigarrow \nu \widetilde{a}.\Big(M_1 \parallel \left\langle \mathbb{K}_w[k_w(x).P_w] \cdot \sigma_w \setminus x \cdot \widetilde{u}'_w \right\rangle_{\delta_w} \parallel \left\langle \mathbb{K}_z[k_z\langle e \rangle.P_z] \cdot \sigma_z \cdot \widetilde{u}'_z \right\rangle_{\delta_z} \parallel$$
$$\overline{s} \lfloor \mathbb{T}_1[?U.\,\hat{}\, S_m] \cdot \widetilde{e}_m \rfloor \parallel s \lfloor \mathbb{T}_2[!U.\,\hat{}\, S_n] \cdot \widetilde{e}_n \rfloor \Big) = M'$$

It is easy to see that by applying Rule COM from M' we get back to M, as desired.

□

Appendix A.3. Proof of Lemma 4.1

Lemma 4.1 (Square Lemma)**.** If $t_1 : M \xrightarrow{\eta_1} M_1$ and $t_2 : M \xrightarrow{\eta_2} M_2$ are two coinitial concurrent reductions, then there exist two cofinal reductions $t_2/t_1 = M_1 \xrightarrow{\eta_2} N$ and $t_1/t_2 = M_2 \xrightarrow{\eta_1} N$.

Proof. By case analysis on the form of reductions t_1 and t_2. Our analysis considers different cases: both t_1 and t_2 are forward reductions; t_1 is a forward reduction and t_2 is a backward reduction; and both t_1 and t_2 are backward reductions.

t_1 **and** t_2 **forward:** We have then 9 sub-cases, corresponding to combinations of Rules (OPEN) and (OPEN), (OPEN) and (COM), (OPEN) and (CHOICE), (COM) and (OPEN), (COM) and (COM), (COM) and (CHOICE), (CHOICE) and (OPEN), (CHOICE) and (COM), (CHOICE) and (CHOICE). All the cases are similar, we will just detail the one of Rules (OPEN) and (CHOICE). We have that:

$$\begin{aligned} M \equiv \nu\widetilde{a}.\Big(& \langle \mathbb{K}_1[u(x:S).P] \cdot \sigma_1 \cdot \widetilde{u}_1 \rangle_{\delta_1} \parallel \langle \mathbb{K}_2[u'\langle y:R\rangle.Q] \cdot \sigma_2 \cdot \widetilde{u}_2 \rangle_{\delta_2} \\ & \langle \mathbb{K}_3[k \triangleleft l_i.P_i] \cdot \sigma_3 \cdot \widetilde{u}_3 \rangle_{\delta_3} \parallel \langle \mathbb{K}_4[k' \triangleright \{l_j : Q_j \,,\, l_i : Q_i\}] \cdot \sigma_4 \cdot \widetilde{u}_4 \rangle_{\delta_4} \parallel \\ & s\lfloor \mathbb{T}[\hat{} \oplus \{l_j{:}S_j \,,\, l_i{:}S_i\}] \cdot \widetilde{e}_3 \rfloor \parallel \overline{s}\lfloor \mathbb{S}[\hat{} \& \{l_j{:}T_j \,,\, l_i{:}T_i\}] \cdot \widetilde{e}_4 \rfloor \parallel M_0 \Big) \end{aligned}$$

with:

$$\sigma_1(u) = \sigma_2(u') \quad \mathsf{dual}(S,R) \quad \sigma_3(k) = s \quad \sigma_4(k') = \overline{s}$$

Since M is well formed, we have that all the δ_i ($1 \leq i \leq 4$) are pairwise disjoint. We have then $M \twoheadrightarrow M_1$ by using Rule OPEN with:

$$\begin{aligned} M_1 \equiv \nu\widetilde{a}, r, \overline{r}.\Big(& \langle \mathbb{K}_1[P] \cdot \sigma_1 \cdot \widetilde{u}_1, u \rangle_{\delta_1, r} \parallel \langle \mathbb{K}_2[Q] \cdot \sigma_2[y \mapsto \overline{r}] \cdot \widetilde{u}_2, u' \rangle_{\delta_2, \overline{r}} \parallel \\ & r\lfloor \hat{} S \cdot x \rfloor \parallel \overline{r}\lfloor \hat{} R \cdot y \rfloor \parallel \langle \mathbb{K}_3[k \triangleleft l_i.P_i] \cdot \sigma_3 \cdot \widetilde{u}_3 \rangle_{\delta_3} \parallel \\ & \langle \mathbb{K}_4[k' \triangleright \{l_j : Q_j \,,\, l_i : Q_i\}] \cdot \sigma_4 \cdot \widetilde{u}_4 \rangle_{\delta_4} \parallel \\ & s\lfloor \mathbb{T}[\hat{} \oplus \{l_j{:}S_j \,,\, l_i{:}S_i\}] \cdot \widetilde{e}_3 \rfloor \parallel \overline{s}\lfloor \mathbb{S}[\hat{} \& \{l_j{:}T_j \,,\, l_i{:}T_i\}] \cdot \widetilde{e}_4 \rfloor \parallel M_0 \Big) \end{aligned}$$

and $M \twoheadrightarrow M_2$ by using Rule CHOICE with:

$$\begin{aligned} M_2 \equiv \nu\widetilde{a}.\Big(& \langle \mathbb{K}_1[u(x:S).P] \cdot \sigma_1 \cdot \widetilde{u}_1 \rangle_{\delta_1} \parallel \langle \mathbb{K}_2[u'\langle y:R\rangle.Q] \cdot \sigma_2 \cdot \widetilde{u}_2 \rangle_{\delta_2} \\ & \langle \mathbb{K}_3[P] \cdot \sigma_3 \cdot \widetilde{u}_3, k \rangle_{\delta_3} \parallel \langle \mathbb{K}_4[k' \triangleright \{l_j : \langle Q_j \rangle \,,\, l_i : Q_i\}] \cdot \sigma_4 \cdot \widetilde{u}_4, k' \rangle_{\delta_4} \parallel \\ & s\lfloor \mathbb{T}[\oplus\{l_j{:}S_j \,,\, l_i{:}\hat{} S_i\}] \cdot \widetilde{e}_3, l_i \rfloor \parallel \overline{s}\lfloor \mathbb{S}[\&\{l_j{:}T_j \,,\, l_i{:}\hat{} T_i\}] \cdot \widetilde{e}_4 \rfloor \parallel M_0 \Big) \end{aligned}$$

Now both M_1 and M_2 reduce to:

$$N \equiv \nu\widetilde{a}, r, \overline{r}.\Big(\big\langle \mathbb{K}_1[P]\cdot\sigma_1\cdot\widetilde{u}_1, u\big\rangle_{\delta_{1,r}} \parallel \big\langle \mathbb{K}_2[Q]\cdot\sigma_2[y\mapsto\overline{r}]\cdot\widetilde{u}_2, u'\big\rangle_{\delta_{2,\overline{r}}} \parallel r\lfloor\hat{}\, S\cdot x\rfloor \parallel \overline{r}\lfloor\hat{}\, R\cdot y\rfloor \parallel \big\langle \mathbb{K}_3[P]\cdot\sigma_3\cdot\widetilde{u}_3, k\big\rangle_{\delta_3} \parallel \big\langle \mathbb{K}_4[k'\triangleright\{l_j:\langle Q_j\rangle\ ,\ l_i:Q_i\}]\cdot\sigma_4\cdot\widetilde{u}_4, k'\big\rangle_{\delta_4} \parallel s\lfloor\mathbb{T}[\oplus\{l_j{:}S_j\ ,\ l_i{:}\hat{}\, S_i\}]\cdot\widetilde{e}_3, l_i\rfloor \parallel \overline{s}\lfloor\hat{}\,\mathbb{S}[\&\{l_j{:}T_j\ ,\ l_i{:}\hat{}\, T_i\}]\cdot\widetilde{e}_4\rfloor \parallel M_0\Big)$$

which concludes the sub-case.

t_1 **forward and** t_2 **backward:** We have then 9 sub-cases, corresponding to combinations of Rules (OPEN) and (OPEN*), (OPEN) and (COM*), (OPEN) and (CHOICE*), (COM) and (OPEN*), (COM) and (COM*), (COM) and (CHOICE*), (CHOICE) and (OPEN*), (CHOICE) and (COM*), (CHOICE) and (CHOICE*). All the cases are similar, we just detail the case involving Rules (OPEN*) and (CHOICE). We have:

$$M \equiv \nu\widetilde{a}, r, \overline{r}.\Big(\big\langle \mathbb{K}_1[P]\cdot\sigma_1[x\mapsto r]\cdot\widetilde{u}_1, u\big\rangle_{\delta_{1,r}} \parallel \big\langle \mathbb{K}_2[Q]\cdot\sigma_2[y\mapsto\overline{r}]\cdot\widetilde{u}_2, u'\big\rangle_{\delta_{2,\overline{r}}} \parallel r\lfloor\hat{}\, S\cdot x\rfloor \parallel \overline{r}\lfloor\hat{}\, R\cdot y\rfloor \parallel \big\langle \mathbb{K}_3[k\triangleleft l_i.P_i]\cdot\sigma_3\cdot\widetilde{u}_3\big\rangle_{\delta_3} \parallel \big\langle \mathbb{K}_4[k'\triangleright\{l_j:Q_j\ ,\ l_i:Q_i\}]\cdot\sigma_4\cdot\widetilde{u}_4\big\rangle_{\delta_4} \parallel s\lfloor\mathbb{T}[\hat{}\,\oplus\{l_j{:}S_j\ ,\ l_i{:}S_i\}]\cdot\widetilde{e}_3\rfloor \parallel \overline{s}\lfloor\mathbb{S}[\hat{}\,\&\{l_j{:}T_j\ ,\ l_i{:}T_i\}]\cdot\widetilde{e}_4\rfloor \parallel M_0\Big)$$

with:

$$\sigma_1[x\mapsto r](u) = \sigma_2[y\mapsto\overline{r}](u') \quad \sigma_3(k) = s \quad \sigma_4(k') = \overline{s}$$

We have that $M \rightsquigarrow M_1$ by using Rule OPEN*

$$M_1 \equiv \nu\widetilde{a}.\Big(\big\langle \mathbb{K}_1[u(x:S).P]\cdot\sigma_1\cdot\widetilde{u}_1\big\rangle_{\delta_1} \parallel \big\langle \mathbb{K}_2[u'\langle y:R\rangle.Q]\cdot\sigma_2\cdot\widetilde{u}_2\big\rangle_{\delta_2} \big\langle \mathbb{K}_3[k\triangleleft l_i.P_i]\cdot\sigma_3\cdot\widetilde{u}_3\big\rangle_{\delta_3} \parallel \big\langle \mathbb{K}_4[k'\triangleright\{l_j:Q_j\ ,\ l_i:Q_i\}]\cdot\sigma_4\cdot\widetilde{u}_4\big\rangle_{\delta_4} \parallel s\lfloor\mathbb{T}[\hat{}\,\oplus\{l_j{:}S_j\ ,\ l_i{:}S_i\}]\cdot\widetilde{e}_3\rfloor \parallel \overline{s}\lfloor\mathbb{S}[\hat{}\,\&\{l_j{:}T_j\ ,\ l_i{:}T_i\}]\cdot\widetilde{e}_4\rfloor \parallel M_0\Big)$$

and $M \twoheadrightarrow M_2$ by using Rule CHOICE with:

$$M_2 \equiv \nu\widetilde{a}, r, \overline{r}.\Big(\big\langle \mathbb{K}_1[P]\cdot\sigma_1\cdot\widetilde{u}_1, u\big\rangle_{\delta_{1,r}} \parallel \big\langle \mathbb{K}_2[Q]\cdot\sigma_2[y\mapsto\overline{r}]\cdot\widetilde{u}_2, u'\big\rangle_{\delta_{2,\overline{r}}} \parallel r\lfloor\hat{}\, S\cdot x\rfloor \parallel \overline{r}\lfloor\hat{}\, R\cdot y\rfloor \parallel \big\langle \mathbb{K}_3[P]\cdot\sigma_3\cdot\widetilde{u}_3, k\big\rangle_{\delta_3} \parallel \big\langle \mathbb{K}_4[k'\triangleright\{l_j:\langle Q_j\rangle\ ,\ l_i:Q_i\}]\cdot\sigma_4\cdot\widetilde{u}_4, k'\big\rangle_{\delta_4} \parallel s\lfloor\mathbb{T}[\oplus\{l_j{:}S_j\ ,\ l_i{:}\hat{}\, S_i\}]\cdot\widetilde{e}_3, l_i\rfloor \parallel \overline{s}\lfloor\hat{}\,\mathbb{S}[\&\{l_j{:}T_j\ ,\ l_i{:}\hat{}\, T_i\}]\cdot\widetilde{e}_4\rfloor \parallel M_0\Big)$$

Now both M_1 and M_2 reduce to:

$$N \equiv \nu\widetilde{a}.\Big(\big\langle \mathbb{K}_1[u(x:S).P]\cdot\sigma_1\cdot\widetilde{u}_1\big\rangle_{\delta_1} \parallel \big\langle \mathbb{K}_2[u'\langle y:R\rangle.Q]\cdot\sigma_2\cdot\widetilde{u}_2\big\rangle_{\delta_2} \\ \big\langle \mathbb{K}_3[P]\cdot\sigma_3\cdot\widetilde{u}_3, k\big\rangle_{\delta_3} \parallel \big\langle \mathbb{K}_4[k' \triangleright \{l_j : \langle Q_j\rangle\ ,\ l_i : Q_i\}]\cdot\sigma_4\cdot\widetilde{u}_4, k'\big\rangle_{\delta_4} \parallel \\ s\lfloor \mathbb{T}[\oplus\{l_j{:}S_j\ ,\ l_i{:}\hat{}\,S_i\}]\cdot\widetilde{e}_3, l_i\rfloor \parallel \overline{s}\lfloor \mathbb{S}[\&\{l_j{:}T_j\ ,\ l_i{:}\hat{}\,T_i\}]\cdot\widetilde{e}_4\rfloor \parallel M_0\Big)$$

which concludes the sub-case.

t_1 **and** t_2 **backward:** We then have 9 sub-cases, corresponding to combinations of Rules (OPEN*) and (OPEN*), (OPEN*) and (COM*), (OPEN*) and (CHOICE*), (COM*) and (OPEN*), (COM*) and (COM*), (COM*) and (CHOICE*), (CHOICE*) and (OPEN*), (CHOICE*) and (COM*), (CHOICE*) and (CHOICE*). All the cases are similar to their forward version by just considering the arrival configuration N as the initial one and viceversa.

□

Appendix A.4. Proof of Lemma 4.2

Lemma 4.2 (Rearranging Lemma)**.** Given a trace ρ, there exist forward traces ρ' and ρ'' such that $\rho \asymp \rho'_\bullet; \rho''$.

Proof. By lexicographic induction on $\mathsf{len}(\rho)$ and on the distance between the first reduction in ρ and the earliest pair $t, t'_\bullet$ of opposing reductions in ρ with t and t' being forward reductions. If there is no such a pair then either ρ' or ρ'' are empty and we are done. If there is at least one pair then the analysis considers whether t and t' (with stamps η_1 and η_2, respectively) are concurrent or in conflict:

t **and** t' **are concurrent.** Then $\lambda(\eta_1) \cap \lambda(\eta_2) = \emptyset$. By the Square Lemma (Lemma 4.1), we know that the execution order between the two reductions is unimportant; we can therefore swap them, which results in an earliest contradicting pair occurring later in ρ. The thesis then follows by induction, since the swapping of reductions keeps $\mathsf{len}(\rho)$ unchanged.

t **and** t' **are in conflict.** Then $\lambda(\eta_1) \cap \lambda(\eta_2) \neq \emptyset$ and we consider two possibilities: either the two stamps are equal (a "total" conflict) or not (a "partial" conflict). In the following, we let $\lambda(\eta_1) = \{\kappa_1 : s_1\ ;\ \kappa_2 : \overline{s_1}\}$ and $\lambda(\eta_2) = \{\kappa_3 : s_2\ ;\ \kappa_4 : \overline{s_2}\}$.

Sub-case $\lambda(\eta_1) = \lambda(\eta_2)$: Then $t = t'$, i.e., $t'_\bullet$ undoes t. By applying Loop Lemma (Lemma 3.2) we can remove $t; t'_\bullet$ from ρ. As a result, $\mathsf{len}(\rho)$ decreases and we can conclude by induction on a shorter trace.

Sub-case $\lambda(\eta_1) \neq \lambda(\eta_2)$: Then there is a κ_i or a session name (s_i or $\overline{s_i}$) present in both stamps. We have several possibilities:

- $(\kappa_1 \in \{\kappa_3, \kappa_4\}) \vee (\kappa_2 \in \{\kappa_3, \kappa_4\})$

- $(\kappa_3 \in \{\kappa_1, \kappa_2\}) \vee (\kappa_4 \in \{\kappa_1, \kappa_2\})$
- $(s_1 \in \{s_2, \overline{s_2}\}) \vee (\overline{s_1} \in \{s_2, \overline{s_2}\})$
- $(s_2 \in \{s_1, \overline{s_1}\}) \vee (\overline{s_2} \in \{s_1, \overline{s_1}\})$

None of this possibilities can occur, due to well-formedness conditions on configurations (cf. Definition 3.7). To show this, we content ourselves with detailing two main possibilities: in the first we have a clash on the names of the named sequence; in the second we have a clash on session endpoints. The analysis for the remaining cases is similar.

Sub-case $\kappa_1 = \kappa_3$**.** Condition (1) of Definition 3.7 stipulates that running processes do not share identifiers κ_i. Therefore, exactly one monitored process bears name κ_1, which in turn implies that this monitored process first performs reduction t (on names $s_1, \overline{s_1}$) and subsequently performs a reduction (on names $s_2, \overline{s_2}$, different from $s_1, \overline{s_1}$) that undoes t'. But this is impossible, because sessions are linear and monitored processes do not contain parallel processes: this has to do with the fact that the subject used by t is put on the top of the subject list, and should be used by t' to revert the action. We thus conclude that this case never occurs.

Sub-case $s_1 = s_2$**.** Conditions (2), (3), and (4) of Definition 3.7 together ensure that for each session (along endpoints $s_i, \overline{s_i}$) there exist exactly two monitors, and that each endpoint belongs exactly to one running process identifier. This excludes the possibility that a reduction such as $t'_\bullet$ (involving monitored processes identified by κ_3, κ_4) can revert a reduction such as t, which originated in different monitored processes identified by κ_1, κ_2 (which are different from κ_3, κ_4). Hence, this case cannot occur either.

□

Appendix A.5. Proof of Lemma 4.3

Lemma 4.3 (Shortening Lemma)**.** Let ρ_1 and ρ_2 be coinitial and cofinal traces, with ρ_2 forward. Then, there exists a forward trace ρ'_1 such that $\rho'_1 \asymp \rho_1$ and $\mathsf{len}(\rho'_1) \leq \mathsf{len}(\rho_1)$.

Proof. By induction on $\mathsf{len}(\rho_1)$. If ρ_1 is a forward trace then $\rho'_1 = \rho_1$ and we are done.

Otherwise, by Rearranging Lemma (Lemma 4.2) we can write ρ_1 as $\rho_\bullet; \rho'$ (with both ρ and ρ' forward). Let $t_\bullet; t'$ be the only two opposing reductions in ρ_1, with $t_\bullet$ being the last reduction of $\rho_\bullet$, and t' being the first reduction of ρ'. Since ρ_1 and ρ_2 are coinitial and cofinal, the reduction reversed by $t_\bullet$ has to be redone by another forward reduction in ρ', otherwise this difference will remain visible since ρ_2 is forward.

Let η be the stamp of t, and let t_1 be the earliest reduction in ρ' with stamp η_1 such that $\lambda(\eta) = \lambda(\eta_1)$. By linearity of the session, and since there are no parallel processes inside a session, we know that t_1 redoes the action deleted by $t_\bullet$. That is to say, $t = t_1$.

When $t_\bullet$ and t_1 are already contiguous (i.e., $t_1 = t$) we can remove them using $\asymp$. The resulting trace is shorter, thus the thesis follows by inductive hypothesis. Otherwise, if $t_\bullet$ and t_1

are not contiguous, we can use the Square Lemma (Lemma 4.1) to swap t_1 with all of its preceding reductions, and obtain a trace in which $t_\bullet$ and t_1 are contiguous: we use the fact that all reductions in between are concurrent to t_1. To prove this latter claim, suppose that $\lambda(\eta_1) = \{\kappa_1 : \overline{s_1}, \kappa_2 : s_1\}$ and assume that there exists a forward reduction t_2 with stamp η_2 such that $\lambda(\eta_2) = \{\kappa_3 : \overline{s_2}, \kappa_4 : s_2\}$ and $\lambda(\eta_1) \cap \lambda(\eta_2) \neq \emptyset$. Since $t_1 \neq t_2$, we have several possibilities:

- $(\kappa_1 \in \{\kappa_3, \kappa_4\}) \vee (\kappa_2 \in \{\kappa_3, \kappa_4\})$
- $(\kappa_3 \in \{\kappa_1, \kappa_2\}) \vee (\kappa_4 \in \{\kappa_1, \kappa_2\})$
- $(s_1 \in \{s_2, \overline{s_2}\}) \vee (\overline{s_1} \in \{s_2, \overline{s_2}\})$
- $(s_2 \in \{s_1, \overline{s_1}\}) \vee (\overline{s_2} \in \{s_1, \overline{s_1}\})$

None of this possibilities can occur, due to well-formedness conditions on configurations (cf. Definition 3.7). We will consider just two cases, the others are similar.

$\kappa_1 = \kappa_3$**.** By well-formedness there exists only one monitored process bearing this name. This implies that the monitored process identified by κ_1 does a forward action in t_2 different from the one undone by $t_\bullet$. But this is impossible by linearity and by the fact that if in t_2 there cannot be a point of decision (e.g. a choice) since t_1 can still do the action undone by $t_\bullet$. So this case can never happen.

$\overline{s_1} = \overline{s_2}$**.** By well formedness conditions, we have that for each session $(s, \overline{s})$ there exist exactly two monitors, and that each session endpoint belongs exactly to one running process identifier. Then it is not the case that reduction t' reverts an action on session $(s, \overline{s})$ different from the one of t (unless $t = t'$).

We then conclude that all (forward) reductions between $t_\bullet$ and t_1 (such as t_2) are concurrent to t_1 and therefore they can be swapped as described earlier. □

Appendix A.6. Proof of Theorem 4.1

Theorem 4.1 (Causal Consistency)**.** Let ρ_1 and ρ_2 be two traces. $\rho_1 \asymp \rho_2$ if and only if ρ_1 and ρ_2 are coinitial and cofinal.

Proof. We first prove the 'if' direction, i.e., if $\rho_1 \asymp \rho_2$ then ρ_1 and ρ_2 are coinitial and cofinal. First, notice that if $\rho_1 \asymp \rho_2$ then it must be the case that ρ_1 can be transformed into ρ_2 (and vice versa) through $n \geq 0$ applications of the rules in Definition 4.8. We then proceed by induction on n. In the base case, $n = 0$, we have that $\rho_1 \asymp \rho_2$ by applying 0 times the rules of $\asymp$. Since $\asymp$ is an equivalence, this means that $\rho_1 = \rho_2$ which in turn implies that the traces are coinitial and cofinal. In the inductive case, we have that there exist n traces ρ^k (with $0 \leq k \leq n$) obtained as a result of applying the rules of $\asymp$ to ρ_1 exactly k times; hence, $\rho^0 = \rho_1$ and $\rho^n = \rho_2$. We then have that $\rho^{n-1} \asymp \rho_2$, i.e., traces ρ^{n-1} and ρ_2 differ in one axiom application; this means that we can decompose both traces as follows:

$$\rho^{n-1} = \rho_a; \rho'; \rho_b \qquad \rho_2 = \rho_a; \rho''; \rho_b$$

with ρ' and ρ'' differing just by one application of $\asymp$. We have then three cases, informed by Definition 4.8:

1. $\rho' = t_1; t_2/t_1$ and $\rho'' = t_2; t_1/t_2$
2. $\rho' = t; t_\bullet$ and $\rho'' = \epsilon_{\mathtt{source}(t)}$
3. $\rho' = t_\bullet; t$ and $\rho'' = \epsilon_{\mathtt{target}(t)}$

In all cases it is easy to see that ρ^{n-1} and ρ_2 are both coinitial and cofinal. By inductive hypothesis, $\rho_1 \asymp \rho^{n-1}$ implies that ρ_1 and ρ^{n-1} are coinitial and cofinal; since $\rho^{n-1} \asymp \rho_2$ implies that they are coinitial and cofinal, we can conclude that also ρ_1 and ρ_2 are coinitial and coifinal if $\rho_1 \asymp \rho_2$.

We now prove the 'only if' direction, i.e., if ρ_1 and ρ_2 are coinitial and cofinal then $\rho_1 \asymp \rho_2$. The proof is by induction on $\mathsf{len}(\rho_1) + \mathsf{len}(\rho_2)$, and on the distance between the end of ρ_1 and the earliest pair of differing reductions $t_1 : M_1 \xrightarrow{\eta_1} N_1$ and $t_2 : M_2 \xrightarrow{\eta_2} N_2$, with $t_1 \in \rho_1$ and $t_2 \in \rho_2$. If there is no such a pair, then ρ_1 and ρ_2 are equal and the theorem trivially holds. Otherwise, we consider four cases depending on the direction of t_1 and t_2. We use the Rearranging Lemma (Lemma 4.2), which ensures that any trace can be written as a composition of a backward sub-trace followed by a forward sub-trace.

t_1 **forward and** t_2 **backward:** We then infer that $\rho_1 = \rho_\bullet; t_1; \rho_1'$ and $\rho_2 = \rho_\bullet; t_2; \rho_2'$, where $\rho_\bullet$ is the common backward sub-trace and $t_1; \rho_1'$ is a forward trace. Since by hypothesis ρ_1 and ρ_2 are coinitial and cofinal, also $t_1; \rho_1'$ and $t_2; \rho_2'$ are coinitial and cofinal. By applying the Shortening Lemma (Lemma 4.3) on the sub-traces $t_1; \rho_1'$ and $t_2; \rho_2'$, we obtain that $t_2; \rho_2'$ has a shorter causally equivalent forward trace and therefore ρ_2 has a shorter causally equivalent forward trace. We can then conclude by induction.

t_2 **forward and** t_1 **backward:** This case is similar to the previous one.

t_1 **and** t_2 **forward:** By assumption $t_1 \neq t_2$. We first establish whether t_1 and t_2 are in conflict or not (i.e., they are concurrent). Let $\lambda(\eta_1) = \{\kappa_1 : \overline{s_1}, \kappa_2 : s_1\}$ and $\lambda(\eta_2) = \{\kappa_3 : \overline{s_2}, \kappa_4 : s_2\}$. Suppose that t_1 and t_2 are in conflict; this means that $\lambda(\eta_1) \cap \lambda(\eta_2) \neq \emptyset$, together with several possibilities:

- $(\kappa_1 \in \{\kappa_3, \kappa_4\}) \vee (\kappa_2 \in \{\kappa_3, \kappa_4\})$
- $(\kappa_3 \in \{\kappa_1, \kappa_2\}) \vee (\kappa_4 \in \{\kappa_1, \kappa_2\})$
- $(s_1 \in \{s_2, \overline{s_2}\}) \vee (\overline{s_1} \in \{s_2, \overline{s_2}\})$
- $(s_2 \in \{s_1, \overline{s_1}\}) \vee (\overline{s_2} \in \{s_1, \overline{s_1}\})$

We consider two representative cases, the others are similar:

$\kappa_1 = \kappa_3$**.** By well-formedness conditions only one running process bears this name. This implies that the running process identified by κ_1 contributes in parallel to both reductions, t_1 and t_2, using different names ($\overline{s_1}$ and $\overline{s_2}$). But this is impossible by sequentiality of processes, which forces t_1 and t_2 to occur in sequence, but not in parallel.

$\overline{s_1} = \overline{s_2}$**.** By well formedness conditions, for each session $(s_i, \overline{s_i})$ there exist exactly two monitors, and each endpoint belongs exactly to one running process. But this contradicts the assumption that reductions t_1 and t_2 are different.

We therefore conclude that t_1 and t_2 are not in conflict—they are concurrent. Moreover, since ρ_1 and ρ_2 are coinitial and cofinal there must exist a reduction $t_2' \in \rho_1$ equal to t_2. We have to show that t_2' is concurrent with respect to all previous reductions in ρ_1. Since all of them are forward reductions, none of them reverses the state of the monitors used by t_2'. Moreover, by well formedness conditions, we have that for each session $(s_i, \overline{s_i})$ there exist exactly two monitors, and that each endpoint belongs exactly to one running process. Since all the reductions that precede t_2' are concurrent with respect to it, we can repeatedly apply the Square Lemma (Lemma 4.1) to derive a trace causally equivalent to ρ_1 in which t_1 and t_2' are consecutive. By applying the Square Lemma one more time, we obtain a trace causally equivalent to ρ_1, with the same length, but in which the difference with ρ_2 appears later on. We can then conclude by induction.

t_1 **and** t_2 **backward:** t_1 and t_2 cannot undo the same action. Since ρ_1 and ρ_2 are coinitial and cofinal, then either (i) there is a later forward reduction in ρ_1 that cancels out t_1 (i.e., the reverse step by t_1 is "local" to ρ_1, and does not appear in ρ_2), or (ii) there is a backward reduction in ρ_2 that corresponds to t_1, for the two traces are coinitial and cofinal. We examine these two cases separately:

(i) We have that t_1 is concurrent to all the subsequent backward reductions, excepting to those operating on the same session. Each of these backward reductions of this kind has a corresponding forward reduction that deletes its effects; otherwise, its effect would be visible in ρ_2, contradicting the assumption that t_1 is eventually undone. We take the last of such backward reductions, and by using the Square Lemma, we make it the last backward one. Now, the forward reduction that reverses such a reduction is concurrent with all the preceding forward reductions. We use the Square Lemma to make it the first forward reduction. At this point, we may apply axiom $t; t_\bullet \asymp \epsilon_{\mathtt{source}(t)}$ to obtain a causally equivalent, but shorter, trace. The thesis then follows by induction.

(ii) The analysis for this case is similar to the one in which both t_1 and t_2 are forward.

□