On asynchrony and reversibility in CCS

Asynchronous communication is a fundamental feature of modern distributed systems, where messages are emitted without requiring immediate synchronization with receivers. In process calculi, this behaviour is typically modelled by separating message emission from message consumption. At the same time, reversible computation has emerged as an important paradigm for analysing concurrent systems, enabling computations to be undone while preserving causal dependencies between actions. While reversible semantics have been extensively studied for synchronous process calculi such as CCS, their integration with asynchronous communication remains largely unexplored.

In this paper we investigate the interaction between asynchrony and reversibility in the setting of CCS. We first introduce CCS$^{\mathtt{a}}$, an asynchronous variant of CCS in which output actions generate explicit message entities that can later be consumed by matching input actions. We then define $\mathtt{r}$CCS$^{\mathtt{a}}$, a reversible extension of CCS$^{\mathtt{a}}$ obtained by adapting the framework of Phillips and Ulidowski [35]. In $\mathtt{r}$CCS$^{\mathtt{a}}$, prefixes and messages are annotated with unique keys that record message emission and consumption events, allowing computations to be reversed while preserving causal dependencies. We show that the resulting reversible semantics satisfies causal consistency, ensuring that computations can be reversed exactly up to causal equivalence. The proof relies on the axiomatic framework for reversible computation proposed by Lanese et al. [27].

1 Introduction

Concurrency theory provides the foundations for understanding systems composed of multiple interacting computational entities. Over time, several formal approaches have been developed, including Petri nets, event structures, and process calculi and algebras. Among these, the Calculus of Communicating Systems (CCS) [34] is a seminal reference for reasoning about communication, synchronisation, and behavioral equivalence in distributed systems. CCS models systems as collections of sequential processes that interact by exchanging messages through communication channels. Concretely, a process ready to receive a message on channel a and then continue as P is written $a.P$, while a process ready to send a message on a and then continue as Q is written $\overline{a}.Q$. Here, a denotes an input prefix and $\overline{a}$ an output prefix; prefixes capture the idea that a process must perform the corresponding action before evolving into its continuation. Processes can be composed in parallel using the operator $\|$, so that $P \| Q$ represents the concurrent execution of P and Q. When complementary actions are enabled, they may interact via a reduction step, for instance: $a.P \| \overline{a}.Q \xrightarrow{\tau} P \| Q$, which models the synchronisation and consumption of the communication action. This reduction embodies a synchronous view of communication, where message exchange occurs atomically through rendezvous.

In contrast, real-world distributed systems typically communicate asynchronously, with messages sent without requiring immediate synchronisation with a receiver. Buffering mechanisms further enable communication to occur even when the receiver is not ready. Modern distributed programming languages embrace this paradigm. For instance, in Go ([3]), communication through channels allows processes to send messages that may be buffered and consumed later by receivers. Similarly, Erlang ([19]) relies on asynchronous message passing, where messages are placed in process mailboxes and consumed when the receiver performs a matching input. These mechanisms naturally correspond to a semantics in which message emission and message consumption are distinct events.

Submitted to:
ICE 2026

This mismatch between the synchronous semantics of CCS and the asynchronous nature of practical systems has led to the development of asynchronous process calculi. There exist two main approaches to modelling asynchrony, differing in how (non-blocking) output actions are represented [5]. In the first approach, outputs are treated as processes, typically enforced via a syntactic restriction whereby output prefixes have no continuation. This is the case, for example, in asynchronous CCS (ACCS [5, 6]) and the asynchronous π-calculus [1]. In the second approach, output prefixes are allowed to have continuations, and the operational semantics generates the corresponding message processes. This is the case in CCS with Linda-like communication primitives [7] and in asynchronous session types [17].

In this work, we follow the second approach: we retain the standard CCS prefix syntax for outputs and let the operational semantics generate explicit message processes representing messages in transit. This choice keeps the language close to the structure of programming languages while modelling asynchronous communication through the presence of explicit message entities. In particular, output actions are no longer interpreted as one half of a rendezvous, but as independent emission events that generate persistent message entities. Operationally, this corresponds to an emission rule where the output action materialises as a message process $\langle a \rangle$ that can later be consumed by a matching input. Communication is thus decomposed into two causally distinct steps: message emission and message reception. In this model, the communication medium can be understood as a "bag" of messages (output actions), waiting to be consumed by corresponding input actions. Communication is then captured by the following two reductions, representing output on the left and input on the right:

$$\overline{a}.P \xrightarrow{\tau} P \parallel \langle a \rangle \qquad\qquad a.P \parallel \langle a \rangle \xrightarrow{\tau} P \tag{1}$$

The decoupling of message emission and consumption makes it harder to analyse executions, diagnose errors, and recover from undesired states. This motivates the study of mechanisms that allow computations not only to proceed forward, but also to be reversed in a controlled manner. Reversibility has emerged as a fundamental concept in the theory of computation [2]. In concurrent systems, reversibility allows computational steps to be undone, enabling systems to move both forward and backward in their execution. This notion, known as *causal-consistent reversibility* [12], has proved useful in debugging concurrent systems [15, 16, 28], where reversing erroneous computations is more natural than replaying from scratch; in biochemical modelling [29, 18], given that many biochemical reactions are inherently reversible; in fault tolerance, where recovery requires controlled rollback to consistent states [25, 33, 37, 21]; in quantum computing, where reversibility is forced by the laws of physics [13]; and in low-energy computation inspired by thermodynamic considerations [20]. In the last decade, several reversible extensions of process calculi have been proposed, enriching classical models with mechanisms to record and undo computational history while preserving causal dependencies between actions. A seminal contribution in this direction is Reversible CCS (RCCS) by Danos and Krivine [12], which extends CCS with explicit memories that record past actions and enable undos. Phillips and Ulidowski proposed a general framework for equipping process calculi with reversibility, applicable to calculi whose operational semantics is specified in a suitable SOS format [35]. The main idea of their approach is to keep the structure of processes unchanged during execution–making the operators of the calculus *static*–and to record the information needed for reversing computations by attaching unique *keys* to communications. When this framework is instantiated for CCS, it yields CCSK (CCS with communication keys). This model is closely related to Reversible CCS (RCCS): the two calculi are equivalent in terms of labelled transition system (LTS) isomorphism [22], differing only in the way history information is recorded. Despite the significant progress in reversible process calculi [32], the integration of reversibility with asynchronous communication remains less explored, with the exception of asynchronous RCCS [9], and asynchronous higher order pi [24].

In asynchronous settings, where communication is split into distinct emission and consumption events, this decoupling introduces additional challenges for reversible semantics. In particular, reversing a computation requires restoring not only the processes involved, but also the correct configuration of messages in transit, while preserving the causal dependencies between emission and reception events. As a result, designing a reversible semantics for asynchronous communication demands a careful treatment of the information used to track causality. Existing approaches developed for synchronous calculi do not directly apply in this setting. For instance, the framework of Phillips and Ulidowski [35] relies on a fixed SOS format, which is not compatible with rules such as emission, where the syntactic structure of processes may change. On the other hand, RCCS [12] is based on memories that record complete synchronisation events, whereas asynchronous communication involves partial events that are split across emission and consumption.

In this paper, we investigate the integration of reversibility with asynchronous communication in the setting of CCS. We first introduce CCS$^{\mathtt{a}}$ an asynchronous variant of CCS in which output actions generate explicit message processes that can be consumed at a later stage. We then define a reversible extension of CCS$^{\mathtt{a}}$ enriched with key annotations that record message emission and consumption events.

The remainder of the paper is structured as follows. In Section 2 the syntax and the semantics of CCS$^{\mathtt{a}}$ are presented; while in Section 3 the syntax and semantics of $\mathtt{r}$CCS$^{\mathtt{a}}$ are presented. In Section 4, we discuss the main properties of $\mathtt{r}$CCS$^{\mathtt{a}}$, while Section 5 demonstrates how the framework can be used to model the leader election problem. Finally, Section 6 concludes the paper with a discussion of the related work.

2 Asynchronous CCS

In this section, we introduce an asynchronous variant of CCS, which we refer to as CCS$^{\mathtt{a}}$. In CCS$^{\mathtt{a}}$, message sending and reception are not synchronised [4]. Messages are transmitted through a medium until they reach their destination. Consequently, sending is non-blocking, allowing a process to send regardless of the receiver's state, whereas receiving is blocking because processes must wait for a message to arrive.

$$
\begin{array}{rl}
(\textit{Prefixes}) & \mu,\eta ::= a \mid \overline{a} \mid \tau \\
(\textit{Processes}) & P,Q ::= \mathbf{0} \mid \sum_{i\in I}\mu_i.P_i \mid P \parallel Q \mid (\nu a)P \mid A \\
(\textit{Systems}) & R,S ::= \langle a \rangle \mid P \mid R \parallel S \mid (\nu a)R
\end{array}
$$

Figure 1: CCS$^{\mathtt{a}}$ syntax

Let $\mathscr{N} = \{a,b,c,\ldots\}$ be a set of *names*, and let $\overline{\mathscr{N}} = \{\overline{a} \mid a \in \mathscr{N}\}$ be the set of its corresponding *co-names*. Elements in $\mathscr{N}$ represent input actions, while elements in $\overline{\mathscr{N}}$ represent output actions. We use $\alpha,\beta,\ldots$ to range over $\mathscr{N} \cup \overline{\mathscr{N}}$, and we assume $\overline{\overline{\alpha}} = \alpha$, calling α and $\overline{\alpha}$ *complementary* actions. We also consider a special *silent action* τ. The syntax of CCS$^{\mathtt{a}}$ is reported in Figure 1. A prefix (or action), denoted by μ,η, is an element of $\mathtt{Act} = \mathscr{N} \cup \overline{\mathscr{N}} \cup \{\tau\}$. We let $P,Q,\ldots$ range over *Processes*. The term $\mathbf{0}$ represents the idle process, while $\sum_{i\in I}\mu_i.P_i$ represents a non-deterministic choice that begins by performing some prefix μ_i and continues as P_i. We write $\mu_i.P_i$ for a singleton sum where $I = \{i\}$, and

use $\mu_1.P_1 + \sum_{i\in I\setminus\{1\}} \mu_i.P_i$ to distinguish a specific branch, assuming guarded choice is associative and commutative. The term $P \parallel Q$ represents the parallel composition of processes P and Q, while $(\nu a)P$ denotes the process P where the name a is restricted. We use A for process constants, assuming a unique definition $A \triangleq P$ for each A. These constants are employed to model recursive behaviors.

Systems, ranged over by R, S, consist of the parallel composition of sent messages (each denoted by $\langle\cdot\rangle$) and processes. Analogously to processes, we denote by $R \parallel S$ the parallel composition of systems, and by $(\nu a)R$ the system in which the name a is restricted.

We use the term *term* to refer to either a system or a process.

Definition 1 (CCS$^{\mathtt{a}}$ LTS)**.** The operational semantics of CCS$^{\mathtt{a}}$ is defined as the LTS $(\mathit{Systems}, \mathtt{Act}, \rightarrow)$ where the transition relation $\rightarrow$ is the smallest relation induced by the rules reported in Figure 2.

$$\text{(Out)}\ \overline{a}.P \xrightarrow{\tau} P \parallel \langle a\rangle \qquad \text{(Msg)}\ \langle a\rangle \xrightarrow{\overline{a}} \mathbf{0} \qquad \text{(In)}\ a.P \xrightarrow{a} P \qquad \text{(Tau)}\ \tau.P \xrightarrow{\tau} P$$

$$\text{(Sum)}\ \frac{j \in I \qquad P_j \xrightarrow{\mu} P'}{\sum_{i\in I} P_i \xrightarrow{\mu} P'} \qquad \text{(Par)}\ \frac{R \xrightarrow{\mu} R'}{R \parallel S \xrightarrow{\mu} R' \parallel S} \qquad \text{(Syn)}\ \frac{R \xrightarrow{\alpha} R' \qquad S \xrightarrow{\overline{\alpha}} S'}{R \parallel S \xrightarrow{\tau} R' \parallel S'}$$

$$\text{(Res)}\ \frac{R \xrightarrow{\mu} R' \qquad \alpha \notin \{a, \overline{a}\}}{(\nu a)R \xrightarrow{\mu} (\nu a)R'} \qquad \text{(Const)}\ \frac{A \triangleq P \qquad P \xrightarrow{\mu} P'}{A \xrightarrow{\mu} P'}$$

Figure 2: CCS$^{\mathtt{a}}$ semantics, symmetric rules for Par and Syn are omitted.

As usual, we write $S \xrightarrow{\mu} S'$ to denote a transition in $\rightarrow$. The operational rules follow those of CCS [34], with the exception of (Out) and (Msg), which handle asynchrony. Under rule (Out), the output-prefixed term $\overline{a}.P$ emits a message $\langle a\rangle$ which is placed in the communication media, which is represented as the parallel composition of the message with the continuation P. Note that this emission results in a silent action. Rule (Msg) models the consumption of a message from the communication media by a input-prefixed process $a.P$.

Example 2. *Let $R = (a.\mathbf{0} + b.\mathbf{0}) \parallel \overline{a}.\overline{b}.\mathbf{0}$. A possible execution is as follows*

$$\begin{aligned} R &\xrightarrow{\tau} (a.\mathbf{0} + b.\mathbf{0}) \parallel \overline{b}.\mathbf{0} \parallel \langle a\rangle \\ &\xrightarrow{\tau} (a.\mathbf{0} + b.\mathbf{0}) \parallel \langle a\rangle \parallel \langle b\rangle \parallel \mathbf{0} \\ &\xrightarrow{\tau} \mathbf{0} \parallel \mathbf{0} \parallel \langle b\rangle \end{aligned}$$

where the second process first emits messages a and b in succession, followed by the first process consuming a.

3 Reversible asynchronous CCS

In this section, we introduce rCCS$^{\mathtt{a}}$, a reversible variant of CCS$^{\mathtt{a}}$. The syntax of rCCS$^{\mathtt{a}}$ is presented in Figure 3. Unlike CCS$^{\mathtt{a}}$, messages and prefixes in rCCS$^{\mathtt{a}}$ may be decorated with unique *keys*. We let $\mathcal{K} = \{i, j, \ldots\}$ be the set of keys. A message $[i]\langle a\rangle$ represents an output produced by a prefix marked

with i, while $[i]\langle a\rangle[j]$ denotes a message (originally produced by i) that has been consumed by an input prefix decorated with j. Notably, consumed messages *remain* in the system as inactive processes; they act as memories, storing the computational history required to revert both the consumption and the original emission. *Marked prefixes*, denoted by $\mu[i]$, indicate that the action μ has already been executed. Let us note that now, since all the operators are static, also sums are static. This means that a choice may be composed with some marked prefix and some unmarked prefixes. This will become clear once introduced the operational semanitcs.

$$\begin{array}{rl} (Prefixes) & \mu ::= a \mid \overline{a} \mid \tau \\ (Processes) & P,Q ::= \mathbf{0} \mid \sum_{i\in I}\pi_i.P_i \mid P \parallel Q \mid (\nu a)P \mid A \\ (Systems) & R,S ::= [i]\langle a\rangle \mid [i]\langle a\rangle[j] \mid P \mid R \parallel S \mid (\nu a)R \\ (Actions) & \pi ::= \mu \mid \mu[i] \end{array}$$

Figure 3: $\mathtt{rCCS}^{\mathtt{a}}$ syntax

We remark that $\mathtt{rCCS}^{\mathtt{a}}$ generalizes $\mathrm{CCS}^{\mathtt{a}}$, in the sense that each $\mathrm{CCS}^{\mathtt{a}}$ term is also an $\mathtt{rCCS}^{\mathtt{a}}$ term; specifically, these are the $\mathtt{rCCS}^{\mathtt{a}}$ terms that do not contain keys. We call such processes *standard*, as formally defined below.

Definition 3 (System and process keys)**.** The set of keys of a $\mathtt{rCCS}^{\mathtt{a}}$ term is inductively defined as follows:

$$\begin{array}{lll} \mathtt{key}([i]\langle a\rangle) = \{i\} & \mathtt{key}([i]\langle a\rangle[j]) = \{i,j\} & \mathtt{key}(R \parallel S) = \mathtt{key}(R) \cup \mathtt{key}(S) \\ \mathtt{key}(R\backslash a) = \mathtt{key}(R) & \mathtt{key}(\sum_{i\in I} P_i) = \bigcup_{i\in I} \mathtt{key}(P_i) & \mathtt{key}(P \parallel Q) = \mathtt{key}(P) \cup \mathtt{key}(Q) \\ \mathtt{key}(\pi[i].P) = \{i\} \cup \mathtt{key}(P) & \mathtt{key}(P\backslash a) = \mathtt{key}(P) & \mathtt{key}(A) = \emptyset \end{array}$$

Definition 4 (Standard System)**.** A system R is said to be standard, written $\mathtt{std}(R)$, if it contains no key, i.e., $\mathtt{key}(R) = \emptyset$.

The operational semantics of $\mathtt{rCCS}^{\mathtt{a}}$ is defined by two transition relations: $\rightarrow$, which captures forward computation, and $\hookrightarrow$, which captures backward computation.

Definition 5. [$\mathtt{rCCS}^{\mathtt{a}}$ semantics] The operational semantics of $\mathtt{rCCS}^{\mathtt{a}}$ is formally defined as the LTS $(Systems, \mathtt{Act} \times \mathscr{K}, \mapsto)$, where $\mapsto = \rightarrow \cup \hookrightarrow$, and $\rightarrow$ and $\hookrightarrow$ are the smallest relations generated by the rules reported in Figures 4 and 5, respectively.

We begin by commenting the forward rules shown in Figure 4. All rules, except (Act), mirror those of the non-reversible language in Figure 2, but are enriched with key annotations to track causality. Moreover, executed prefixes are persistent, i.e., they are not consumed by reduction.

Rule (Out), which models message emission, uses a key i to bind the emitted message, which is tagged with i, to the prefix that produced it. Notably, the prefix is preserved by the reduction and annotated with i, indicating that it has been executed. This establishes a link between the message and its generating prefix, which is essential for reversing this computation step. Note that the used key is reflected in the transition label. Rule (Msg) governs message consumption. In contrast to synchronous reversible semantics, where causality is associated with a single communication event, here causality is distributed across two distinct steps: emission and consumption. To account for this, it is necessary to

$$\text{(Out)}\ \frac{\mathtt{std}(P)}{\overline{a}.P \xrightarrow{\tau[i]} \overline{a}[i].P \parallel [i]\langle a\rangle} \qquad \text{(Msg)}\ [i]\langle a\rangle \xrightarrow{\overline{a}[j]} [i]\langle a\rangle[j] \qquad \text{(In)}\ \frac{\mathtt{std}(P)}{a.P \xrightarrow{a[i]} a[i].P}$$

$$\text{(Tau)}\ \frac{\mathtt{std}(P)}{\tau.P \xrightarrow{\tau[i]} \tau[i].P} \qquad \text{(Act)}\ \frac{P \xrightarrow{\eta[j]} P' \qquad i \neq j}{\mu[i].P \xrightarrow{\eta[j]} \mu[i].P'} \qquad \text{(Sum)}\ \frac{P_i \xrightarrow{\mu[k]} P_i' \qquad \forall j \neq i.(P_j' = P_j \wedge \mathtt{std}(P_j))}{\sum_{i\in I} P_i \xrightarrow{\mu[k]} \sum_{i\in I} P_i'}$$

$$\text{(Par)}\ \frac{P \xrightarrow{\mu[i]} P' \qquad i \notin \mathtt{key}(Q)}{P \parallel Q \xrightarrow{\mu[i]} P' \parallel Q} \qquad \text{(Syn)}\ \frac{P \xrightarrow{\alpha[i]} P' \qquad Q \xrightarrow{\overline{\alpha}[i]} Q'}{P \parallel Q \xrightarrow{\tau[i]} P' \parallel Q'} \qquad \text{(Res)}\ \frac{P \xrightarrow{\mu[i]} P' \qquad \mu \notin \{a, \overline{a}\}}{(\nu a)P \xrightarrow{\mu[i]} (\nu a)P'}$$

$$\text{(Const)}\ \frac{A \triangleq P \qquad P \xrightarrow{\mu[i]} P'}{A \xrightarrow{\mu[i]} P'}$$

Figure 4: $\mathtt{rCCS}^{\mathtt{a}}$ forward semantics, symmetric rules for Par and Syn are omitted.

record both the sender and the receiver of a message. Accordingly, a consumed message is represented as $[i]\langle a\rangle[j]$, indicating that it was emitted by an output prefix marked with i and consumed by a input prefix marked with j. Rule (In) uses a key for decorating the input prefix. Finally, rule (Tau) records the execution of a τ-prefix.

Rule (Act) is the only additional rule, introduced to account for the persistency of executed prefixes. It inductively enables actions to be propagated through prefixes that have already been executed. The second premise requires that the key i decorating the already executed prefix be different from the key j associated with the action that is being performed. This condition is essential to ensure that each reduction step uses a fresh key, thereby preserving the uniqueness of keys. Rule (Sum) captures the behaviour of a non-deterministic choice. If the process is standard (see Definition 4), then the choice has not yet been resolved, and any of its branches may proceed. Once the choice is resolved, however, only the selected (non-standard) branch is allowed to execute. In this way, the choice operator, together with the discarded branches, is preserved in the syntax and acts as a decoration of the forward computation. This information is crucial for enabling the reversal of a choice. The additional premise in rule (Par) ensures that the key in the label, that is, the one assigned to the action being executed, has not been previously used within the parallel component Q. This condition enforces the uniqueness of keys. Rules (Syn), (Res) and (Const) are a straightforward adaptation of the corresponding rules in Figure 2.

Example 6. *Consider the system R and the execution shown in Example 2. The same execution can be mimicked in $\mathtt{rCCS}^{\mathtt{a}}$ as follows*

$$\begin{aligned} R &\xrightarrow{\tau[i]} (a.\mathbf{0}+b.\mathbf{0}) \parallel \overline{a}[i].\overline{b}.\mathbf{0} \parallel [i]\langle a\rangle \\ &\xrightarrow{\tau[j]} (a.\mathbf{0}+b.\mathbf{0}) \parallel \overline{a}[i].(\overline{b}[j].\mathbf{0} \parallel [j]\langle b\rangle) \parallel [i]\langle a\rangle \\ &\xrightarrow{\tau[w]} (\underline{a[w]}.\mathbf{0}+b.\mathbf{0}) \parallel \underline{\overline{a}[i]}.(\overline{b}[j].\mathbf{0} \parallel [j]\langle b\rangle) \parallel \underline{[i]\langle a\rangle[w]} = S \end{aligned}$$

where the parts of the system involved in the communication are underlined. As indicated by the

message $[i]\langle a\rangle[w]$*, the communication takes place in two distinct steps: the emission, decorated by i, and the consumption, decorated by w*

It is worth noting that prefixes are not consumed by reduction, but rather marked with keys. Moreover, emitted messages are placed in parallel with the emitting prefix. This changes the structure of the process and is therefore not compliant with the framework of [35], which requires operators to remain static. Similarly, discarded branches of a choice are not removed; instead, they are retained in the syntax as a form of decoration. For instance, after the consumption of the message $[i]\langle a\rangle$*, the process* $a.\mathbf{0}+b.\mathbf{0}$ *evolves into* $a[w].\mathbf{0}+b.\mathbf{0}$*, where the branch* $+,b.\mathbf{0}$ *serves as a decoration/context of the process* $a[w].\mathbf{0}$*.*

The backward rules are reported in Figure 5. Each forward rule has a corresponding backward rule that restores the configuration preceding the action. In particular, reversing an emission removes the generated message and restores the original output prefix, while reversing a message consumption transforms a consumed message $[i]\langle a\rangle[j]$ back into an available message $[i]\langle a\rangle$. The symmetry between forward and backward rules ensures that the system satisfies the Loop Lemma (Lemma 13), stating that every forward transition can be undone. The premise $\mathtt{std}(P)$ in rules (Out$^\bullet$), (In$^\bullet$), and (Tau$^\bullet$) ensures that a sequential process having several previously executed prefixes reverses the last one first.

$$(\text{Out}^\bullet)\ \frac{\mathtt{std}(P)}{\overline{a}[i].P \parallel [i]\langle a\rangle \xleftarrow{\tau[i]} \overline{a}.P} \qquad (\text{Msg}^\bullet)\ [i]\langle a\rangle[j] \xleftarrow{\overline{a}[j]} [i]\langle a\rangle \qquad (\text{In}^\bullet)\ \frac{\mathtt{std}(P)}{a[i].P \xleftarrow{a[i]} a.P}$$

$$(\text{Tau}^\bullet)\ \frac{\mathtt{std}(P)}{\tau[i].P \xleftarrow{\tau[i]} \tau.P} \qquad (\text{Act}^\bullet)\ \frac{P \xleftarrow{\eta[j]} P' \qquad i \neq j}{\mu[i].P \xleftarrow{\eta[j]} \mu[i].P'}$$

$$(\text{Sum}^\bullet)\ \frac{P_i \xleftarrow{\mu[k]} P_i' \qquad \forall j \neq i.(P_j' = P_j \wedge \mathtt{std}(P_j))}{\sum_{i\in I} P_i \xleftarrow{\mu[k]} \sum_{i\in I} P_i'} \qquad (\text{Par}^\bullet)\ \frac{P \xleftarrow{\mu[i]} P' \qquad i \notin \mathtt{key}(Q)}{P \parallel Q \xleftarrow{\mu[i]} P' \parallel Q}$$

$$(\text{Syn}^\bullet)\ \frac{P \xleftarrow{\alpha[i]} P' \qquad Q \xleftarrow{\overline{\alpha}[i]} Q'}{P \parallel Q \xleftarrow{\tau[i]} P' \parallel Q'} \qquad (\text{Res}^\bullet)\ \frac{P \xleftarrow{\mu[i]} P' \qquad \mu \notin \{a,\overline{a}\}}{(\nu a)P \xleftarrow{\mu[i]} (\nu a)P'} \qquad (\text{Const}^\bullet)\ \frac{A \triangleq P \qquad P' \xleftarrow{\mu[i]} P}{P' \xleftarrow{\mu[i]} A}$$

Figure 5: rCCS$^{\mathtt{a}}$ backward semantics, symmetric rules for Par and Syn are omitted.

Example 7. *Consider now the system S in Example 6, which can retract its decision of consuming message* $\langle a\rangle$ *and consume the message* $\langle b\rangle$ *as follows:*

$$\begin{aligned} S &\xrightarrow{\tau[w]} a.\mathbf{0}+b.\mathbf{0} \parallel \overline{a}[i].(\overline{b}[j].\mathbf{0} \parallel [j]\langle b\rangle) \parallel [i]\langle a\rangle \\ &\xrightarrow{\tau[z]} a.\mathbf{0}+b[z].\mathbf{0} \parallel \overline{a}[i].(\overline{b}[j].\mathbf{0} \parallel [j]\langle b\rangle[z]) \parallel [i]\langle a\rangle \end{aligned}$$

It is worth noting that the message $[i]\langle a\rangle$ *cannot be reversed in the reached processes, since the prefix* $\overline{a}[i]$ *cannot be undone after having caused* $\overline{b}[j]$*. Therefore, in order to revert the emission on a, it is first necessary to revert the emission on b. This enforces a cause-respecting reversal discipline.*

Not all systems that can be written using the grammar in Figure 3 are meaningful. For instance, $a.b[i].\mathbf{0}$ is not. Therefore, we introduce the notion of *reachable* systems, namely those that are derivable from a standard one.

Definition 8 (Reachable System)**.** A system R is said to be *reachable* if it can be derived from a standard system using the rules reported in Figures 4 and 5.

From now on, we only consider reachable systems.

Lemma 9. *Let R and S two a reachable systems. We have:*

- *if* $R \xrightarrow{\mu[i]} S$ *then* $\mathtt{key}(S) = \mathtt{key}(R) \cup \{i\}$
- *if* $R \xleftarrow{\mu[i]} S$ *then* $\mathtt{key}(R) = \mathtt{key}(R) \setminus \{i\}$

Every reachable system induces a partial order on its keys, which reflects the causal execution order of the actions that produced it, as formally stated in the definition below.

Definition 10 (Ordering on keys)**.** The function $\mathtt{po}(\cdot)\colon \mathit{Systems} \to 2^{\mathscr{K}\times\mathscr{K}}$ is inductively defined as follows:

$$\mathtt{po}([i]\langle a\rangle[j]) = \{(i,j)\} \qquad \mathtt{po}(A) = \emptyset \qquad \mathtt{po}([i]\langle a\rangle) = \{(i,i)\}$$
$$\mathtt{po}(R) = \emptyset \text{ if } \mathtt{std}(R) \qquad \mathtt{po}((\nu a)R) = \mathtt{po}(R) \qquad \mathtt{po}(R \parallel S) = \mathtt{po}(R) \cup \mathtt{po}(S)$$
$$\mathtt{po}(\mu[i].P) = \{(i,j) \mid j \in \mathtt{key}(P)\} \cup \mathtt{po}(P) \qquad \mathtt{po}(\textstyle\sum_{i\in I}\pi_i.P_i) = \bigcup_{i\in I}\mathtt{po}(P_i)$$

We let $\leq_R$ denote the reflexive and transitive closure of $\mathtt{po}(R)$.

Note that the pair (i,j), denoted by $i < j$, implies that the action with key i is a *cause* of the action with key j. This accounts for both the *structural causality* inherent in the process (e.g., a prefix causes its continuation) and the *causality arising* from message consumption; specifically, $[i]\langle a\rangle[j]$ indicates that the output action with key i precedes the input action with key j.

The following result establishes that $\leq_R$ is indeed a partial order on the set of keys.

Proposition 11. *Let R be a reachable system. Then $(\mathtt{key}(R), \leq_R)$ is a partial order.*

Proof. Since $\leq_R$ is defined as the reflexive and transitive closure of $\mathtt{po}(R)$ it remains to show that it is antisymmetric. If R is reachable, then there exists a standard process S s.t. $S \mapsto^* R$. The proof follows by induction on the length of the derivation.

Base case: $R = S$ and $\mathtt{po}(S) = \emptyset$. The thesis follows trivially.

Inductive step. Then, $S \mapsto^* R' \mapsto R$ and $(\mathtt{key}(R'), \leq_{R'})$ is a partial order. The proof proceeds by case analysis on the transition $R' \mapsto R$. Without loss of generality, we only consider forward rules, as these are the only ones that introduce new keys. If the last applied rule is (OUT) then R is of the form $a[i].P \parallel [i]\langle a\rangle$ and we have that the new key i is the element of the partial order $(\{i\}, \{(i,i)\})$, as $[i]\langle a\rangle$ does not induce any order, beside the pair (i,i), and P is standard. If the rule is (MSG) then the order is clearly a partial order as $\mathtt{po}([i]\langle a\rangle[j]) = \{(i,j)\}$ and the new key j is set as the maximal element of the partial order. In the case of (IN) and (TAU) the new key is the new minimal element. If the rule is (SUM) it is enough to observe that the key is added in one of the summand and the other are standard, hence they do not contribute to the partial order. If the rule is (PAR) the thesis follows as the keys are disjoint. If the rule is (SYN) we have the partial orders $(\mathtt{key}(P'), \leq_{P'})$ and $(\mathtt{key}(Q'), \leq_{Q'})$, and the new key i is the maximal in $(\mathtt{key}(Q'), \leq_{Q'})$ and the minimal in $(\mathtt{key}(P'), \leq_{P'})$. As $\mathtt{key}(P') \cap \mathtt{key}(Q') = \{i\}$ the thesis follows. If the rule was (RES) or (CONST) the thesis follows easily. Finally assume that the rule is (ACT), then the new key i is added to $(\mathtt{key}(P'), \leq_{P'}$ which has the key j as minimal element. As $\mathtt{po}(\mu[i].P') = \{(i,j) \mid j \in \mathtt{key}(P')\} \cup \mathtt{po}(P'))$ we have the thesis. □

We conclude this section by defining the notion of *choice context*.

Definition 12 (Choice Context)**.** A *choice context* is a process with a hole $\bullet$ generated by the following grammar:

$$\mathtt{C} ::= \mu[i].\mathtt{C} \mid R \parallel \mathtt{C} \mid \nu a(\mathtt{C}) \mid \mu_j[k].\mathtt{C} + \sum_{I\setminus\{j\}} \mu_i.R_i \mid \bullet$$

We write $\mathtt{C}[R]$ to denote the process obtained by replacing the hole $\bullet$ in $\mathtt{C}$ with R.

4 rCCS$^{\mathtt{a}}$ Properties

In this section, we investigate the properties of rCCS$^{\mathtt{a}}$. We begin with a fundamental property for any reversible calculus: every action can be undone. Formally:

Lemma 13 (Loop Lemma)**.** *Let R and S two reachable systems. Then $R \xrightarrow{\mu[i]} S$ iff $S \xhookrightarrow{\mu[i]} R$.*

Proof. The proof proceeds by case analysis on the transition rule applied. The analysis is straightforward, as each forward rule has a unique corresponding backward rule, and vice versa. □

Another fundamental property is *causal-consistent reversibility* [12]. This property essentially states that the calculus stores the correct amount of causal information; indeed, a notion of *causal consistency* is required to assess it. To discuss this property, we borrow several definitions from [12]. We use t, t', s, s' to range over transitions. In a transition $t : R \xrightarrow{\mu[i]} S$ we call R the *source* of the transition, and S the *target* of the transition. Two transitions are said to be *coinitial* if they have the same source, and cofinal if they have the same target. Given a transition t, we indicate with $\underline{t}$ its inverse, i.e., if $t : R \xrightarrow{\mu[i]} S$ (resp. $t : R \xhookrightarrow{\mu[i]} S$) then $\underline{t} : S \xhookrightarrow{\mu[i]} R$ (resp. $\underline{t} : S \xrightarrow{\mu[i]} R$).

We now define a notion of *independence* between coinitial rCCS$^{\mathtt{a}}$ transitions, based on a causality preorder on keys (inspired by [26]). Intuitively, independent transitions can be executed in any order (formally stated in Proposition 18) whereas non-independent transitions represent a *choice*: the execution of one precludes the other.

Definition 14 (Conflict)**.** Given a rCCS$^{\mathtt{a}}$ reachable system R, two coinitial transitions t and s are *conflicting* if one of the following holds:

1. $t : R \xrightarrow{\tau[i]} S_1$, $s : R \xrightarrow{\tau[j]} S_2$ and there exists a message $[z]\langle a\rangle$ in R such that $[z]\langle a\rangle[i]$ belongs to S_1 and $[z]\langle a\rangle[j]$ belongs to S_2;
2. $t : R \xrightarrow{\mu[i]} S_1$, $s : R \xrightarrow{\eta[j]} S_2$, and the two transitions are generated by the same sum operator;
3. $t : R \xrightarrow{\mu[i]} S_1$, $s : R \xhookrightarrow{\eta[j]} S_2$ and $j \leq_{S_1} i$ or vice versa.

Let us comment on the notion of conflicting transitions of Definition 14. The first item tells us that two (forward) consumption of the same message (marked by the key z) are in conflict. The second item just says that all the branches of a choice operator are in conflict with each others. For example, if we take the process $R = a.\mathbf{0} + b.\mathbf{0}$ we have that $R \xrightarrow{a[i]} a[i].\mathbf{0} + b.\mathbf{0}$ and $R \xrightarrow{b[j]} a.\mathbf{0} + b[j].\mathbf{0}$ and these two transitions are in conflict. The last item tells that two transitions are in conflict when a reverse step eliminates some causes of a forward step. For example, the process $a[i].b.\mathbf{0}$, can do a forward step with label $b[j]$, or a backward step with label $a[i]$, and we have that $i \leq j$

Definition 15 (Independence)**.** Given a rCCS$^{\mathtt{a}}$ reachable system R, two coinitial transitions t and s are *independent*, written $t \mathtt{I} s$, if they are not conflicting.

4.1 Causal Consistency

Intuitively, coinitial and cofinal computations share the same causal information. Hence, we require that they admit the same reversals. To formalize this, we first introduce the notions of *path* and *path equivalence*.

We let χ, ω range over sequences of transitions, which we call *paths*. We denote the empty sequence by ε. We denote the number of transitions in a path χ by $|\chi|$. When χ is forward, i.e. composed by forward only transitions, we denote by $\underline{\chi}$ the corresponding backward path, where the order of the transitions is reversed. Moreover, we let $\chi_1\chi_2$ denote the composition of two paths, χ_1 and χ_2, provided they are *composable*; that is, when the target of χ_1 coincides with the source of χ_2.

Definition 16 (Causal Equivalence)**.** Let $\asymp$ be the smallest equivalence on paths closed under composition and satisfying:

swap: $ts' \asymp st'$ for every two coinitial independent transitions $t : R \xrightarrow{\mu[i]} R_1$ and $s : R \xrightarrow{\eta[j]} R_2$ and every two cofinal transitions $s' : R_1 \xrightarrow{\eta[j]} S$, $t' : R_2 \xrightarrow{\mu[i]} S$;

cancellation: $t\underline{t} \asymp \varepsilon$ and $\underline{t}t \asymp \varepsilon$

Intuitively, two paths are *causally equivalent* if they differ only by the reordering of independent transitions or the addition (or removal) of *do–undo* and *undo–redo* transition pairs.

Definition 17 (Causal Consistency (CC))**.** An LTS is causal consistent if for any coinitial and cofinal paths χ and ω we have $\chi \asymp \omega$.

We can now prove causal consistency, using the theory in [27]. Given an LTS in which loop lemma holds, and with a notion of independence on transitions, thanks to theory in [27] one can derive causal consistency from three basic properties:

- SP (Square Property),
- BTI (Backward Transitions are Independent),
- WF (Well Foundedness).

The Square Property tells that two coinitial independent transitions commute, thus closing a diamond. Formally:

Property 18 (Square Property - SP)**.** *Given a* $\mathtt{rCCS}^{\mathtt{a}}$ *reachable system R and two coinitial transitions $t : R \xrightarrow{\mu[i]} R_1$ and $s : R \xrightarrow{\eta[j]} R_2$ with $t \,\mathtt{I}\, s$, then there exist two cofinal transitions $t' : R_1 \xrightarrow{\eta[j]} S$ and $s' : R_2 \xrightarrow{\mu[i]} S$ for some reachable system S.*

Proof. Assume $t : R \xrightarrow{\mu[i]} R_1$ and $s : R \xrightarrow{\eta[j]} R_2$ and $t \,\mathtt{I}\, s$. We proceed by cases on the kind of the applied transitions: either forward or backward.

1. $t : R \xrightarrow{\mu[i]} R_1$ and $s : R \xrightarrow{\eta[j]} R_2$ are both forward. As $t \,\mathtt{I}\, s$ we know that the two transitions do not imply consuming the same message nor they are different branches of the same sum. We proceed by case analysis on the forward rules. We just show the most significative cases.

   If the two transition are an emission of a message, w.l.o.g, we can assume that $R = (\nu\tilde{a})(R_l \parallel R_r)$ with $\tilde{a}$ a set of restricted names, and $R_l \xrightarrow{\mu[i]} R'_l$ and $R_r \xrightarrow{\eta[j]} R'_r$. It is easy to see that by applying twice the rule (PAR) we have:

$$\nu\tilde{a}(R_l \parallel R_r) \xrightarrow{\mu[i]} \nu\tilde{a}(R'_l \parallel R_r) \xrightarrow{\eta[j]} \nu\tilde{a}(R'_l \parallel R'_r) \quad \text{and}$$
$$\nu\tilde{a}(R_l \parallel R_r) \xrightarrow{\eta[j]} \nu\tilde{a}(R_l \parallel R'_r) \xrightarrow{\mu[i]} \nu\tilde{a}(R'_l \parallel R'_r)$$

If the two reductions consume different messages we have have to distinguish two cases: either the two communications happen in two separate parts of the system, or not. In the first case we can proceed as in the previous case by assuming $R = \nu\tilde{a}(R_l \parallel R_r)$ with the two parallel systems not interacting each other. In the second case instead, we have that the two systems interact, that is the message consumed by R_r is present in R_l and vice versa. In this case we have that $R = \nu\tilde{a}(\mathtt{C}_1[R_l] \parallel \mathtt{C}_2[R_r])$ with $\mathtt{C}_1[\bullet]$ and $\mathtt{C}_2[\bullet]$ being two choice contexts (see Definition 12). Also in this case we have that both R_l and R_r can do two different and independent forward transitions. We can then conclude by making the cases analysis on smaller terms.

2. $t : R \xmapsto{\mu[i]} R_1$ is forward and $s : R \xmapsto{\eta[j]} R_2$ is backward. By case analysis on the applied rules. The reasoning is similar to the forward case.
3. $t : R \xmapsto{\mu[i]} R_1$ is backward and $s : R \xmapsto{\eta[j]} R_2$ is forward. By case analysis on the applied rules. The reasoning is similar to the previous case.
4. $t : R \xmapsto{\mu[i]} R_1$ and $s : R \xmapsto{\eta[j]} R_2$ are both backward. By case analysis on the applied rules. The reasoning is similar to the previous case.

□

BTI generalises the concept of backward determinism used for reversible sequential languages. It specifies that two backward transitions from a same configuration are always independent.

Property 19. *[Backward Transitions are Independent - BTI] Given a reachable* $\mathtt{rCCS}^{\mathtt{a}}$ *system R, any two distinct coinitial backward transitions* $t : R \xhookrightarrow{\mu[i]} S_1$ *and* $t : R \xhookrightarrow{\eta[j]} S_2$ *are independent.*

The *BTI property* holds trivially: according to the definitions of conflicting and independent transitions (Definitions 14 and 15), there are no cases in which two backward transitions are in conflict. Consequently, any pair of backward transitions is always independent.

We now show that reachable configurations have a finite past.

Proposition 20 (Well-Foundedness - WF)**.** *Let* R_k *be a reachable* $\mathtt{rCCS}^{\mathtt{a}}$ *system. There is no infinite sequence starting from* R_k *such that* $R_i \xhookrightarrow{\mu_i[j_i]} R_{i-1}$ *for all* $i \geq k$.

Proof. Well-foundedness (WF) follows from the fact that each backward transition removes a key. Thanks to Lemma 9 we have that number of keys $|\mathtt{key}(R)|$ in a reachable system R is finite, only a finite number of backward steps can be performed. □

The *Parabolic Lemma* [12, Lemma 11] states that any path is causally equivalent to a path consisting of a sequence of backward steps followed by a sequence of forward steps. In other words, up to causal equivalence, paths can be rearranged to first reach the maximum freedom of choice by moving backwards, before proceeding forward. Here we state the following.

Property 21 (Parabolic Property - PP)**.** *An LTS satisfies the Parabolic Property iff for any path* χ*, there exist two forward-only paths* ω, ω' *such that* $\chi \asymp \underline{\omega}\omega'$ *and* $|\omega| + |\omega'| \leq |\chi|$.

Observe that if the LTS of reversing calculi has the Parabolic Property then the parabolic lemma holds.

We can now prove that the parabolic lemma holds, thanks to the proof schema of [27] recalled in the following propositions.

Proposition 22 (cf. Proposition 3.4 [27])**.** *Suppose BTI and SP hold, then PP holds.*

Proposition 23 (cf. Proposition 3.8 [27])**.** *Suppose WF and PP hold, then CC holds.*

Summarizing our achievements we have:

Theorem 24. *The LTS of Definition 5 satisfies PP.*

Proof. This is a consequence of properties 19 and 18. □

Theorem 25. *The LTS of Definition 5 satisfies CC.*

Proof. This is a consequence of theorem 24 and property 20. □

Another important result says that reachable states are reachable via forward-only paths (cf. [27]).

Theorem 26. *An system S is reachable iff there exists a standard system R and a forward only path $R \to^* S$.*

Proof. From PP, by noticing that the backward path is empty since R cannot take backward actions. □

Example 27. *Let us consider the system $R = (a.\mathbf{0} + b.\mathbf{0}) \parallel \overline{a}.\overline{b}.\mathbf{0}$, of Example 2 and the following two paths:*

$$\chi = R \xrightarrow{\tau[i]} \xrightarrow{\tau[j]} \xrightarrow{\tau[z]} (a.\mathbf{0} + b[z].\mathbf{0}) \parallel \overline{a}[i].(\overline{b}[j].\mathbf{0} \parallel [j]\langle b\rangle[z]) \parallel [i]\langle a\rangle$$

$$\omega = R \xrightarrow{\tau[i]} \xrightarrow{\tau[j]} \xrightarrow{\tau[w]} (a[w].\mathbf{0} + b.\mathbf{0}) \parallel \overline{a}[i].(\overline{b}[j].\mathbf{0} \parallel [j]\langle b\rangle) \parallel [i]\langle a\rangle[w]$$
$$\xrightarrow{\tau[w]} \xrightarrow{\tau[z]} (a.\mathbf{0} + b[z].\mathbf{0}) \parallel \overline{a}[i].(\overline{b}[j].\mathbf{0} \parallel [j]\langle b\rangle[z]) \parallel [i]\langle a\rangle$$

Since χ and ω are coinitial, they start in R, and cofinal, they end up in the same system, thanks to Theorem 25 we have that they are causally equivalent, that is $\chi \asymp \omega$.

5 rCCS$^{\mathsf{a}}$ at work: a leader election example

We illustrate the behaviour of rCCS$^{\mathsf{a}}$ with a simple race protocol modelling a leader election scenario, by adapting the example of [31]. Several processes compete to react to a message announcing an election. The process that reacts first proposes itself as the leader (by emitting $n-1$ messages $\overline{p}_i$), while the others processes may choose to follow its lead (by replying $\overline{ok}_i$) or to refuse it (by replying $\overline{no}_i$). This naturally may lead to a deadlock, but thanks to the reversible semantics the system can get back from a global deadlock and try other solutions. Also, when a process proposes itself as a leader (via the prefix $\overline{p}_i$), it must collect $n-1$ confirmative replies before considering itself the new leader. Let $\mathtt{seq}(a,n) = a^1.a^2.\ldots.a^n.\mathbf{0}$ denote the sequential composition of n instances of the prefix a.

$$R_i \triangleq \left(l.(\textstyle\prod_{j\in I\setminus\{i\}} \overline{p}_i.r_i.\overline{a}_i) \parallel \mathtt{seq}(a_i, |I|-1)\right) + \textstyle\sum_{I\setminus\{i\}} p_j.(\tau.\overline{r}_i + \tau.\overline{no}_i) \qquad (2)$$

As an example, let us suppose a system made of 3 processes, that is:

$$\mathtt{S} = R_1 \parallel R_2 \parallel R_3 \parallel \overline{l}.\mathbf{0}$$

where $\overline{l}$ acts as the token emitte to initiate an election. Each R_i can become a leader by consuming the message $\langle l\rangle$ (left branch of the main choice of (2)) or react to someone else's proposal (right-branch of

(2))). We now describe an execution in which R_1 proposes itself as leader, R_2 agrees to it and R_3 refuses. This leads the entire system to a bad state $\mathtt{S}_e$ where: R_1 has not enough confirmation to become leader, no other process can take over and the entire system is deadlocked. In what follows we will use $[\cdot]$ to abstract away keys that are not important for the example, and we will use natural numbers as keys, just to ease the reading of the example.

$$\begin{aligned}\mathtt{S}_e = {} & l[2].\big((\overline{p}_1[3].r_2.\overline{a_1} \parallel p_1[4].r_2.\overline{a_1} \parallel [2]\langle p_1\rangle[5] \parallel [3]\langle p_1\rangle[6]) \parallel \mathtt{seq}(a_1,2)\big) + \textstyle\sum_{I\setminus\{1\}} p_j.(\tau.\bar{r}_i + \tau.\overline{no}_i) \\ & \parallel \big(l.(\textstyle\prod_{j\in I\setminus\{2\}} \overline{p}_i.r_i.\bar{a}_i) \parallel \mathtt{seq}(a_2,2)\big) + p_1[3].(\tau[\cdot].\bar{r}_2 + \tau.\overline{no}_2) + p_3.(\tau.\bar{r}_2 + \tau.\overline{no}_2) \\ & \parallel \big(l.(\textstyle\prod_{j\in I\setminus\{3\}} \overline{p}_i.r_i.\bar{a}_i) \parallel \mathtt{seq}(a_3,2)\big) + p_1[4].(\tau.\bar{r}_3 + \tau.[\cdot].\overline{no}_3[7] \parallel [7]\langle no_3\rangle) + p_2.(\tau.\bar{r}_3 + \tau.\overline{no}_3) \\ & \parallel \bar{l}[1].\mathbf{0} \parallel [1]\langle l\rangle[2]\end{aligned}$$

Thanks to the reversible semantics, process R_3 can revert its decision and agree with R_1 and let the system reach a valid state as follows:

$$\mathtt{S}_e \xrightarrow{\tau[7]} \xrightarrow{\tau[\cdot]} \xrightarrow{\tau[\cdot]} \mathtt{S}_{ok}$$

$$\begin{aligned}\mathtt{S}_{ok} = {} & l[2].\big((\overline{p}_1[3].r_2.\overline{a_1} \parallel p_1[4].r_2.\overline{a_1} \parallel [2]\langle p_1\rangle[5] \parallel [3]\langle p_1\rangle[6]) \parallel \mathtt{seq}(a_1,2)\big) + \textstyle\sum_{I\setminus\{1\}} p_j.(\tau.\bar{r}_i + \tau.\overline{no}_i) \\ & \parallel \big(l.(\textstyle\prod_{j\in I\setminus\{2\}} \overline{p}_i.r_i.\bar{a}_i) \parallel \mathtt{seq}(a_2,2)\big) + p_1[3].(\tau[\cdot].\bar{r}_2 + \tau.\overline{no}_2) + p_3.(\tau.\bar{r}_2 + \tau.\overline{no}_2) \\ & \parallel \big(l.(\textstyle\prod_{j\in I\setminus\{3\}} \overline{p}_i.r_i.\bar{a}_i) \parallel \mathtt{seq}(a_3,2)\big) + p_1[4].(\tau[\cdot].\bar{r}_3 + \tau.\overline{no}_3) + p_2.(\tau.\bar{r}_3 + \tau.\overline{no}_3) \\ & \parallel \bar{l}[1].\mathbf{0} \parallel [1]\langle l\rangle[2]\end{aligned}$$

from which the two messages $\langle r_2\rangle$ and $\langle r_3\rangle$ can be emitted respectively by R_2 and R_3, and consumed by R_1.

6 Conclusions

In this paper, we investigate the integration of reversibility with asynchrony in CCS. As a design choice, we model asynchrony at the semantic level, making CCSa more closely aligned with real-world programming languages. This approach is consistent with recent work on session types, where messages in transit–i.e., messages that have been sent but not yet received–are explicitly considered in the setting of asynchronous session types [10, 36].

Most existing reversible calculi either assume synchronous communication [12, 35, 11, 14], or model asynchronous communication in a restricted way, for instance by disallowing continuations after output prefixes [23, 9]. In this paper, we take a first step toward defining a reversible semantics for asynchronous variants of CCS in which message emission and reception are decoupled events. This separation introduces additional causal dependencies that must be carefully accounted for in the reversible setting. Existing approaches to derive reversible semantics for CCS, such as [12, 35], are not directly applicable and fail to yield a reversible semantics for CCS$^{\mathtt{a}}$. The static approach of [35] fails since the emitting rule does not fit with the considered SOS format. This is due to the fact the arguments of the rule changes and in the conclusion we have a additional argument, that is the emitted message. The dynamic approach of [12] fails since the memory mechanism used to log events is devised for syncrhonisations, and cannot deal with partial events as a message emission. Hence we have devised an ad-hoc reversible semantics by leveraging the key ideas of [35]: making static all the operators of the calculus and using identifiers to mark events. This leads to the use of half marked messages of the form $[i]\langle a\rangle$, indicating a message in

transit, and fully marked messages, of the form $[i]\langle a\rangle[j]$, indicating a message which has been consumed. We remark that in the framework of [35] an elements tagged with two keys, like consumed messages, $[i]\langle a\rangle[j]$ does not exists and contrast the assumptions of the approach. We have then showed that the obtained reversible semantics is cause-respecting, that is causally consistent.

Several directions for future work remain open. One possible extension is to study behavioural equivalences for $\mathtt{rCCS}^{\mathtt{a}}$, for instance by adapting forward-reverse bisimulation notion [35] to the asynchronous setting in the flavour of [1]. Another direction is to investigate a truly concurrent semantics for $\mathrm{CCS}^{\mathtt{a}}$ and $\mathtt{rCCS}^{\mathtt{a}}$, along the lines of recent work on RCCS [30]. Also, another promising direction is to investigate the impact of reversibility in variants of CCS equipped with Linda-like coordination primitives [7, 8], inspired by the Linda model of shared tuple spaces. On the practical side, it would be interesting to explore connections with programming languages that rely on asynchronous message passing, such as Erlang or Go, in order to better understand how reversible semantics could support debugging or rollback mechanisms in distributed systems. This would require to add channel queues, in the case of Go, or mailboxes, in the case of Erlang, to $\mathrm{CCS}^{\mathtt{a}}$.

References

[1] Roberto M. Amadio, Ilaria Castellani & Davide Sangiorgi (1998): *On Bisimulations for the Asynchronous pi-Calculus*. *Theor. Comput. Sci.* 195(2), pp. 291–324, doi:10.1016/S0304-3975(97)00223-5. Available at `https://doi.org/10.1016/S0304-3975(97)00223-5`.

[2] Bogdan Aman, Gabriel Ciobanu, Robert Glück, Robin Kaarsgaard, Jarkko Kari, Martin Kutrib, Ivan Lanese, Claudio Antares Mezzina, Lukasz Mikulski, Rajagopal Nagarajan, Iain Phillips, G. Michele Pinna, Luca Prigioniero, Irek Ulidowski & Germán Vidal (2020): *Foundations of Reversible Computation*. In Irek Ulidowski, Ivan Lanese, Ulrik Pagh Schultz & Carla Ferreira, editors: *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, *Lecture Notes in Computer Science* 12070, Springer, pp. 1–40, doi:10.1007/978-3-030-47361-7_1. Available at `https://doi.org/10.1007/978-3-030-47361-7_1`.

[3] The Go Authors: *The Go programming language*. `https://go.dev`.

[4] Paolo Baldan, Filippo Bonchi, Fabio Gadducci & Giacoma Valentina Monreale (2015): *Modular encoding of synchronous and asynchronous interactions using open Petri nets*. *Sci. Comput. Program.* 109, pp. 96–124, doi:10.1016/J.SCICO.2014.11.019. Available at `https://doi.org/10.1016/j.scico.2014.11.019`.

[5] Michele Boreale, Rocco De Nicola & Rosario Pugliese (1998): *Asynchronous Observations of Processes*. In Maurice Nivat, editor: *Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98*, *Lecture Notes in Computer Science* 1378, Springer, pp. 95–109, doi:10.1007/BFB0053544. Available at `https://doi.org/10.1007/BFb0053544`.

[6] Michele Boreale, Rocco De Nicola & Rosario Pugliese (2002): *Trace and Testing Equivalence on Asynchronous Processes*. *Inf. Comput.* 172(2), pp. 139–164, doi:10.1006/INCO.2001.3080. Available at `https://doi.org/10.1006/inco.2001.3080`.

[7] Nadia Busi, Roberto Gorrieri & Gianluigi Zavattaro (1998): *A Process Algebraic View of Linda Coordination Primitives*. *Theor. Comput. Sci.* 192(2), pp. 167–199, doi:10.1016/S0304-3975(97)00149-7. Available at `https://doi.org/10.1016/S0304-3975(97)00149-7`.

[8] Nadia Busi & Gianluigi Zavattaro (2008): *A process algebraic view of shared dataspace coordination*. *J. Log. Algebraic Methods Program.* 75(1), pp. 52–85, doi:10.1016/J.JLAP.2007.06.003. Available at `https://doi.org/10.1016/j.jlap.2007.06.003`.

[9] Luca Cardelli & Cosimo Laneve (2011): *Reversible structures*. In François Fages, editor: *Computational Methods in Systems Biology, 9th International Conference, CMSB 2011*, ACM, pp. 131–140, doi:10.1145/2037509.2037529. Available at `https://doi.org/10.1145/2037509.2037529`.

[10] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas & Nobuko Yoshida (2017): *On the Preciseness of Subtyping in Session Types*. *Log. Methods Comput. Sci.* 13(2), doi:10.23638/LMCS-13(2:12)2017. Available at `https://doi.org/10.23638/LMCS-13(2:12)2017`.

[11] Ioana Cristescu, Jean Krivine & Daniele Varacca (2013): *A Compositional Semantics for the Reversible p-Calculus*. In: *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*, IEEE Computer Society, pp. 388–397, doi:10.1109/LICS.2013.45. Available at `https://doi.org/10.1109/LICS.2013.45`.

[12] Vincent Danos & Jean Krivine (2004): *Reversible Communicating Systems*. In Philippa Gardner & Nobuko Yoshida, editors: *CONCUR 2004 - Concurrency Theory, 15th International Conference*, *Lecture Notes in Computer Science* 3170, Springer, pp. 292–307, doi:10.1007/978-3-540-28644-8_19. Available at `https://doi.org/10.1007/978-3-540-28644-8_19`.

[13] Michael P. Frank & Karpur Shukla (2021): *Quantum Foundations of Classical Reversible Computing*. *Entropy* 23(6), p. 701, doi:10.3390/E23060701. Available at `https://doi.org/10.3390/e23060701`.

[14] Carlos Galindo, Naoki Nishida, Josep Silva & Salvador Tamarit (2021): *Reversible CSP Computations*. *IEEE Trans. Parallel Distributed Syst.* 32(6), pp. 1425–1436, doi:10.1109/TPDS.2021.3051747. Available at `https://doi.org/10.1109/TPDS.2021.3051747`.

[15] Elena Giachino, Ivan Lanese & Claudio Antares Mezzina (2014): *Causal-Consistent Reversible Debugging*. In Stefania Gnesi & Arend Rensink, editors: *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014*, *Lecture Notes in Computer Science* 8411, Springer, pp. 370–384, doi:10.1007/978-3-642-54804-8_26. Available at `https://doi.org/10.1007/978-3-642-54804-8_26`.

[16] James Hoey & Irek Ulidowski (2019): *Reversible Imperative Parallel Programs and Debugging*. In Michael Kirkedal Thomsen & Mathias Soeken, editors: *Reversible Computation - 11th International Conference, RC 2019*, *Lecture Notes in Computer Science* 11497, Springer, pp. 108–127, doi:10.1007/978-3-030-21500-2_7. Available at `https://doi.org/10.1007/978-3-030-21500-2_7`.

[17] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In George C. Necula & Philip Wadler, editors: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, ACM, pp. 273–284, doi:10.1145/1328438.1328472. Available at `https://doi.org/10.1145/1328438.1328472`.

[18] Stefan Kuhn & Irek Ulidowski (2022): *Modelling of DNA mismatch repair with a reversible process calculus*. *Theor. Comput. Sci.* 925, pp. 68–86, doi:10.1016/J.TCS.2022.06.009. Available at `https://doi.org/10.1016/j.tcs.2022.06.009`.

[19] Ericsson Computer Science Laboratory & OTP (Open Telecom Platform): *The Erlang programming language*. `https://www.erlang.org`.

[20] Rolf Landauer (1961): *Irreversibility and Heat Generation in the Computing Process*. *IBM J. Res. Dev.* 5(3), pp. 183–191, doi:10.1147/RD.53.0183. Available at `https://doi.org/10.1147/rd.53.0183`.

[21] Ivan Lanese, Michael Lienhardt, Claudio Antares Mezzina, Alan Schmitt & Jean-Bernard Stefani (2013): *Concurrent Flexible Reversibility*. In Matthias Felleisen & Philippa Gardner, editors: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013*, *Lecture Notes in Computer Science* 7792, Springer, pp. 370–390, doi:10.1007/978-3-642-37036-6_21. Available at `https://doi.org/10.1007/978-3-642-37036-6_21`.

[22] Ivan Lanese, Doriana Medic & Claudio Antares Mezzina (2021): *Static versus dynamic reversibility in CCS*. *Acta Informatica* 58(1-2), pp. 1–34, doi:10.1007/S00236-019-00346-6. Available at `https://doi.org/10.1007/s00236-019-00346-6`.

[23] Ivan Lanese, Claudio Antares Mezzina & Jean-Bernard Stefani (2010): *Reversing Higher-Order Pi*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 - Concurrency Theory, 21th International Conference*, *Lecture Notes in Computer Science* 6269, Springer, pp. 478–493, doi:10.1007/978-3-642-15375-4_33. Available at `https://doi.org/10.1007/978-3-642-15375-4_33`.

[24] Ivan Lanese, Claudio Antares Mezzina & Jean-Bernard Stefani (2016): *Reversibility in the higher-order π-calculus*. *Theor. Comput. Sci.* 625, pp. 25–84, doi:10.1016/J.TCS.2016.02.019. Available at `https://doi.org/10.1016/j.tcs.2016.02.019`.

[25] Ivan Lanese, Claudio Antares Mezzina & Martin Vassor (2026): *Bounded Reversibility in HO$$\pi $$*. In Claudio Antares Mezzina & Alan Schmitt, editors: *Components Operationally: Reversibility and System Engineering: Essays Dedicated to Jean-Bernard Stefani on the Occasion of His 65th Birthday*, *Lecture Notes in Computer Science* 16065, Springer, pp. 24–45, doi:10.1007/978-3-031-99717-4_2. Available at `https://doi.org/10.1007/978-3-031-99717-4_2`.

[26] Ivan Lanese & Iain Phillips (2021): *Forward-Reverse Observational Equivalences in CCSK*. In Shigeru Yamashita & Tetsuo Yokoyama, editors: *Reversible Computation - 13th International Conference, RC 2021*, *Lecture Notes in Computer Science* 12805, Springer, pp. 126–143, doi:10.1007/978-3-030-79837-6_8. Available at `https://doi.org/10.1007/978-3-030-79837-6_8`.

[27] Ivan Lanese, Iain Phillips & Irek Ulidowski (2024): *An Axiomatic Theory for Reversible Computation*. *ACM Trans. Comput. Log.* 25(2), pp. 11:1–11:40, doi:10.1145/3648474. Available at `https://doi.org/10.1145/3648474`.

[28] Ivan Lanese, Ulrik Pagh Schultz & Irek Ulidowski (2022): *Reversible Computing in Debugging of Erlang Programs*. *IT Prof.* 24(1), pp. 74–80, doi:10.1109/MITP.2021.3117920. Available at `https://doi.org/10.1109/MITP.2021.3117920`.

[29] Hernán C. Melgratti, Claudio Antares Mezzina & G. Michele Pinna (2022): *A Petri net view of covalent bonds*. *Theor. Comput. Sci.* 908, pp. 89–119, doi:10.1016/J.TCS.2022.01.013. Available at `https://doi.org/10.1016/j.tcs.2022.01.013`.

[30] Hernán C. Melgratti, Claudio Antares Mezzina & G. Michele Pinna (2024): *A Truly Concurrent Semantics for Reversible CCS*. *Log. Methods Comput. Sci.* 20(4), doi:10.46298/LMCS-20(4:20)2024. Available at `https://doi.org/10.46298/lmcs-20(4:20)2024`.

[31] Claudio Antares Mezzina (2018): *On Reversibility and Broadcast*. In Jarkko Kari & Irek Ulidowski, editors: *Reversible Computation - 10th International Conference, RC 2018*, *Lecture Notes in Computer Science* 11106, Springer, pp. 67–83, doi:10.1007/978-3-319-99498-7_5. Available at `https://doi.org/10.1007/978-3-319-99498-7_5`.

[32] Claudio Antares Mezzina, Rudolf Schlatte, Robert Glück, Tue Haulund, James Hoey, Martin Holm Cservenka, Ivan Lanese, Torben Æ. Mogensen, Harun Siljak, Ulrik Pagh Schultz & Irek Ulidowski (2020): *Software and Reversible Systems: A Survey of Recent Activities*. In Irek Ulidowski, Ivan Lanese, Ulrik Pagh Schultz & Carla Ferreira, editors: *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, *Lecture Notes in Computer Science* 12070, Springer, pp. 41–59, doi:10.1007/978-3-030-47361-7_2. Available at `https://doi.org/10.1007/978-3-030-47361-7_2`.

[33] Claudio Antares Mezzina, Francesco Tiezzi & Nobuko Yoshida (2025): *Checkpoint-based rollback recovery in session programming*. *Log. Methods Comput. Sci.* 21(1), p. 2, doi:10.46298/LMCS-21(1:2)2025. Available at `https://doi.org/10.46298/lmcs-21(1:2)2025`.

[34] Robin Milner (1980): *A Calculus of Communicating Systems*. *Lecture Notes in Computer Science* 92, Springer, doi:10.1007/3-540-10235-3. Available at `https://doi.org/10.1007/3-540-10235-3`.

[35] Iain Phillips & Irek Ulidowski (2007): *Reversing algebraic process calculi*. *J. Log. Algebraic Methods Program.* 73(1-2), pp. 70–96, doi:10.1016/J.JLAP.2006.11.002. Available at `https://doi.org/10.1016/j.jlap.2006.11.002`.

[36] Kai Pischke, Jake Masters & Nobuko Yoshida (2026): *Asynchronous Global Protocols, Precisely: Full Proofs*. arXiv:2505.17676.

[37] Martin Vassor & Jean-Bernard Stefani (2018): *Checkpoint/Rollback vs Causally-Consistent Reversibility*. In Jarkko Kari & Irek Ulidowski, editors: *Reversible Computation*, *Lecture Notes in Computer Science* 11106, Springer, pp. 286–303, doi:10.1007/978-3-319-99498-7_20. Available at `https://doi.org/10.1007/978-3-319-99498-7_20`.