The Concurrency Column

by

Nobuko Yoshida

Department of Computing
Imperial College London
180 Queen's Gate, London, SW7 2AZ
n.yoshida@imperial.ac.uk, http://mrg.doc.ic.ac.uk/

Relative expressiveness of calculi for reversible concurrency

Doriana Medić
Focus Team, Inria, Sophia Antipolis - Méditerranée
doriana.medic@gmail.com

Abstract

A number of formalisms have been proposed to model various approaches to reversibility and to better understand its properties and characteristics. However, the relation between these formalisms has hardly been studied. This paper examines the expressiveness of the causal-consistent reversibility in process algebras CCS and π-calculus. In particular, we show, by means of encodings, that LTSs of two reversible extensions of CCS, Reversible CCS [1] and CCS with Keys [2], are isomorphic up to some structural transformations of processes. To study different causal semantics for π-calculus, we devise a uniform framework for reversible π-calculi that is parametric with respect to a data structure that stores information about the extrusion of a name. Depending on the used data structure, different causal semantics can be obtained. We show that reversibility induced by our framework when instantiated with three different data structures is causally-consistent and prove a causal correspondence between certain causal semantics and matching instance of the framework.

1 Introduction

Context and Motivation The interest in reversible computing is motivated by Landauer's observation [3] that only irreversible computation generates heat. With the aim to reduce heat dissipation in computing machinery and thus achieve higher density and speed, reversibility was initially studied in circuits. Nowadays, reversible computing is examined in many different contexts, from biological and chemical modelling [4, 5, 6, 7], program debugging and testing [8, 9, 10], quantum computing [11], up to reliable systems where reversibility can be used to express different transactional models [12, 13].

An important feature of a reversible system is the ability to detect the *last action* of a computation. While in a sequential setting, there is only one order of

the computation and the last action is easily detected, in the concurrent setting, the notion of the last action is not clearly defined. For example, if we take a system $a.b \mid c$ which performs actions a, c and b, the question is, what is the correct reversible computation? It could be the execution of the backward actions in the opposite order of the forward steps (reversing b, c and then a) or maybe reversing the action c as the first one followed by b and a. The definition of the last action in a concurrent system is given by the notion of causally-consistent reversibility for reversible CCS [1]: an action can be reversed (considered as the last one), only if all its consequences have been reversed. Aside from causal-consistent, there are other works considering different notions of reversibility: out-of-causal order [6, 14, 15] and backtracking [14, 16]. In this paper, we are concentrated on the causal-consistent reversibility in concurrent systems described via process algebras [17], in particular: Calculus of Communicating Systems (CCS) [18] and π-calculus [19].

In CCS, despite the fact that there is only one notion of causality, so-called *structural causality*, induced by the prefixing operator '.' and by synchronisations, there exist two different approaches to reversibility, namely Reversible CCS (RCCS) [1] and CCS with Communication Keys (CCSK) [2]. The calculus given in [1] is the first reversible extension for CCS, where the main idea is to attach memories organised as stacks to every CCS process. Then the reversible process is of the form $m \triangleright P$, where m is a memory containing the information about the past actions that process did and P is a CCS process. The second reversible calculus is obtained by applying a general method for reversing process calculi [2], given in a SOS [20] format, on CCS. The main ideas are to identify every action with the unique key and to make each operator of the calculus static. In this way after the execution, actions are not discarded but annotated with the key and preserved in the structure of the process. Hence, there is no need for external memories. For more than 10 years, these two approaches have evolved independently giving rise to many extensions and results (here are just some of them [5, 21, 22, 7, 23, 24]). The question is: are these two approaches to reversibility equivalent?

In π-calculus, differently from CCS, the possibility of creating new channel names and treating channels as sent values generates two kinds of dependencies between the actions: *subject* (structural) and the *object* dependency. Subject dependency, as in CCS, is determined by the nesting of the prefixes and by synchronisations. For instance, in the process $\overline{b}a.c(x)$, action $c(x)$ structurally depends on the action $\overline{b}a$. On the other hand, object dependency is generated by extruding (or opening) a name. For instance, in the process $\nu a(\overline{b}a \mid a(x))$, the action $\overline{b}a$ extrudes bound name a and enables the execution of action $a(x)$. Hence, we can say that action $a(x)$ is object dependent on the action $\overline{b}a$. The object dependency brings up the particular problem of a parallel extrusion of the same name. For example, if the considered process is $\nu a(\overline{b}a \mid \overline{c}a \mid a(x))$ and the output actions $\overline{b}a$ and $\overline{c}a$ are

executed both before the action $a(x)$, the discussion is about which of the output actions is the cause of the action $a(x)$. In the following, we consider three different approaches to a given problem.

The first approach, we represent by the compositional causal semantics for the forward π-calculus given in [25]. In this case, the order of the actions is important and only the very first action that extrudes the bound name is the cause of every action using it. By abstracting away from the technique to keep track about the causality, as stated in [26], this approach coincides also with [27, 26].

For the second approach we consider a compositional event structure semantics for the forward π-calculus, introduced in [28]. Following it, the output actions that extrude a bound name are executed concurrently and one of them is the cause of the action $a(x)$ but it is not necessary to know which one exactly (it is enough to be sure that at least one of the extruders happened before the action $a(x)$). As argued in [29], since we consider causal-consistent reversibility, where it is obligatory to keep track of the causes, we will revise the semantics of [28] to obtain this feature.

The final approach is represented by a compositional semantics for the reversible π-calculus [22]. In this case, the action $a(x)$ can choose as its cause one of the outputs executed concurrently and this will be recorded into the memory of the process. Which one is chosen is determined by the context. This causal semantics satisfies several correctness criteria for causal models [30], what is not the case for the other causal semantics. When taking into account just the order of the actions, under this approach we can consider also the semantics of [31].

Contributions and Structure of the Paper In this work, we first answer to the question about the relation between two reversible extensions of CCS. In particular, in Section 2 we briefly recall RCCS and CCSK, while in Sections 3.1 and 3.2 we provide the encodings of CCSK into RCCS and of RCCS into CCSK and prove their correctness in terms of the operational correspondence. The main result, the isomorphism between labelled transition systems of CCSK and RCCS (up to a few structural transformations on processes) is shown in Section 3.3. Additionally, in the same section, we discussed the uniform encoding and argue that no uniform encoding can exist since reachable RCCS processes are not closed under the parallel composition.

Moving to π-calculus, to study different approaches to causality in the context of reversible computation, we devise a framework for reversible π-calculi, parametric with respect to the data structure that stores information about the extrusions of a name. The reversibility is obtained by extending the approach of [2] to work with π-calculus. The three approaches to causality in π-calculus (mentioned above), can be obtained by using different data structures. Additionally, the framework adds reversibility to the semantics defined only for the forward computations and in that way permits different orders of the causally-consistent backward steps. In particular,

in Section 4, we show the idea behind the three causal semantics [25, 28, 22] that we consider. The parametric framework for reversible π-calculi we present in Section 5, by giving its syntax (Section 5.1) and operational semantics (Section 5.2). We show how different causal semantics can be mapped into the framework in Section 6 and prove the causal correspondence between the semantics of [25] and matching instance of the framework. In Section 7 we show that reversibility in the framework, when considered data structures are used, is causally-consistent. Section 8 concludes the paper.

Origin of the Results The work presented here is based on the author's PhD thesis "Relative expressiveness of calculi for reversible concurrency" [29]. The result of the isomorphism between two reversible CCS, namely RCCS [1] and CCSK [2], is done in collaboration with Claudio A. Mezzina and Ivan Lanese and it is the continuation of the work presented in [32]. The parametric framework for reversible π-calculi is joint work with Claudio A. Mezzina, Nobuko Yoshida and Iain Phillips, published in [33].

2 Reversibility in CCS

In this Section, we briefly recall the syntax of CCS [18], and its two reversible extensions, namely RCCS [1, 21] and CCSK [2].

There is a slight difference in CCS used as a base for these two reversible calculi. RCCS is built on the top of CCS defined with a guarded choice and it features silent actions, while CCSK considers CCS in which unguarded choice is allowed but it features no silent actions. Both calculi do not take into account the recursion operator, still, it can be easily added following the guidance in [21, 2]. In this work we are focused on the reversibility mechanism, hence to have a finer correspondence, we considered as the base of both reversible calculi, CCS used in [1]. In what follows, we give the syntax of CCS.

Let a range over the set of actions, written $\mathcal{A}$, and $\overline{\mathcal{A}} = \{\overline{a} \mid a \in \mathcal{A}\}$ be the corresponding set of co-actions. We let α, β with their decorated versions to range over the set $\mathtt{Act}_\tau = \mathtt{Act} \cup \{\tau\}$ ($\mathtt{Act} = \mathcal{A} \cup \overline{\mathcal{A}}$), where $\tau \notin \mathtt{Act}$ is the *silent* action.

The syntax of CCS is defined as:

$$P, Q ::= \mathbf{0} \mid \sum_{i \in I} \alpha_i.P_i \mid (P \mid Q) \mid P\backslash a \qquad\qquad \alpha ::= a \mid \overline{a} \mid \tau$$

The idle process is represented with $\mathbf{0}$, while a prefix (or action) can be an input a, an output $\overline{a}$ or the silent action τ. Term $\sum_{i \in I} \alpha_i.P_i$ defines the choice operator, where P_i stands for the continuation to be executed after the action α_i. We write $\alpha.P$ for unary choice and $\alpha_j.P_j + Q$ where $Q = \sum_{i \in I \setminus \{j\}} \alpha_i.P_i$ to highlight a specific alternative. Parallel composition of processes P and Q is represented with $P \mid Q$, while $P\backslash a$ indicates that name a is restricted to the process P (scope of the name

$$\text{(RCCS Processes)} \quad R, S ::= m \triangleright P \mid (R \mid S) \mid R\backslash a$$
$$\text{(Memories)} \quad m ::= \langle\rangle \mid \langle k, \alpha, Q\rangle \cdot m \mid \langle\uparrow\rangle \cdot m$$

Figure 1: RCCS syntax

a is the process P). The restriction operator is the only binder in CCS. We write $\mathtt{fn}(P)$ and $\mathtt{bn}(P)$ for the sets of free and bound names of a process P, respectively. The set of all CCS processes is denoted with $\mathcal{P}$.

Reversible CCS In this work we will use the representation of RCCS appeared in [21] since it simplifies our technical development.[1]

In RCCS, reversibility is obtained by adding a memory to each process. Memories organised as stacks of events record all necessary information about the past actions. Memory event representing the last executed action is positioned on the top of the stack (on the left in the textual representation).

The syntax of RCCS is given in Figure 1. We let $k, h, w, \ldots$ to range over an infinitive denumerable set of action keys, denoted with $\mathcal{K}$ ($\mathcal{K} \cap \mathtt{Act} = \emptyset$) and we let $\mathtt{ActK}_\tau = \mathtt{Act}_\tau \times \mathcal{K}$ be the set of pairs formed by an action α and a key k. Reversible processes are built on the top of CCS by adding a memory to each CCS process. A term $m \triangleright P$ represents a *monitored* process, where m is a *memory* and P is a CCS process. Two RCCS processes can be composed in parallel via $R \mid S$, while $R\backslash a$ indicates that the name a is restricted in the process R. Event $\langle\rangle$ stands for the empty memory. A memory event $\langle k, \alpha, Q\rangle$ keeps track of the executed action, where elements of a triple are action α, identified with the *key* k and alternative process Q, discarded with the execution of the action α. The process can split into two parallel subprocesses what is symbolised with the event $\langle\uparrow\rangle$. We shall introduce the operator @[2] and write $m_1 @ m_2$ for the memory obtained by pushing all the elements in m_1 on top of m_2 (preserving their order). The set of all memories and the set of all monitored processes are denoted with $\mathtt{Mem}$ and $\mathcal{P}_R$, respectively.

As in CCS, the restriction is the only binder in RCCS. The functions $\mathtt{fn}(\cdot)$ and $\mathtt{bn}(\cdot)$ can be extended to RCCS processes and memories.

The semantics of RCCS and more details about the calculus can be found in [1, 21, 29], for the lack of space, we just illustrate it through an example.

The particularity of the RCCS is in the following structural rules:

$$\text{(Split)} \quad m \triangleright (P \mid Q) \equiv \langle\uparrow\rangle \cdot m \triangleright P \mid \langle\uparrow\rangle \cdot m \triangleright Q$$
$$\text{(Res)} \quad m \triangleright P\backslash a \equiv (m \triangleright P)\backslash a \qquad \text{if } a \notin \mathtt{fn}(m)$$

A monitored process with a top-level parallel composition splits into two subprocesses, by applying the rule Split. The memory is duplicated and annotated with

[1] The first approach to reversibility in CCS was introduced in [1]. It differs from the presentation in [21] in a way how memories and splitting of the memory through the parallel composition are handled. As remarked in [21], the two presentations are conceptually the same.

[2] Operator @ is not appearing in [1, 21]. The formal definition can be found in [29].

$\langle\uparrow\rangle$. After that, the obtained monitored processes evolve independently. With the rule Res the restriction operator can be pushed outside of the monitored process, if the restricted name is not in the memory of a process.

We give the following example to illustrate the semantics of RCCS.

Example 1. Let us consider CCS process $P = (a.b + c) \mid d$. The initial RCCS process is $R = \langle\rangle \triangleright P = \langle\rangle \triangleright ((a.b + c) \mid d)$. To execute the action a, process R, needs first to split parallel composition into two separated components by applying structural rule Split:

$$\langle\rangle \triangleright ((a.b + c) \mid d) \equiv \langle\uparrow\rangle \cdot \langle\rangle \triangleright a.b + c \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright d$$

Now, action a identified by key k can be executed. Then we have:

$$\langle\uparrow\rangle \cdot \langle\rangle \triangleright a.b + c \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright d \xrightarrow{k,a} \langle k, a, c\rangle \cdot \langle\uparrow\rangle \cdot \langle\rangle \triangleright b \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright d = R'$$

The executed action a, together with the identifier k and alternative process c is saved into the memory. Resulting process R' is able to execute the backward action a with the key k, by restoring the information from the memory $\langle k, a, c\rangle$.

Any RCCS process R can be thought of as a context C^R built from parallel and restriction operators. The context contains numbered holes filled by monitored processes. Therefore, we write a RCCS process R as $C^R[1 \mapsto m_1 \triangleright P_1, \ldots, n \mapsto m_n \triangleright P_n]$, or more compactly, as $C^R_{i\in\{1,\ldots,n\}}[i \mapsto m_i \triangleright P_i]$. If R is not relevant we may drop it.

CCS with Communication Keys The calculus that we present here is obtained by applying the general approach[3] for reversing process calculi introduced in [2], on the CCS version underlying RCCS. The resulting calculus, we call CCSK since it is very close to the one in [2].

The main ideas behind the approach in [2] are to use communication keys to uniquely identify the actions and to make all the operators of the calculus static. Given that all operators are static, there is no loss of information, hence there is no need to use external memory to record the past action of the process. Performed actions are not discarded, but annotated with the keys and saved in the structure of the process, representing its past.

The syntax of CCSK is depicted in Figure 2. The difference regarding CCS is that the prefix α can be annotated with key k, written as $\alpha[k]$ and that the alternatives in the choice operator are not discarded (Example 2). We will use the same set of keys $\mathcal{K}$ as in RCCS since they have the same role in both calculi. The set of processes from CCSK we denote with $\mathcal{P}_K$.

[3]The approach of [2] can be applied on the process calculi defined by using Structural Operational Semantics (SOS) rules [20].

$$\text{(CCSK Processes)} \quad X, Y ::= \mathbf{0} \mid \sum_{i \in I} \pi_i.X_i \mid (X \mid Y) \mid X \backslash a$$

$$\text{(CCSK Prefixes)} \quad \pi ::= \alpha \mid \alpha[k]$$

Figure 2: CCSK syntax

The restriction operator is the only binder in CCSK and functions $\mathtt{fn}(\cdot)$ and $\mathtt{bn}(\cdot)$ can be extended from CCS to CCSK in an expected way.

The semantics of CCSK and more details about the calculus can be found in [2, 29], here we just illustrate it through an example.

Example 2. Let us consider CCS process $P = (a.b + c) \mid d$ from Example 1 and execution of the action a identified with key k:

$$(a.b + c) \mid d \xrightarrow{a[k]} (a[k].b + c) \mid d = X'$$

We can notice that the executed action a, annotated with the key k, remains in the resulting process. The choice operator is static, hence process c is not discarded. The process $(a[k].b + c) \mid d$ can perform the same forward steps as process $b \mid d$. The backward action removes key k from the process X'.

3 Encodings and Isomorphism between RCCS and CCSK

In this Section, we answer on the question about the relation between two reversible extensions of CCS. CCSK and RCCS differ in the way how information about past actions is stored. In Section 2 we were able to see that in RCCS history information is recorded in the memory attached to each process, while in CCSK information about the executed actions is saved directly in the structure and distributed along the whole process. To manage the memories in RCCS, in particular when two processes in parallel are sharing the same memory, it is necessary to use the structural rule. Duplicating the memory whenever applying the structural rule Split leads to the fact that one process in CCSK can correspond to the composition of several monitored processes in RCCS (Example 3).

We proceed by defining the two encodings and proving their correctness. More details can be found in the PhD thesis of the author [29] and in [32] where our preliminary results appeared.

3.1 Encoding CCSK into RCCS

Considering the differences between two reversible calculi, to encode CCSK process into RCCS process it is necessary that encoding function inductively

$$
\begin{array}{rcl}
[\![X]\!] & = & [\![X, \langle\rangle]\!] \\
[\![P, m]\!] & = & m \triangleright P \\
[\![\alpha[k].X + \sum_{j\in J} \alpha_j.P_j, m]\!] & = & [\![X, \langle k, \alpha, \sum_{j\in J} \alpha_j.P_j\rangle \cdot m]\!] \\
[\![X \mid Y, m]\!] & = & [\![X, \langle\uparrow\rangle \cdot m]\!] \mid [\![Y, \langle\uparrow\rangle \cdot m]\!] \\
[\![X\backslash a, m]\!] & = & [\![X\{^b/_a\}, m]\!]\backslash b \\
 & & \text{if } b \notin \mathtt{fn}(m) \wedge (b = a \vee b \notin \mathtt{fn}(X))
\end{array}
$$

Figure 3: Encoding of CCSK into RCCS

analyses the structure of the CCSK process to be able to reconstruct the final memory of every monitored process. For each annotated prefix in CCSK process, there is a corresponding memory event in RCCS process.

The encoding function $[\![\cdot]\!] : \mathcal{P}_K \to \mathcal{P}_R$ is defined in terms of an auxiliary encoding function $[\![\cdot]\!] : \mathcal{P}_K \times \mathtt{Mem} \to \mathcal{P}_R$, which in addition takes RCCS memory as an argument. Since two functions are easily distinguishable with the number of argument, we will use the same notation for both.

The encoding functions are depicted in Figure 3. The encoding of the CCSK process X initialises by calling $[\![X, \langle\rangle]\!]$ with the empty memory as a parameter. A standard[4] process P with some memory m is translated into the monitored process $m \triangleright P$. The encoding of the non-standard process X is given by structural induction. Considering the choice operator and thanks to properties of CCSK, we have that exactly one alternative corresponds to a non-standard process ($\alpha[k].X$), while the rest of them ($\sum_{j\in J} \alpha_j.P_j$) are CCS processes. The executed (annotated) action α, together with its key k and alternative $\sum_{j\in J} \alpha_j.P_j$, is translated into the memory event $\langle k, \alpha, \sum_{j\in J} \alpha_j.P_j\rangle$. The encoding of the parallel composition needs to be split into two independent encodings, while memory m is duplicated and annotated with $\langle\uparrow\rangle$. To mimic the behaviour of the structural congruence rule Res in RCCS, the encoding rule for restriction needs to avoid capturing free names. It is done by replacing the name a with the new name b which is granted to not occur free by the side conditions. If a is not occurring in the memory m, then one can choose $b = a$.

Example 3. Let $X = b + c[k].(d[h] \mid P)$. The encoding of X can be computed as:

$$
\begin{array}{rl}
[\![X]\!] = [\![X, \langle\rangle]\!] = [\![b + c[k].(d[h] \mid P), \langle\rangle]\!] & \\
= [\![d[h] \mid P, \langle k, c, b\rangle \cdot \langle\rangle]\!] & \\
= [\![d[h], \langle\uparrow\rangle \cdot \langle k, c, b\rangle \cdot \langle\rangle]\!] \mid [\![P, \langle\uparrow\rangle \cdot \langle k, c, b\rangle \cdot \langle\rangle]\!] & \\
= [\![\mathbf{0}, \langle h, d, \mathbf{0}\rangle \cdot \langle\uparrow\rangle \cdot \langle k, c, b\rangle \cdot \langle\rangle]\!] \mid \langle\uparrow\rangle \cdot \langle k, c, b\rangle \cdot \langle\rangle \triangleright P & \\
= \langle h, d, \mathbf{0}\rangle \cdot \langle\uparrow\rangle \cdot \langle k, c, b\rangle \cdot \langle\rangle \triangleright \mathbf{0} \mid \langle\uparrow\rangle \cdot \langle k, c, b\rangle \cdot \langle\rangle \triangleright P = R &
\end{array}
$$

[4]The standard process is the process without history, hence CCS process.

As we can notice CCSK process X corresponds to the composition of two monitored processes in R.

In what follows we prove the operational correspondence of the encoding. Full proofs and more details can be found in [29]. First, we show that by having a reachable CCSK process X and its encoding $R = [\![X, \langle\rangle]\!]$, if X does an action α in CCSK, the same action can be done by the process R in RCCS. Resulting process in RCCS is equal to the encoding of the resulting process in CCSK.

Theorem 1 (Forward Correctness)**.** *Let X be a reachable CCSK process and $R = [\![X, \langle\rangle]\!]$. For each CCSK transition $X \xrightarrow{\alpha[k]} X'$ there exists a corresponding RCCS transition $R \xrightarrow{k,\alpha} R'$ with $[\![X', \langle\rangle]\!] = R'$.*

Theorem 2 (Backward Correctness)**.** *Let X be a reachable CCSK process and $R = [\![X, \langle\rangle]\!]$. For each CCSK transition $X \overset{\alpha[k]}{\rightsquigarrow} X'$ there exists a corresponding RCCS transition $R \overset{k,\alpha}{\rightsquigarrow} R'$ with $[\![X', \langle\rangle]\!] = R'$.*

Now we need to show the opposite direction, that every transition of the process $R = [\![X, \langle\rangle]\!]$ in RCCS, can be mimic by process X in CCSK, where the resulting process in RCCS is congruent with the encoding of the resulting process in CCSK.

Theorem 3 (Forward Completeness)**.** *Let X be a reachable CCSK process and $R = [\![X, \langle\rangle]\!]$. For each RCCS transition $R \xrightarrow{k,\alpha} R'$ there exists a corresponding CCSK transition $X \xrightarrow{\alpha[k]} X'$ with $R' \equiv [\![X', \langle\rangle]\!]$.*

Theorem 4 (Backward Completeness)**.** *Let X be a reachable CCSK process and $R = [\![X, \langle\rangle]\!]$. For each RCCS transition $R \overset{k,\alpha}{\rightsquigarrow} R'$ there exists a corresponding CCSK transition $X \overset{\alpha[k]}{\rightsquigarrow} X'$ with $R' \equiv [\![X', \langle\rangle]\!]$.*

3.2 Encoding RCCS into CCSK

The fact that the composition of several monitored processes in RCCS might correspond to the single process in CCSK requires that the encoding procedure starts from the very first action that RCCS process did (the right in the textual representation). While translating parallel components of the RCCS process separately (local), it is required that encoding has knowledge about the whole process (global). In this way, partially encoded processes which have the same part of the memory will be merged (these are the processes which were split during the forward execution by applying the rule Split).

Before defining the encoding, we need to introduce an auxiliary function and a history context. We define a *trimming function* δ to remove the split events $\langle\uparrow\rangle$ from

$$
\begin{aligned}
(\!|R\backslash a|\!) &= (\!|R|\!)\backslash a \\
(\!|C_{l\in L}[l \mapsto m_l@(\langle\uparrow\rangle \cdot m) \triangleright P_l]|\!) &= \mathtt{H}_m[C_{l\in L}[l \mapsto (\!|\delta(m_l) \triangleright P_l|\!)]] \\
&\quad \text{if } \langle\uparrow\rangle \notin m \text{ and the top operator in } C \text{ is } | \\
(\!|m \triangleright P|\!) &= \mathtt{H}_m[P]
\end{aligned}
$$

Figure 4: Encoding of RCCS into CCSK

the memory, starting from the right side of the memory. For example, the result of applying δ to the memory $\langle\uparrow\rangle \cdot \langle k, \alpha, Q\rangle \cdot \langle\uparrow\rangle \cdot \langle\uparrow\rangle \cdot \langle\rangle$ will be $\langle\uparrow\rangle \cdot \langle k, \alpha, Q\rangle \cdot \langle\rangle$. We define the trimming function δ formally as:

Definition 1. *The function* $\delta : \mathtt{Mem} \to \mathtt{Mem}$*, is inductively defined as follows:*

$$\delta(\langle\rangle) = \langle\rangle \qquad \delta(m@\langle\uparrow\rangle \cdot \langle\rangle) = \delta(m) \qquad \delta(m@\langle k, \alpha, Q\rangle \cdot \langle\rangle) = m@\langle k, \alpha, Q\rangle \cdot \langle\rangle$$

The *history context* represents the correspondence between RCCS memory without element $\langle\uparrow\rangle$ and the CCSK context. Formally:

Definition 2 (History Context)**.** *Given a memory* m *such that* $\langle\uparrow\rangle \notin m$*, the corresponding* history context $\mathtt{H}_m$ *is defined as follows:*

$$\mathtt{H}_{\langle\rangle} = \bullet \qquad \mathtt{H}_{\langle k,\alpha,\mathbf{0}\rangle\cdot m} = \mathtt{H}_m[\alpha[k].\bullet] \qquad \mathtt{H}_{\langle k,\alpha,Q\rangle\cdot m} = \mathtt{H}_m[\alpha[k].\bullet + Q] \quad \textit{if } Q \neq \mathbf{0}$$

To illustrate definition above, consider RCCS process $\langle h, b, \mathbf{0}\rangle \cdot \langle k, a, Q\rangle \triangleright P$. History context, of the given memory $\langle h, b, \mathbf{0}\rangle \cdot \langle k, a, Q\rangle$, is defined as:

$$\mathtt{H}_{\langle h,b,\mathbf{0}\rangle\cdot\langle k,a,Q\rangle}[P] = \mathtt{H}_{\langle k,a,Q\rangle}[b[h].P] = \mathtt{H}_{\langle\rangle}[a[k].b[h].P + Q].$$

Finally, the encoding function $(\!|\cdot|\!) : \mathcal{P}_R \to \mathcal{P}_K$ is inductively defined on Figure 4. Let us comment on the rules. The restriction rule is defined to push restriction outside of the encoding. The function[5] $(\!|\cdot|\!)$ analyses the memory starting from the oldest element (right in the textual representation) and by taking the common part m such that $\langle\uparrow\rangle \notin m$, reconstructs the CCSK structure with the history context $\mathtt{H}_m$. The first split element ($\langle\uparrow\rangle$) from the right is discarded and the rest of the memories m_l are being trimmed by function δ and kept for further encoding. Single monitored process $m \triangleright P$ is encoded by translating the memory m into the history context $\mathtt{H}_m$. The intuition of how encoding works is given in the following example.

Example 4. Let us take the RCCS process R (the resulting process in Example 3): $R = \langle h, d, \mathbf{0}\rangle \cdot \langle\uparrow\rangle \cdot \langle k, c, b\rangle \cdot \langle\rangle \triangleright \mathbf{0} \mid \langle\uparrow\rangle \cdot \langle k, c, b\rangle \cdot \langle\rangle \triangleright P$

[5]We would like to remark that in the second rule, since the context C is composed by parallel and restriction operators available in both calculi, we use the same notation for the context.

Its encoding in CCSK is:

$$
\begin{aligned}
(\!|R|\!) &= (\!|\langle h, d, \mathbf{0}\rangle \cdot \langle\uparrow\rangle \cdot \langle k, c, b\rangle \cdot \langle\rangle \triangleright \mathbf{0} \mid \langle\uparrow\rangle \cdot \langle k, c, b\rangle \cdot \langle\rangle \triangleright P|\!) \\
&= \mathtt{H}_{\langle k,c,b\rangle\cdot\langle\rangle}[(\!|\delta(\langle h, d, \mathbf{0}\rangle \cdot \langle\rangle) \triangleright \mathbf{0}|\!) \mid (\!|\delta(\langle\rangle) \triangleright P|\!)] \\
&= \mathtt{H}_{\langle k,c,b\rangle\cdot\langle\rangle}[(\!|\langle h, d, \mathbf{0}\rangle \cdot \langle\rangle \triangleright \mathbf{0}|\!) \mid (\!|\langle\rangle \triangleright P|\!)] \\
&= \mathtt{H}_{\langle k,c,b\rangle\cdot\langle\rangle}[\mathtt{H}_{\langle h,d,\mathbf{0}\rangle\cdot\langle\rangle}[\mathbf{0}] \mid \mathtt{H}_{\langle\rangle}[P]] \\
&= \mathtt{H}_{\langle k,c,b\rangle\cdot\langle\rangle}[d[h] \mid P] \\
&= b + c[k].(d[h] \mid P)
\end{aligned}
$$

In what follows, we show the correctness of the encoding in terms of the operational correspondence. Full proofs and more details can be found in [29]. We prove that every action done by RCCS process R can bi mimicked by its encoding $(\!|R|\!)$ in CCSK and vice versa.

Theorem 5 (Forward Correctness and Completeness)**.** *Let R and S be two reachable RCCS processes. There exists an RCCS transition $R \xrightarrow{k,\alpha} S$ iff there exists a CCSK transition $(\!|R|\!) \xrightarrow{\alpha[k]} (\!|S'|\!)$ with $S \equiv S'$.*

Theorem 6 (Backward Correctness and Completeness)**.** *Let R and S be two reachable RCCS processes. There exists an RCCS transition $R \overset{k,\alpha}{\rightsquigarrow} S$ iff there exists a CCSK transition $(\!|R|\!) \overset{\alpha[k]}{\rightsquigarrow} (\!|S'|\!)$ with $S \equiv S'$.*

3.3 Isomorphism and cross-fertilisation results

In this Section, we show that the LTS of RCCS up to structural congruence and the LTS of CCSK up to α-conversion and normal form (Definition 4), are isomorphic. After that, we mention some cross-fertilisation results and argue about the *uniform* encoding. More details can be found in [29].

We start by defining the isomorphism between two LTSs.

Definition 3 (LTS Isomorphism)**.** *Two LTSs $LTS_i : \langle \mathcal{P}_i, \mathcal{L}_i, \rightarrow_i \rangle$, with $i \in \{1, 2\}$ are* isomorphic *iff there exist two bijective functions $\gamma_L : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ and $\gamma_P : \mathcal{P}_1 \rightarrow \mathcal{P}_2$ such that $P_1 \xrightarrow{\alpha} P_2$ iff $\gamma_P(P_1) \xrightarrow{\gamma_L(\alpha)} \gamma_P(P_2)$.*

Before giving the theorem of isomorphism, we need to introduce the notion of a *normal form* for CCSK processes. It is defined to push all the restrictions in the non-standard part of a sequential process on the top-level.

Definition 4 (CCSK normal form)**.** *The normal form $\mathtt{nf}(X)$ of a CCSK process X is a CCSK process Y obtained from X by applying as many times as possible the following rewriting rules (in any context):*

- $\alpha[k].(X_1 \backslash a) \rightarrow (\alpha[k].X_1\{^b/_a\})\backslash b \quad \textit{if } b \notin \mathtt{fn}(\alpha.\mathbf{0}) \wedge (b = a \vee b \notin \mathtt{fn}(X_1))$
- $\alpha[k].(X_1 \backslash a) + Q \rightarrow (\alpha[k].X_1\{^b/_a\} + Q)\backslash b \quad \textit{if } b \notin \mathtt{fn}(\alpha.Q) \wedge (b = a \vee b \notin \mathtt{fn}(X_1))$

where X_1 is not a standard process.

Now, we give the equivalence relations for both calculi and show the isomorphism between their LTSs. Looking at the RCCS processes, we will consider structural congruence as the equivalence relation. On CCSK processes, we define the equivalence relation $\overset{ccsk}{\sim}$ as: $X \overset{ccsk}{\sim} Y$ iff $\mathtt{nf}(X) =_\alpha \mathtt{nf}(Y)$.

Finally, we can state our main result in this Section and show that forward LTSs of RCCS and CCSK are isomorphic. The same is holding for the backward ones.

Theorem 7 (RCCS and CCSK are isomorphic)**.** *The forward LTS of RCCS up to structural congruence is isomorphic to the forward LTS of CCSK up to $\overset{ccsk}{\sim}$. Both LTSs are restricted to reachable processes. The same result holds for backward LTSs.*

It is very important to remark that the result of isomorphism between two calculi, implies that they can be equated by any behavioural equivalence, such as bisimilarity or trace equivalence.

The proof of isomorphism relies on the operational correspondence results for two encodings that we presented in Sections 3.2 and 3.1. The interesting fact about encodings is that none of them is uniform [34]. Furthermore, uniform encoding cannot exist since reachable RCCS processes are not closed under the parallel composition [29, Proposition 2]. Hence, there does not exist the encoding from CCSK into RCCS which maps parallel operator of CCSK into a parallel operator of RCCS (what is required for uniform encoding). If we consider the encoding from RCCS into CCSK, it is not possible to state uniformity result, since terms in the statement would not be reachable. The fact that the uniform encoding does not exist, has no impact on our result of isomorphism.

Cross-fertilisation results The result of isomorphism between two reversible calculi, we use to bring some properties from one calculus to the other one. Namely, we took the *Reverse Diamond Property*, stating that backward transitions are confluent, from CCSK and import it into RCCS. In the opposite direction, from RCCS into CCSK, we brought *Parabolic Lemma*, stating that reversible computation can be decomposed into a backward only computation followed by forward only computation. We remark that CCSK Parabolic Lemma has been proved in the literature [2, Lemma 5.12], still, the direct proof is more complex than importing the result from RCCS. For more details, check [29, Section 3.5].

4 Causality and Reversibility in π-calculus

In π-calculus, Boreale and Sangiorgi in [25] separated causal dependencies between the actions into two categories: the first one is the subject (structural) dependency which is, as in CCS, determined by the nesting of the prefixes and by synchronisations; the second one is the object (link) dependency which is generated by extruding (or opening) a name. Different understanding of the object dependency generates different causal semantics for π-calculus. Through the problem of a parallel extrusion of the same name, we will give the intuition about the three causal semantics that we considered in this work. More information about causal semantics can be found in [25, 28, 22, 29]. The common example that we will use is the process $va(\overline{b}a \mid \overline{c}a \mid a(x))$ where actions are executed in the following order $\overline{b}a, \overline{c}a, a(x)$.

Causal semantics by Boreale and Sangiorgi In [25], the authors introduce a compositional causal semantics for standard π-calculus (forward only)[6]. As we mentioned above, two forms of dependencies are considered: subject and the object dependency. To capture the subject dependency, the authors introduce a causal term $\mathtt{K} :: A$, where the set of causes $\mathtt{K}$ records that every action performed by A depends on the set $\mathtt{K}$. Every visible transition is bound with unique cause k. The object causality can be detected by observing the trace (run) of the process. The first action that extrudes a bound name will cause every further action using that name in the subject or in the object position of the label.

Considering our common example, we will have that the first action that extrudes bound name a is $\overline{b}a$. The other two actions are caused by action $\overline{b}a$ since they contain name a in object ($\overline{c}a$) and subject ($a(x)$) position of a label.

Causal semantics by Crafa, Varacca and Yoshida A compositional event structure semantics for π-calculus (forward only) is introduced in [28]. The pair $(\mathbf{E}, \mathbf{X})$ where $\mathbf{E}$ is a prime event structure and $\mathbf{X}$ is a set of bound names, represents a π-calculus process. The structural causality is captured by causal relation of $\mathbf{E}$, while the object one is defined through the notion of *permitted configuration* (the configuration that contains the action with a bound subject and the action which extruded the bound name). The authors define the disjunctive object causality with the so-called inclusive approach according to which the requirement to execute an action with a bound subject is that at least one of the extruders of a bound name was executed previously. It is not necessary to remember the exact one.

Considering our common example and process $va(\overline{b}a \mid \overline{c}a \mid a(x))$, we are

[6]Causal semantics introduced in [25] is defined for a polyadic π-calculus, with an early input semantics. In this work, to have a direct correspondence with our framework, we adapt it to work with monadic π-calculus and late input semantics.

interested in the permitted configuration $\{\overline{b}a, \overline{c}a, a(x)\}$. From it, we can notice that one of the actions $\overline{b}a$ and $\overline{c}a$ is the cause of the action $a(x)$, but we do not know which one exactly.

Causal semantics for reversible π-calculus by Cristescu, Krivine and Varacca The first compositional reversible semantics for π-calculus $R\pi$, is introduced in [22]. The reversibility is obtained by adapting the approach given in [1, 21] to π-calculus. Therefore, the memory recording the information about the past actions is added to every process. The reversible process is represented by the term $m \triangleright P$, where memory m is a stack of events and P is the process itself. There are two types of memory events: the event recording the executed action π, its unique identifier i and the contextual cause[7] k, written as $\langle i, k, \pi \rangle$; and the event symbolising that the process split during its computation, written as $\langle \uparrow \rangle$ (as in RCCS, before executing in parallel, the structural rule needs to be applied). The indexed restriction νa_Γ, behaves as the classical π-calculus restriction when set Γ is empty, otherwise, it represents the memory and keeps track about the extruders of the name a. As we will see, the indexed restriction νa_Γ was the inspiration for our parametric indexed restriction νa_Δ. The significance of the semantics given in [22] is in the several correctness criteria for causal models [30] which it satisfies, what is not the case for the other causal semantics considered in our work.

In our common example, we have that the action $a(x)$ chooses as its cause one of the concurrent actions $\overline{b}a$ and $\overline{c}a$.

5 Parametric Framework for Reversible π-Calculi

A general approach to reverse CCS-like calculi defined through SOS rules is given in [2]. Here, we expand this idea and adapt it to a more complex setting, π-calculus. Let us start from CCSK, for instance, if we consider the process $P = a.Q_1 \mid \overline{a}.Q_2$ the synchronisation between parallel components of the process P is:

$$a.Q_1 \mid \overline{a}.Q_2 \xrightarrow{\tau[i]} a[i].Q_1 \mid \overline{a}[i].Q_2$$

As we highlighted in Section 2, prefixes are not discarded, but annotated with the key i and used only for the backward steps.

By moving to the π-calculus, possibility of creating new channel names and treating channels as sent values is enabled. For instance, we could adapt the process P to π-calculus, and have the computation:

$$a(x).Q_1 \mid \overline{a}b.Q_2 \xrightarrow{i:\tau} a(x)[i].Q_1\{^{b^i}/_x\} \mid \overline{a}b[i].Q_2$$

[7] Contextual cause is used to keep track of the causalities imposed by the extrusions. More information can be found in [22].

$$X,\ Y ::= \quad \mathbf{P} \mid\ \overline{b}^{j} a^{j_1}[i, K].X \ \mid\ b^{j}(x)[i, K].X \ \mid\ X \mid Y \ \mid \nu a_{\Delta}(X)$$
$$P,\ Q ::= \quad \mathbf{0} \mid \overline{b}a.P \mid b(x).P \mid\ P \mid Q \ \mid \nu a(P)$$

Figure 5: Syntax of the framework

In the input subprocess it is necessary to apply the substitution $\{^{b^i}/_x\}$ on the continuation Q_1. Variable x is substituted by the name b decorated with the key i of the executed action. In this way decoration on the name keeps track of the substitution.

By choosing to keep the history information of the process statically, as in [2] there is no necessity of using the structural rule Split[8] of $R\pi$ [22].

To be able to capture causalities in π-calculus, the framework has to keep track of the extruders of a name. As in [22] this can be done by using the indexed restrictions and contextual causes. For instance in the computation

$$\nu a\,(\overline{b}a \mid a(x).P) \xrightarrow{i:\overline{b}\langle \nu a\rangle} \nu a_{\{i\}}(\overline{b}a[i] \mid a(x).P) \xrightarrow{j:a(x)} \nu a_{\{i\}}(\overline{b}a[i] \mid a(x)[j, i].P)$$

after the execution of the action $i : \overline{b}\langle \nu a\rangle$, restriction νa becomes the memory $\nu a_{\{i\}}$ where the index i symbolises that name a is extruded by the action identified with i. In the resulting process, the name a is free and the input action on the channel a is enabled. Since the first action extruded the name a, in the process $a(x)[j, i].P$, identifier i is saved as the contextual cause of the action $a(x)$.

5.1 Syntax of the framework

In this Section, we present the syntax of the framework. We let $\mathcal{N}$, $\mathcal{K}$ and $\mathcal{V}$ be the denumerable infinite mutually disjoint sets of names, keys and variables, respectively. The symbol $*$ is a distinguished key such that $\mathcal{K}_* = \mathcal{K} \cup \{*\}$. We let a, b, c range over set $\mathcal{N}$, i, j, k range over set $\mathcal{K}$ and x, y range over set $\mathcal{V}$.

The syntax of the framework is depicted in Figure 5. P, Q productions represent the π-calculus processes. The idle process is denoted with $\mathbf{0}$. The name a can be sent over a channel b and it is represented by the output prefixed process $\overline{b}a.P$. The input prefixed process $b(x).P$ indicates the possibility of receiving some name over the channel b and bounding it to the variable x. Parallel and restriction operator are represented with $P \mid Q$ and $\nu a(P)$, respectively.

Productions X, Y represent *reversible processes* and they are defined on the top of π-calculus processes. Similarly as in CCSK, executed actions are not discarded, but annotated and preserved in the structure of a process, representing its *history*. A reversible process $\mathbf{P}$ is a π-calculus process P decorated with the instantiators (the behaviours of the processes $\mathbf{P}$ and P are the same). We use the instantiators to keep track of the substitutions. With the *past output*, we call the prefix $\overline{b}^{j} a^{j_1}[i, K]$

[8]The rule Split is not associative and as a consequence has that equivalent processes performing the same action may become non-equivalent processes (shown in [35]).

stating that the output action identified by key i was executed and its contextual cause was $K \subseteq \mathcal{K}_*$. The *past input* given with prefix $b^j(x)[i, K]$ is representing the input action (with the key i and the contextual cause set K), executed in the past. We shall use notation b^* and $K = \{*\}$ to specify that name b has no instantiator and that executed action is not caused by any other action. Two reversible processes can be composed in parallel as $X \mid Y$. Following [22], we use the restriction νa_Δ decorated with the memory Δ to record the extruders of the name a. νa_Δ will act as a π-calculus restriction operator νa when Δ is empty (we will give the precise definition in the further text). The set of reversible processes is denoted by $\mathcal{X}$.

Now we define the notion of *history* and *general* contexts. A history context represents the past prefixes of a process. For instance, the process $X = \overline{b}^* a^*[i, K].\overline{c}^* a^*[i', K'].\mathbf{P}$ can be written as $X = \mathtt{H}[\mathbf{P}]$, where $\mathtt{H} = \overline{b}^* a^*[i, K].\overline{c}^* a^*[i', K']$. A general context is defined on the top of the history context and it is composed from parallel and restriction operators. For instance, the process $Z \mid Y \mid X$ can be written as $C[X]$ where $C = Z \mid Y$. Formally:

Definition 5 (History and General context)**.** *History contexts* $\mathtt{H}$ *and general contexts* C *are reversible processes defined by the following grammar:*

$$\mathtt{H} ::= \mathtt{H} \mid \pi[i, K].\mathtt{H} \qquad\qquad C ::= \mathtt{H} \mid X \mid C \mid \nu a_\Delta(C)$$

In the framework, there are two constructs with binders: $\nu a_\Delta(X)$ when Δ is empty, in which the scope of the name a is process X; and $b(x).\mathbf{P}$ in which scope of the variable x is process $\mathbf{P}$. We denote with $\mathtt{bn}(X)$ and $\mathtt{fn}(X)$ sets of bound and free names of the process X.

Data structure Δ The framework is parametric with respect to the data structure Δ and in what follows, we give the operation defined on it.

Definition 6. Δ *is a data structure with the following defined operations:*

(*i*) $\mathtt{init} : \Delta \to \Delta$ *initialises the data structure*

(*ii*) $\mathtt{empty} : \Delta \to \mathtt{bool}$ *predicate telling whether* Δ *is empty*

(*iii*) $+ : \Delta \times \mathcal{K} \to \Delta$ *operation adding a key to* Δ

(*iv*) $\#i : \Delta \times \mathcal{K} \to \Delta$ *operation removing a key from* Δ

(*v*) $\in : \Delta \times \mathcal{K} \to \mathtt{bool}$ *predicate telling whether a key belongs to* Δ

The three instances of the data structure Δ that we use are: sets, sets indexed with an element and sets indexed with a set.

Set. If the data structure is a set Γ, then it contains the keys of the actions that extruded a bound name (i.e. $\Gamma \subseteq \mathcal{K}$).

Definition 7 (Operations on a set)**.** *The operations on a set* Γ *are defined as:*

(*i*) $\texttt{init}(\Gamma) = \emptyset$ (*iv*) $\#i$ *is defined as the identity* $\Gamma_{\#i} = \Gamma$

(*ii*) $\texttt{empty}(\Gamma) = \mathit{true}$, *when* $\Gamma = \emptyset$ (*v*) $i \in \Gamma$ *the key* i *belongs to the set* Γ

(*iii*) $+$ *is the addition of elements to a set*

Indexed set. If the data structure is an indexed set Γ_w, then Γ is a set containing keys of the actions that extruded a bound name ($\Gamma \subseteq \mathcal{K}$) and w is the key of the very first action that extruded a bound name. If no extruder of a bound name is executed, we write $w = *$.

Definition 8 (Operations on an indexed set)**.** *The operations on an indexed set* Γ_w *are defined as:*

(*i*) $\texttt{init}(\Gamma_w) = \emptyset_*$ (*ii*) $\texttt{empty}(\Gamma_w) = \mathit{true}$, *when* $\Gamma = \emptyset \ \wedge \ w = *$

(*iii*) *operation* $+$ *is defined as:* $\Gamma_w + i = \begin{cases} (\Gamma \cup \{i\})_i, & \text{when } w = * \\ (\Gamma \cup \{i\})_w, & \text{when } w \neq * \end{cases}$

(*iv*) *operation* $\#i$ *is defined inductively as:*

$(X \mid Y)_{\#i} = X_{\#i} \mid Y_{\#i}$ $(\texttt{H}[X])_{\#i} = \texttt{H}[X_{\#i}]$ $(\mathbf{P})_{\#i} = \mathbf{P}$

$(\nu a_{\Gamma_i} X)_{\#i} = \nu a_{\Gamma_*} X_{\#i}$ $(\nu a_{\Gamma_w} X)_{\#i} = \nu a_{\Gamma_w} X_{\#i}$

(*v*) $i \in \Gamma_w$ *the key* i *belongs to the set* Γ*, regardless of* w

The data structure is initialised with $\texttt{init}(\Gamma_w) = \emptyset_*$ what implies that $\texttt{empty}(\Gamma_w) = \mathit{true}$. If $w = *$, the $+$ operator adds the key i to the set Γ and on the place of w, otherwise the $+$ adds the key i just to the set Γ. For instance, after adding key i_3 to the $\{i_1, i_2\}_*$ we obtain $\{i_1, i_2, i_3\}_{i_3}$. The operation $\#i$ is defined to substitute the value of w in Γ_w with element $*$, when $w = i$. For instance, the result of applying operation $\#i$ on $\{i, i_1\}_i$ is $\{i, i_1\}_*$. The operation $\in$ is defined on the set Γ, regardless the index w. For instance, the key i belongs to the indexed set $\{i_1, i\}_{i_1}$.

Set indexed with a set. If the data structure is a set indexed with the other set Γ_Ω, then Γ is the same is in previous cases, while $\Omega \in \mathcal{K}_*$ is a set which contains keys of the actions which extruded bound name and are not part of the communications. When there is no any key in Ω, we write $\Omega = \{*\}$.

Definition 9 (Operations on a set indexed with a set)**.** *The operations on a set indexed with a set* (Γ_Ω) *are defined as:*

(*i*) $\texttt{init}(\Gamma_\Omega) = \emptyset_{\{*\}}$ (*ii*) $\texttt{empty}(\Gamma_\Omega) = \mathit{true}$, *when* $\Gamma = \emptyset \ \wedge \ \Omega = \{*\}$

(*iii*) *operation* $+$ *is defined as:* $(\Gamma_\Omega) + i = (\Gamma \cup \{i\})_{(\Omega \cup \{i\})}$

(*iv*) $\#i$ *is defined inductively as:*

$(X \mid Y)_{\#i} = X_{\#i} \mid Y_{\#i}$ $(\nu a_{\Gamma_\Omega} X)_{\#i} = \nu a_{\Gamma_{\Omega \setminus \{i\}}} X_{\#i}$

$(\texttt{H}[X])_{\#i} = \texttt{H}[X_{\#i}]$ $(\mathbf{P})_{\#i} = \mathbf{P}$

(*v*) $i \in \Gamma_\Omega$ *the key* i *belongs to the set* Γ*, regardless of* Ω

We initialise the data structure with $\texttt{init}(\Gamma_\Omega) = \emptyset_{\{*\}}$. The + operator adds the key into both sets Γ and Ω. For example, the key i added into the data structure $\{i_1, i_2\}_{\{i_2\}}$ results in $\{i_1, i_2, i\}_{\{i_2,i\}}$. With the operation $\#i$ key i is removed from the set Ω. For example, operation $\#i$ applied to the data structure $\{i_1, i\}_{\{i_1,i\}}$ gives $\{i_1, i\}_{\{i_1\}}$ (key i is deleted from $\Omega = \{i_2, i\}$). The operation $\in$ is defined on the set Γ, regardless to the set Ω. For instance, the key i belongs to $\{i_1, i\}_{\{i_1\}}$.

5.2 Operational semantics of the framework

We define the grammar of the labels on the transition $t : X \xrightarrow{\mu} Y$ as:

$$\mu ::= (i, K, j) : \alpha \qquad\qquad \alpha ::= \overline{b}a \mid b(x) \mid \overline{b}\langle \nu a_\Delta \rangle \mid \tau$$

where the triple (i, K, j) consists of the key i which uniquely identifies the action α, the contextual cause set $K \subseteq \mathcal{K}_*$ and the instantiator $j \in \mathcal{K}_*$ of the action α. The possible executions for the action α are: the input and the output on the channel b, symbolised with $b(x)$ and $\overline{b}a$, respectively; the τ-action; the action $\overline{b}\langle \nu a_\Delta \rangle$ representing the bound output from the π-calculus when Δ is empty, otherwise $\overline{b}\langle \nu a_\Delta \rangle$ stands for free output decorated with a memory Δ. With the following definition, we give the operational semantics of the framework where the set of the all possible labels is $\mathcal{L} = \mathcal{K} \times \mathcal{K}_* \times \mathcal{K}_* \times \mathcal{A}$.

Definition 10 (Operational Semantics)**.** *The operational semantics of the reversible framework is given as a pair of two LTSs defined on the same set of reversible processes and set of labels: a forward LTS* $(\mathcal{X}, \mathcal{L}, \twoheadrightarrow)$ *and a backward LTS* $(\mathcal{X}, \mathcal{L}, \rightsquigarrow)$*. We define* $\rightarrow = \twoheadrightarrow \cup \rightsquigarrow$*, where* $\twoheadrightarrow$ *is the least transition relation induced by the rules in Figures 6 and 7; and* $\rightsquigarrow$ *is the least transition relation induced by the rules in Figure 8.*

To have a direct access to the set of keys in a given process, we define the function $\texttt{key}(\cdot)$ as:

Definition 11 (Process keys)**.** *The set of* communication keys *of a process X, written* $\texttt{key}(X)$*, is inductively defined as follows:*

$$\texttt{key}(X \mid Y) = \texttt{key}(X) \cup \texttt{key}(Y) \qquad \texttt{key}(\pi[i, K].X) = \{i\} \cup \texttt{key}(X)$$
$$\texttt{key}(\nu a_\Delta(X)) = \texttt{key}(X) \qquad \texttt{key}(\mathbf{P}) = \emptyset$$

A key i is fresh *in a process X, written* $\texttt{fresh}(i, X)$ *if* $i \notin \texttt{key}(X)$.

The semantics of the framework is defined with a *late* input semantics. We divide the forward rules into two groups depending on whether they are common to all instances of the framework or they are parametric with respect to Δ.

Common rules, which are independent of the data structure, are given in Figure 6. In rules Out1 and In1, executed actions are preserved in the structure of

$$\text{(Out1)}\ \overline{b}^{j}a^{j_1}.\mathbf{P} \xrightarrow{(i,K,j):\overline{b}a} \overline{b}^{j}a^{j_1}[i,K].\mathbf{P} \qquad \text{(In1)}\ b^{j}(x).\mathbf{P} \xrightarrow{(i,K,j):b(x)} b^{j}(x)[i,K].\mathbf{P}$$

$$\text{(Ind)}\ \frac{X \xrightarrow{(i,K,j):\alpha} X' \quad \alpha = \overline{b}a \vee \alpha = b(x) \wedge \mathtt{fresh}(i, \mathtt{H}[X])}{\mathtt{H}[X] \xrightarrow{(i,K,j):\alpha} \mathtt{H}[X']}$$

$$\text{(Par)}\ \frac{X \xrightarrow{(i,K,j):\alpha} X' \quad i \notin Y \quad \mathtt{bn}(\alpha) \cap \mathtt{fn}(Y) = \emptyset}{X \mid Y \xrightarrow{(i,K,j):\alpha} X' \mid Y}$$

$$\text{(Com)}\ \frac{X \xrightarrow{(i,K,j):\overline{b}a} X' \quad Y \xrightarrow{(i,K',j'):b(x)} Y' \quad K =_* j' \ \wedge\ K' =_* j}{X \mid Y \xrightarrow{(i,*,*):\tau} X' \mid Y'\{^{a^i}/_x\}}$$

Figure 6: Common rules for all instances of the framework.

the process and annotated with the memory $[i, K]$, where i is the fresh key linked to the executed action and K is its cause set. The rule Ind allows a prefixed process $\mathtt{H}[X]$ to perform a forward step if process X can do it. In the rule Par the action α can be executed only under the condition that key i is not used by other processes in parallel (condition $i \notin Y$ guarantees uniqueness of the keys). The synchronisation between two processes is possible through the rule Com if the side condition is satisfied. $K =_* j$ stands for: $K = j$ or $* \in K$ or $j = *$; for instance, if $K = \{*, i_1\}$ and $j = i_2$, equality holds since $* \in K$. This condition is necessary only to capture the causal semantics of [22], while it holds by default for other semantics since $* \in K$ will be always true. The substitution needs to be applied in the process Y' where every occurrence of variable $x \in \mathtt{fn}(Y')$ is substituted with the name a decorated with the key i of the executed action. Decoration i in a^i is called instaniator and it is used to remember that the substitution happened during the action with the key i. The instaniators do not define the names, for instance having the reversible process $\overline{b}^{j}a^{*}.\mathbf{P} \mid b^{j'}(x).\mathbf{P}'$, the communication between two processes in parallel is allowed even if they do not have the same instantiators on the channel b.

In order to have better intuition about the rules presented above, we give the following example.

Example 5. Let $X = \overline{b}^{*}a^{*}.\overline{d}^{*}e^{*} \mid d^{*}(x).\overline{x}c^{*}$ be a reversible process. First we can execute the output action $\overline{b}a$ with the key i, and we have:

$$\overline{b}^{*}a^{*}.\overline{d}^{*}e^{*} \mid d^{*}(x).\overline{x}c^{*} \xrightarrow{(i,*,*):\overline{b}a} \overline{b}^{*}a^{*}[i,*].\overline{d}^{*}e^{*} \mid d^{*}(x).\overline{x}c^{*}$$

After that, two processes can communicate over the channel d and we have:

$$\frac{\overline{b}^{*}a^{*}[i,*].\overline{d}^{*}e^{*} \xrightarrow{(i',*,*):\overline{d}e} \overline{b}^{*}a^{*}[i,*].\overline{d}^{*}e^{*}[i',*] \qquad d^{*}(x).\overline{x}c^{*} \xrightarrow{(i,*,*):d(x)} d^{*}(x)[i'*].\overline{x}c^{*}}{\overline{b}^{*}a^{*}[i,*].\overline{d}^{*}e^{*} \mid d^{*}(x).\overline{x}c^{*} \xrightarrow{(i',*,*):\tau} \overline{b}^{*}a^{*}[i,*].\overline{d}^{*}e^{*}[i',*] \mid d^{*}(x)[i',*].\overline{e}^{i'}c^{*}}$$

$$\text{(Cause Ref)}\ \frac{X \xrightarrow{(i,K,j):\alpha} X' \quad a \in sub(\alpha) \qquad \texttt{empty}(\Delta) \neq true \quad \texttt{Cause}(\Delta, K, K')}{\nu a_\Delta(X) \xrightarrow{(i,K',j):\alpha} \nu a_\Delta(X'_{[K'/K]@i})}$$

$$\text{(Open)}\ \frac{X \xrightarrow{(i,K,j):\alpha} X' \ \alpha = \overline{b}a \vee \alpha = \overline{b}\langle \nu a_{\Delta'} \rangle \ \texttt{Update}(\Delta, K, K')}{\nu a_\Delta(X) \xrightarrow{(i,K',j):\overline{b}\langle \nu a_\Delta \rangle} \nu a_{\Delta+i}(X'_{[K'/K]@i})}$$

$$\text{(Close)}\ \frac{X \xrightarrow{(i,K,j):\overline{b}\langle \nu a_\Delta \rangle} X' \quad Y \xrightarrow{(i,K',j'):b(x)} Y' \quad K =_* j' \ \wedge \ K' =_* j}{X \mid Y \xrightarrow{(i,*,*):\tau} \nu a_\Delta(X'_{\#i} \mid Y'\{^{a^i}/_x\})}$$

$$\text{(Res)}\ \frac{X \xrightarrow{(i,K,j):\alpha} X' \quad a \notin \alpha}{\nu a_\Delta(X) \xrightarrow{(i,K,j):\alpha} \nu a_\Delta(X')}$$

Figure 7: Parametric rules

The communication was possible since in process $\overline{b}^* a^*[i, *].\overline{d}^* e^*$ we have that $j = *$ and $K = \{*\}$ and in process $d^*(x).\overline{x}c^*$ we have $j' = *$ and $K' = \{*\}$, hence premiss of the rule Com is satisfied. We can notice that during the synchronisation, variable x was substituted with the received name e decorated with the key i'.

Parametric rules are depicted in Figure 7. Depending on the data structure that is used, a different mechanism for choosing contextual cause needs to be applied. This is the reason why we introduce two new predicates $\texttt{Cause}(\cdot)$ and $\texttt{Update}(\cdot)$. Intuitively, predicates define how contextual cause is chosen from the memory Δ. Hence $\texttt{Cause}(\cdot)$ and $\texttt{Update}(\cdot)$ need to be implemented differently when different data structures are used. The precise definitions for predicates is given in Section 6 when we show how to map considered causal semantics into the framework.

Whenever action α, with $a \in \alpha$ is passing the restriction νa_Δ, the check if the contextual cause set needs to be modified is obligatory. For this to happen there are two possibilities: if name $a \in sub j(\alpha)$ and $\texttt{empty}(\Delta) = false$, then rule Cause Ref is applied, otherwise if name $a \in obj(\alpha)$, rule is Open is used. The predicate $\texttt{Cause}(\Delta, K, K')$ of rule Cause Ref gives the definition of the new cause set K' and ensures that contextual cause set K will be substituted with K'. To update the cause in the resulting processes of rules Cause Ref and Open, we define the *contextual cause update* operation on the process X, written as $X_{[K'/K]@i}$. It updates the contextual cause K of the action identified by i with the new cause K'. Formal definition can be found in [29]. Restricted name can be sent out to the environment (extruded) by applying the rule Open which adds the key i into the memory Δ to record that it was the key of the extrusion. The predicate $\texttt{Update}(\Delta, K, K')$ defines a new cause set K'. The communication between two processes, when the action $\overline{b}\langle \nu a_\Delta \rangle$ is involved, is done with the rule Close. The operator $\#i$ is defined on

$$(\textsc{Out1}^\bullet)\ \overline{b}^j a^{j_1}[i,K].\mathbf{P} \overset{(i,K,j):\overline{b}a}{\rightsquigarrow} \overline{b}^j a^{j_1}.\mathbf{P} \qquad (\textsc{In1}^\bullet)\ b^j(x)[i,K].\mathbf{P} \overset{(i,K,j):b(x)}{\rightsquigarrow} b^j(x).\mathbf{P}$$

$$(\textsc{Ind}^\bullet)\ \frac{X' \overset{(i,K,j):\alpha}{\rightsquigarrow} X \ \alpha = \overline{b}a \vee \alpha = b(x) \wedge \mathtt{fresh}(i, \mathtt{H}[X])}{\mathtt{H}[X'] \overset{(i,K,j):\alpha}{\rightsquigarrow} \mathtt{H}[X]} \qquad (\textsc{Par}^\bullet)\ \frac{X' \overset{(i,K,j):\alpha}{\rightsquigarrow} X \quad i \notin Y}{X' \mid Y \overset{(i,K,j):\alpha}{\rightsquigarrow} X \mid Y}$$

$$(\textsc{Res}^\bullet)\ \frac{X' \overset{(i,K,j):\alpha}{\rightsquigarrow} Xa \notin \alpha}{\nu a_\Delta(X') \overset{(i,K,j):\alpha}{\rightsquigarrow} \nu a_\Delta(X)} \qquad (\textsc{Com}^\bullet)\ \frac{X' \overset{(i,K,j):\overline{b}a}{\rightsquigarrow} X\ Y' \overset{(i,K',j'):b(x)}{\rightsquigarrow} Y\ K =_* j' \ \wedge \ K' =_* j}{X' \mid Y' \overset{(i,*,*):\tau}{\rightsquigarrow} X \mid Y\{^x/_{a^i}\}}$$

$$(\textsc{Open}^\bullet)\ \frac{X' \overset{(i,K,j):\alpha}{\rightsquigarrow} X \quad \alpha = \overline{b}a \vee \alpha = \overline{b}\langle \nu a_{\Delta'} \rangle \quad \mathtt{Update}(\Delta, K, K')}{\nu a_{\Delta+i}(X') \overset{(i,K',j):\overline{b}\langle \nu a_\Delta \rangle}{\rightsquigarrow} \nu a_\Delta(X)}$$

$$(\textsc{Cause Ref}^\bullet)\ \frac{X' \overset{(i,K,j):\alpha}{\rightsquigarrow} X \quad a \in sub(\alpha) \qquad \mathtt{empty}(\Delta) \neq true \quad \mathtt{Cause}(\Delta, K, K')}{\nu a_\Delta(X') \overset{(i,K',j):\alpha}{\rightsquigarrow} \nu a_\Delta(X)}$$

$$(\textsc{Close}^\bullet)\ \frac{X' \overset{(i,K,j):\overline{b}\langle \nu a_\Delta \rangle}{\rightsquigarrow} X \quad Y' \overset{(i,K',j'):b(x)}{\rightsquigarrow} Y \quad K =_* j' \ \wedge \ K' =_* j}{\nu a_\Delta(X' \mid Y') \overset{(i,*,*):\tau}{\rightsquigarrow} X \mid Y\{^x/_{a^i}\}}$$

Figure 8: Backward rules.

each data structure (Section 5.1) to delete from the data structure Δ, the keys of the actions that extruded restricted name but are part of synchronisations. It is necessary since in the semantics of [25, 28], τ-actions do not impose contextual causality. Rule Res, we define in the usual way. The examples how parametric rules work are given in Section 6.

Backward rules are symmetric to the forward ones and we present them in Figure 8. In order to have a better understanding of the backward rules, we shall give the following example.

Example 6. We consider the resulting process from Example 5 end its backward executions. First we need to reverse the communication on the channel d:

$$\overline{b}^* a^*[i,*].\overline{d}^* e^*[i',*] \mid d^*(x)[i',*].\overline{e}^{i'} c^* \overset{(i',*,*):\tau}{\rightsquigarrow} \overline{b}^* a^*[i,*].\overline{d}^* e^* \mid d^*(x).\overline{x}^* c^*$$

We can notice that in the resulting process, prefixes $\overline{d}^* e^*$ and $d^*(x)$ are not annotated anymore (the annotation $[i',*]$ is discarded during the backward step) and the substitution is reversed, hence name $e^{i'}$ is substituted with variable x. Now we can reverse the action $\overline{b}a$:

$$\overline{b}^* a^*[i,*].\overline{d}^* e^* \mid d^*(x).\overline{x}^* c^* \overset{(i,*,*):\overline{b}a}{\rightsquigarrow} \overline{b}^* a^*.\overline{d}^* e^* \mid d^*(x).\overline{x}^* c^*$$

and the obtained process is exactly process X from Example 5.

6 Mapping causal semantics

In this Section, we give the definitions for predicates in Figure 7 and use them to capture three different causal semantics [22, 25, 28] with our framework.

Reversible semantics for the π-calculus In Section 4 we gave a hint about the compositional semantics for the reversible π-calculus introduced in [22]. To be able to capture the behaviour of this semantics we shall use the set Γ as a data structure Δ. Any of the keys contained in Γ can be a cause for some action using the extruded name in the subject position. For example, in the process $\nu a_{\{i_1,i_2\}}(Y \mid a(x))$, the contextual cause of the action $a(x)$ can be i_1 or i_2.

To define the causality in their semantics, authors of [22] use the notion of the instantiation relation defined on the process memory. Here we adapt their definition to the framework and define it on the past prefixes. For instance in the process $b^*(x)[i_1, K_1].\overline{a}^{i_1}c^*[i_2, K_2].Y$ actions i_1 and i_2 are in instantiation relation, since action i_1 instantiated the name a. We can notice it in past prefix $\overline{a}^{i_1}c^*[i_2, K_2]$ where name a is decorated with the key i_1. This means that during the action (which was a communication) identified by i_1, name a substituted the variable x.

Definition 12 (Instantiation relation on the framework)**.** *Two keys i_1 and i_2 such that $i_1, i_2 \in \mathtt{key}(X)$ and $X = C[b^{j_1}(x)[i_1, K_1].Y]$ with $Y = C'[\pi[i_2, K_2].Z]$ where $j_2 \in subj(\pi)$, are in* instantiation relation*, denoted with $i_1 \rightsquigarrow_X i_2$, if $j_2 = i_1$. If $i_1 \rightsquigarrow_X i_2$ holds, we will write $K_1 \rightsquigarrow_X K_2$.*

We would like to remark that actions in [22] can be caused only through the subject of the label, hence contextual cause set K is a singleton. Now we can give the final definition necessary to obtain the $R\pi$ causality in the framework.

Definition 13 ($R\pi$ causality)**.** *To capture $R\pi$ causality, data structure is instantiated with the set Γ and the predicates from Figure 7 are defined as:*

1. $\mathtt{Cause}(\Gamma, K, K')$ *stands for $K' = K$ or $\exists K' \in \Gamma\ K \rightsquigarrow_X K'$;*

2. $\mathtt{Update}(\Gamma, K, K')$ *stands for $K' = K$.*

In the rule Open, predicate $\mathtt{Update}(\Gamma, K, K')$ is just saying that there is no difference between causes K and K'. The predicate $\mathtt{Cause}(\Gamma, K, K')$, is used in the rule Cause Ref and it defines how the new contextual cause K' is chosen. To illustrate how the $R\pi$ causality is captured by the framework, we give the following example. More examples can be found in [29].

Example 7. Let us consider the process $X = \nu a_\emptyset(\overline{b}^* a^* \mid \overline{c}^* a^* \mid a^*(x))$ where we a have parallel extrusion of the same name a. By extruding the name a with the rule Open twice, on the channels b and c, we obtain a process:

$$\nu a_{\{i_1,i_2\}}(\overline{b}^* a^*[i_1, *] \mid \overline{c}^* a^*[i_2, *] \mid a^*(x))$$

We can notice that keys i_1 and i_2 are added to the set Γ. The rule Cause Ref is used for the execution of the action $a(x)$ which can choose its cause from the set $\{i_1, i_2\}$. By choosing, for example, i_2 as a cause, we obtain the process:

$$\nu a_{\{i_1,i_2\}}(\overline{b}^* a[i_1, *] \mid \overline{c}^* a[i_2, *] \mid a^*(x)[i_3, i_2])$$

In the memory $[i_3, i_2]$ we can see that the action identified with the key i_3 needs to be reversed before the action with the key i_2. Process $\overline{b}^* a[i_1, *]$ can execute a backward step at any time with the rule Open$^\bullet$.

Boreale and Sangiorgi causal semantics We give the intuition about a compositional causal semantics for forward π-calculus [25] in Section 4. In [29, 33], we revised causal semantics of [25], where we did not take into account a replication operator and we define it with late (rather than early, as it was originally given) semantics. To capture the behaviour of this semantics we shall use indexed set Γ_w as the data structure Δ. The cause of the action using the extruded name will be exactly w. For instance in the process $\nu a_{\{i_1,i_2\}_{i_1}}(Y \mid a(x))$ the contextual cause of the action $a(x)$ is i_1.

To capture the notion of causality obtained from the revised semantics of [25], we give definitions for the predicates in Figure 7.

Definition 14 (Boreale and Sangiorgi causal semantics)**.** *To capture Borale and Sangiorgi causality, the data structure Δ is instantiated with indexed set Γ_w and the predicates from Figure 7 are defined as:*

1. $\texttt{Cause}(\Gamma_w, K, K')$ *stands for* $K' = K \cup \{w\}$

2. $\texttt{Update}(\Gamma_w, K, K')$ *stands for* $K' = K \cup \{w\}$

The predicates $\texttt{Cause}(\cdot)$ and $\texttt{Update}(\cdot)$ define the new cause set K' by adding w to the old set of causes K. The reason for this is that actions could be caused through the subject and object position of the label (detailed explanation given in [29]).

We would like to remark that the silent actions do not exhibit or impose contextual causes, despite our original intent [36] to modify the semantics of [25].

The following example illustrates how framework capture causality of [25].

Example 8. Consider the process $X = \nu a_{\emptyset_*}(\overline{b}^* a^* \mid \overline{c}^* a^* \mid a^*(x))$. By extruding the name a over the channel b (rule Open is applied), we obtain the process:

$$\nu a_{\{i_1\}_{i_1}}(\overline{b}^* a^*[i_1, *] \mid \overline{c}^* a^* \mid a^*(x))$$

We can notice that $w = i_1$. By executing the actions $\overline{c}a$ with the rule Open and $a(x)$ with the rule Cause Ref, predicates on the rules $\texttt{Update}(\cdot)$ and $\texttt{Cause}(\cdot)$ ensure that $i_1 = w$ will be added to cause sets. We obtain the process:

$$\nu a_{\{i_1,i_2\}_{i_1}}(\overline{b}^* a^*[i_1, *] \mid \overline{c}^* a^*[i_2, \{i_1, *\}] \mid a^*(x)[i_3, \{i_1, *\}])$$

In the memories $[i_2, \{i_1, *\}]$ and $[i_3, \{i_1, *\}]$ we can notice that the executed actions are caused by the action i_1 (the first action) and for this reason the action i_1 needs to be reversed as the last one. The actions i_2 and i_3 can be reversed in any order.

After the illustration, we prove a causal correspondence between Boreale and Sangiorgi's late semantics and our framework when the data structure Δ is instantiated with the indexed set Γ_w. For the lack of space, we give just an idea of the proof, for more details look [33, 29]. First, we prove the structural correspondence between two calculi by showing that the trace started from the same π-calculus process in one calculus, can be mimicked in the other one while resulting processes translated into π-calculus are the same. After that, we prove the following theorem, where we include the object causality.

Theorem 8 (Causal correspondence)**.** *The reflexive and transitive closure of causality introduced in [25] coincides with the causality of the framework when* $\Delta = \Gamma_w$.

Crafa, Varacca and Yoshida causal semantics Event structure semantics for forward π-calculus [28] has been briefly revised in Section 4. The authors definition of the disjunctive object causality has as a consequence the feature where events do not have a unique causal history. In [30] it is argued that this type of causality cannot be expressed when are considered processes with contexts.

In our framework, we consider that reversibility is causally-consistent, hence the detection of the causes is fundamental. Otherwise by executing backward actions, one could reach the state that is not consistent[9]. For this reason, we consider some possibilities for keeping track of the causes. One of the options would be to choose one of the possible extruders as a cause. In that case, we would obtain a notion of causality that is similar to causality given in [22]. Another option, that we decided to follow, is to record all extruders of the certain name that happened before the action with a restricted name in the subject position. Since we do not know the exact extruder that caused the action with the bound subject, we record the whole set of them happened in the past. To obtain this behaviour we use set indexed with a set Γ_Ω as the data structure Δ. The cause of the action using an extruded name is the set Ω. For example, in the process $va_{\{i_1,i_2,i_3\}_{\{i_1,i_3\}}}(Y \mid a(x))$ the contextual cause of the action $a(x)$ is $\Omega = \{i_1, i_3\}$.

Now we give the final definition necessary to obtain a revised causality notion of [28] (where we keep track of the extruders) in the framework.

Definition 15 (Revised Crafa, Varacca and Yoshida causal semantics)**.** *If an indexed set* Γ_Ω *is chosen as a data structure for a memory* Δ*, the predicates in Figure 7 are defined as:*

[9]Not consistent state is the state where the action that extruded restricted name is reversed, while action using that name in the subject position is not.

1. $\texttt{Cause}(\Gamma_\Omega, K, K')$ *stands for* $K' = K \cup \Omega$

2. $\texttt{Update}(\Gamma_\Omega, K, K')$ *stands for* $K' = K$

In the predicate $\texttt{Cause}(\Gamma_\Omega, K, K')$, the new cause K' gathers all extrusions, which are not part of the synchronisations, executed previously.

We illustrate the mechanism given above with the following example.

Example 9. Let us consider the process $X = \nu a_{\emptyset_{\{*\}}}(\overline{b}^* a^* \mid \overline{c}^* a^* \mid a^*(x))$. By extruding the name a on the channels b and c (by applying rule Open), we obtain the process:

$$\nu a_{\{i_1,i_2\}_{\{*,i_1,i_2\}}}(\overline{b}^* a^*[i_1, *] \mid \overline{c}^* a^*[i_2, *] \mid a^*(x))$$

As we can notice, keys i_1 and i_2 are added in the data structure Γ_Ω in both sets. The cause set of the action $a(x)$ will be the whole set $\{*, i_1, i_2\}$. By executing the input action we obtain process:

$$\nu a_{\{i_1,i_2\}_{\{*,i_1,i_2\}}}(\overline{b}^* a[i_1, *] \mid \overline{c}^* a[i_2, *] \mid a^*(x)[i_3, \{*, i_1, i_2\}])$$

From the reversible point of view, action $a(x)$ needs to be reversed as the first one (we can notice it in the memory $[i_3, \{*, i_1, i_2\}]$). The other two actions can be reversed in any order.

7 Properties of the framework

In this Section, we show that the framework satisfies typical properties for reversible calculi [1, 2, 22, 35]. We start by proving that the framework is a conservative extension of standard π-calculus. After that, we show that reversibility in our framework, when instantiated with three considered data structures, is causally-consistent. All proofs, together with more examples and detailed explanations can be found in [33, 29].

Correspondence with the π-calculus To show the correspondence between framework and π-calculus, we first define an erasing function φ which generates a π-calculus process from a reversible process X, by deleting all the past information.

Definition 16 (Erasing function)**.** *The function $\varphi : \mathcal{X} \to \mathcal{P}$ that maps reversible processes to the π-calculus, is inductively defined as follows:*

$$\varphi(X \mid Y) = \varphi(X) \mid \varphi(Y) \qquad \varphi(\overline{b}^j a^{j'}.\mathbf{P}) = \overline{b}a.\varphi(\mathbf{P})$$

$$\varphi(\nu a_\Delta(X)) = \varphi(X) \quad \textit{if } \texttt{empty}(\Delta) = \mathit{false} \qquad \varphi(b^j(x).\mathbf{P}) = b(x).\varphi(\mathbf{P})$$

$$\varphi(\nu a_\Delta(X)) = \nu a\, \varphi(X) \quad \textit{if } \texttt{empty}(\Delta) = \mathit{true} \qquad \varphi(\mathbf{0}) = \mathbf{0}$$

$$\varphi(\texttt{H}[X]) = \varphi(X)$$

The erasing function can be extended to labels as:

$$\varphi((i, K, j) : \alpha) = \varphi(\alpha) \qquad \varphi(\overline{b}a) = \overline{b}a$$
$$\varphi(\overline{b}\langle va_{\Delta}\rangle) = \overline{b}\langle va\rangle \quad \textit{when } \texttt{empty}(\Delta) = \textit{true} \qquad \varphi(b(x)) = b(x)$$
$$\varphi(\overline{b}\langle va_{\Delta}\rangle) = \overline{b}a \quad \textit{when } \texttt{empty}(\Delta) = \textit{false} \qquad \varphi(\tau) = \tau$$

Now we shall show that there is a *forward* operational correspondence between a reversible process X and $\varphi(X)$. We prove the statement saying that every forward move of a reversible process X can be mimicked by π-calculus and opposite. Using that we show that the relation between a process X and its corresponding π term P is a strong bisimulation [19, Definition 2.2.1].

Lemma 1. *The relation given by* $(X, \varphi(X))$*, for all reachable processes X, is a strong bisimulation.*

The main properties of the framework In what follows we show that our framework enjoys fundamental properties for reversible calculi: Loop Lemma, Square Lemma and Causal-consistency Theorem. Most of the proof outlines and terminology are adapted from [1, 22] with more complex discussions due to the generality of the framework. For the lack of space, more details and the proofs can be found in [29]. We initiate by showing the property declaring that every transition (reduction step) can be undone.

Lemma 2 (Loop Lemma)**.** *For every reachable process X and forward transition* $t : X \xrightarrow{\mu} Y$ *there exists a backward transition* $t' : Y \overset{\mu}{\rightsquigarrow} X$*, and conversely.*

By exploiting the symmetry of the rules, we define a reverse transition of a transition $t : X \xrightarrow{\mu} Y$, written $t^{\bullet}$, as a transition with the same label and the opposite direction $t^{\bullet} : Y \overset{\mu}{\rightsquigarrow} X$, and vice versa $((t^{\bullet})^{\bullet} = t)$.

In what follows, we define the causality relation on our framework considering the three data structures that we are using. As the reversibility in our framework is causally consistent, it is the fundamental part of the framework. First, we specify the structural dependencies between two actions, defined on the past prefixes. For example, in a reversible process $X = \overline{b}a[i, K].\overline{c}d[i', K']$ the past prefix with key i' structurally depends on the past prefix with key i. Formally, we have:

Definition 17 (Structural cause)**.** *For every two keys* i_1 *and* i_2 *such that* $i_1, i_2 \in \texttt{key}(X)$*, we say that the past prefix with the key* i_1 *is a* structural cause *of the past prefix with the key* i_2 *in the process X, written as* $i_1 \sqsubset_X i_2$ *if* $X = C[\pi[i_1, K_1].Y]$ *and* $i_2 \in \texttt{key}(Y)$*.*

We extend this definition on the transitions.

Definition 18 (Structural causality)**.** *Transition* $t_1 : X \xrightarrow{(i_1,K_1,j_1):\alpha_1} X'$ *is a* structural cause *of transition* $t_2 : X' \xrightarrow{(i_2,K_2,j_2):\alpha_2} X''$*, written* $t_1 \sqsubset t_2$*, if* $i_1 \sqsubset_{X''} i_2$*, or* $i_2 \sqsubset_X i_1$ *if the transitions are backward.* Structural causality*, denoted with* $\sqsubseteq$*, is obtained as the reflexive and transitive closure of* $\sqsubset$*.*

Following [22], we use the contextual cause set K (in [22] the K is a key k, not a set) to keep track of the object causality (causality imposed by extrusions). The object causality has been illustrated through the examples given in Section 6, when we show how to map different causal semantics into the framework. More examples can be found in [29]. Here we give the formal definition.

Definition 19 (Object causality)**.** *Transition* $t_1 : X \xrightarrow{(i_1,K_1,j_1):\alpha_1} X'$ *is an* object cause *of transition* $t_2 : X' \xrightarrow{(i_2,K_2,j_2):\alpha_2} X''$*, written* $t_1 < t_2$*, if* $i_1 \in K_2$ *or* $i_2 \in K_1$ *(for the backward transition) and* $t_1 \neq t_2^\bullet$. Object causality*, denoted with* $\ll$*, is obtained as the reflexive and transitive closure of* $<$*.*

Once having defined object and subject causality we can state the causality relation on the framework.

Definition 20 (Causality relation and concurrency)**.** *The* causality relation $\prec$ *is the reflexive and transitive closure of structural and object causality:* $\prec = (\sqsubseteq \cup \ll)^*$*. Two transitions are* concurrent *if they are not causally related.*

By having established the notion of concurrent transitions, we prove that they can be permuted, where commutation of transitions is preserved up to the label equivalence[10] defined as the least equivalence relation satisfying: $(i, K, j) : \overline{b}\langle \nu a_\Delta \rangle =_\lambda (i, K, j) : \overline{b}\langle \nu a_{\Delta'} \rangle$ for all i, j, K, a, b and $\Delta, \Delta' \subseteq \mathcal{K}$.

Lemma 3 (Square Lemma)**.** *If* $t_1 : X \xrightarrow{\mu_1} Y$ *and* $t_2 : Y \xrightarrow{\mu_2} Z$ *are two concurrent transitions, there exist* $t_2' : X \xrightarrow{\mu_2'} Y_1$ *and* $t_1' : Y_1 \xrightarrow{\mu_1'} Z$ *where* $\mu_i =_\lambda \mu_i'$*.*

Following the standard notation we write $t_2 = t_2'/t_1$ to state that t_2 is a residual of t_2' after t_1. *Coinitial* transitions have the same source, *cofinal* the same target, and they are *composable* if the target of one is the source of the other transition. A *trace* $t_1; t_2$ is a sequence of pairwise composable transitions and ϵ is the *empty* trace.

With the next theorem, we show the correctness of our framework declaring that the reversibility induced by it is causally-consistent. To be able to state the theorem, we first give the notion of the equivalence that we use. It is introduced in [1] as an adaptation of equivalence between traces introduced in [37, 38] which additionally deletes from the trace the transitions executed in both directions.

[10]Label equivalence is necessary since actions bring information about Δ into the labels. By permuting the transitions, the content of Δ is changing. More information can be found in [22].

Definition 21 (Equivalence up-to permutation)**.** *Equivalence up-to permutation,* $\sim$*, is the least equivalence relation on the traces, satisfying:*

$$t_1;(t_2/t_1) \sim t_2;(t_1/t_2) \qquad t;t^\bullet \sim \epsilon$$

Theorem 9 (Causal-consistency)**.** *Two traces are coinitial and cofinal if and only if they are equivalent up-to permutation.*

8 Conclusion

In this work, we studied the relations between different models (represented via process algebra) for reversible concurrent computations and provided a common framework in which different causal models can be compared.

Concerning CCS, we showed the isomorphism between LTSs of RCCS [1] and CCSK [2] which relies on two encodings. An explanation of this result is the existence of one causality notion in CCS. As the future work, we are interested in the further analysis of the relations between the reversible calculi, with the aim to define the classes of calculi to which the CCSK approach can be applied. The other direction is to exploit our result about the isomorphism and to integrate the irreversible actions into CCSK. Having in mind work on transactions in RCCS [12], we believe that CCSK with irreversible actions can be used to describe transactions.

In π-calculus, we developed the parametric framework for reversible π-calculi, able to express different notions of causality. Three different data structures used in the framework lead to the causal semantics given in [22, 25, 28], where we revised the semantics of [28] to be able to keep track about the causes. Additionally, the framework adds reversibility to the semantics where it was not defined originally. We show that reversibility in the framework, when considered data structures are used, is causally-consistent and prove the causal correspondence with the semantics of [25]. Actually, we are working on proving the causal correspondence between $R\pi$ causality [22] and the matching instance of the framework. As the future work, we would like to explore the causal bisimulation that can be defined on the framework, starting from already existing ones given in [39, 40, 25].

Acknowledgements I would like to express my sincere gratitude to all my co-authors mentioned in the introduction, from whom I learned a lot. Currently, I am a postdoc at the Focus Team, Inria, Sophia Antipolis - Méditerranée and supported by the French ANR project DCore ANR-18-CE25-0007.

References

[1] V. Danos, J. Krivine, Reversible communicating systems, in: CONCUR 2004, Vol. 3170 of LNCS, Springer, 2004, pp. 292–307.

[2] I. Phillips, I. Ulidowski, Reversing algebraic process calculi, JLAP. 73 (1-2).

[3] R. Landauer, Irreversibility and heat generation in the computing process, IBM J. Res. Dev. 5 (1961) 183–191.

[4] L. Cardelli, C. Laneve, Reversible structures, in: Computational Methods in Systems Biology, CMSB 2011, Paris, France, September, 2011., 2011, pp. 131–140.

[5] V. Danos, J. Krivine, Formal molecular biology done in CCS-R, Electr. Notes Theor. Comput. Sci. 180 (3) (2007) 31–49.

[6] I. Phillips, I. Ulidowski, S. Yuen, A reversible process calculus and the modelling of the ERK signalling pathway, in: RC, Copenhagen, Denmark, 2012, pp. 218–232.

[7] S. Kuhn, I. Ulidowski, Local reversibility in a calculus of covalent bonding, Sci. Comput. Program. 151 (2018) 18–47.

[8] B. Boothe, Efficient algorithms for bidirectional debugging, in: Conference on Programming Language Design and Implementation, PLDI '00, 2000, pp. 299–310.

[9] E. Giachino, I. Lanese, C. A. Mezzina, Causal-consistent reversible debugging, in: Fundamental Approaches to Software Engineering, FASE, 2014, pp. 370–384.

[10] I. Lanese, A. Palacios, G. Vidal, Causal-consistent replay debugging for message passing programs, in: FORTE 2019, Denmark, June 17-21, 2019, pp. 167–184.

[11] J. Grattage, A functional quantum programming language, in: LICS, IEEE Computer Society, Washington, DC, USA, 2005, pp. 249–258.

[12] V. Danos, J. Krivine, Transactions in RCCS, in: CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, 2005, pp. 398–412.

[13] I. Lanese, M. Lienhardt, C. Mezzina, A. Schmitt, J.-B. Stefani, Concurrent flexible reversibility, in: ESOP 2013, 2013, pp. 370–390.

[14] A. Philippou, K. Psara, Reversible computation in petri nets, in: Reversible Computation, RC 2018, Leicester, UK, September 12-14, 2018, pp. 84–101.

[15] I. Phillips, I. Ulidowski, Reversibility and asymmetric conflict in event structures, in: CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013, pp. 303–318.

[16] J. Hoey, I. Ulidowski, S. Yuen, Reversing parallel programs with blocks and procedures, in: EXPRESS/SOS, Beijing, China, September 3, 2018, pp. 69–86.

[17] J. A. Bergstra, A. Ponse, S. A. Smolka, Handbook of process algebra, Elsevier, 2001.

[18] R. Milner, A Calculus of Communicating Systems, Vol. 92 of LNCS, Springer, 1980.

[19] D. Sangiorgi, D. Walker, The Pi-Calculus - a Theory of Mobile Processes, Cambridge Uni. Press, 2001.

[20] G. D. Plotkin, A structural approach to operational semantics, JLAP 60-61.

[21] J. Krivine, A verification technique for reversible process algebra, in: Reversible Computation, RC 2012, Copenhagen, Denmark, July 2-3, 2012, pp. 204–217.

[22] I. Cristescu, J. Krivine, D. Varacca, A compositional semantics for the reversible π-calculus, in: LICS 2013, 2013, pp. 388–397.

[23] I. Ulidowski, I. Phillips, S. Yuen, Reversing event structures, New Generation Comput. 36 (3) (2018) 281–306.

[24] E. Graversen, I. Phillips, N. Yoshida, Event structure semantics of (controlled) reversible CCS, in: RC, Leicester, UK, September 12-14, 2018, pp. 102–122.

[25] M. Boreale, D. Sangiorgi, A fully abstract semantics for causality in the π-calculus, Acta Inf. 35 (5) (1998) 353–400.

[26] P. Degano, C. Priami, Non-interleaving semantics for mobile processes, Theor. Comput. Sci. 216 (1-2) (1999) 237–270.

[27] N. Busi, R. Gorrieri, A Petri net semantics for pi-calculus, in: CONCUR Philadelphia, PA, USA, August 21-24, 1995, Proceedings, 1995, pp. 145–159.

[28] S. Crafa, D. Varacca, N. Yoshida, Event structure semantics of parallel extrusion in the pi-calculus, in: FOSSACS, Vol. 7213 of LNCS, Springer, 2012, pp. 225–239.

[29] D. Medic, Relative expressiveness of calculi for reversible concurrency, Thesis, IMT School for Advanced Studies, Lucca, Italy, March 2019.

[30] I. Cristescu, J. Krivine, D. Varacca, Rigid families for CCS and the π-calculus, in: ICTAC, Vol. 9399 of LNCS, Springer, 2015, pp. 223–240.

[31] I. Cristescu, J. Krivine, D. Varacca, Rigid families for the reversible π-calculus, in: Reversible Computation, RC, Bologna, Italy, July 7-8, 2016, pp. 3–19.

[32] D. Medic, C. Mezzina, Static VS dynamic reversibility in CCS, in: Reversible Computation RC 2016, Vol. 9720 of LNCS, Springer, 2016, pp. 36–51.

[33] D. Medic, C. A. Mezzina, I. Phillips, N. Yoshida, A parametric framework for reversible pi-calculi, in: EXPRESS/SOS, China, Vol. 276 of EPTCS, 2018.

[34] C. Palamidessi, Comparing the expressive power of the synchronous and asynchronous pi-calculi, MSCS 13 (5) (2003) 685–719.

[35] I. Lanese, C. Mezzina, J.-B. Stefani, Reversibility in the higher-order π-calculus, Theor. Comput. Sci. 625 (2016) 25–84.

[36] D. Medic, C. Mezzina, Towards parametric causal semantics in π-calculus, in: ICTCS/CILC, Naples, Italy, September 26-28., 2017, pp. 121–125.

[37] J. Lévy, An algebraic interpretation of the $\lambda\beta$k-calculus; and an application of a labelled λ -calculus, Theor. Comput. Sci. 2 (1) (1976) 97–114.

[38] G. Boudol, I. Castellani, Permutation of transitions: An event structure semantics for CCS and SCCS, in: LTBT and POLMC, Vol. 354 of LNCS, Springer, 1988, pp. 411–427.

[39] I. Phillips, I. Ulidowski, A hierarchy of reverse bisimulations on stable configuration structures, MSCS 22 (2) (2012) 333–372.

[40] C. Aubert, I. Cristescu, History-preserving bisimulations on reversible calculus of communicating systems, CoRR.