Logical Methods in Computer Science
Volume 20, Issue 4, 2024, pp. 20:1–20:37
https://lmcs.episciences.org/

Submitted Sep. 26, 2023
Published Dec. 03, 2024

A TRULY CONCURRENT SEMANTICS FOR REVERSIBLE CCS

HERNÁN MELGRATTI [a], CLAUDIO ANTARES MEZZINA [b], AND G. MICHELE PINNA [c]

[a] ICC - Universidad de Buenos Aires - Conicet, Argentina

[b] Dipartimento di Scienze Pure e Applicate, Università di Urbino, Italy

[c] Università di Cagliari, Italy

Abstract. Reversible CCS (RCCS) is a well-established, formal model for reversible communicating systems, which has been built on top of the classical Calculus of Communicating Systems (CCS). In its original formulation, each CCS process is equipped with a memory that records its performed actions, which is then used to reverse computations. More recently, abstract models for RCCS have been proposed in the literature, essentialy, by directly associating RCCS processes with (reversible versions of) event structures. In this paper we propose a different abstract model: starting from one of the well-known encoding of CCS into Petri nets we apply a recently proposed approach to incorporate causally-consistent reversibility to Petri nets, obtaining as result the (reversible) net counterpart of every RCCS term.

1. Introduction

The calculus for concurrent systems (CCS) [Mil80] serves as one of the foundational frameworks for concurrent systems. Typically, these systems are described as the parallel composition of *processes* (also referred to as components), which interact by sending and receiving messages through named channels. Processes are defined in terms of communication actions performed over specific channels. For example, we use a and $\overline{a}$ to respectively represent a receive and a send action over the channel a. Basic actions can be combined using prefixing ($\_.\_$), choice ($\_ + \_$), and parallel ($\_ \parallel \_$) operators. Initially, the semantics of CCS were based on the *interleaved* approach, which considers only executions that arise from a single processor. Consequently, parallelism was reduced to non-deterministic choices and prefixing. For instance, under the interleaved approach, the CCS processes $a \parallel b$ and $a.b \ + \ b.a$ are considered *equivalent.* This means that the framework does not distinguish between a process that can perform actions a and b concurrently and one that sequentially executes these

This work has been partially supported by the BehAPI project funded by the EU H2020 RISE under the Marie Sklodowska-Curie action (No: 778233), by the Italian PRIN 2020 project NiRvAna – Noninterference and Reversibility Analysis in Private Blockchains, the Italian Ministry of Education, University and Research through the PRIN 2022 project " Developing Kleene Logics and their Application" (DeKLA), project code: 2022SM4XC8, the INdAM-GNCS E53C22001930001 project RISICO – Reversibilità in Sistemi Concorrenti: Analisi Quantitative e Funzionali, and the European Union - NextGenerationEU SEcurity and RIghts in the CyberSpace (SERICS) Research and Innovation Program PE00000014, projects STRIDE and SWOP.

DOI:10.46298/LMCS-20(4:20)2024

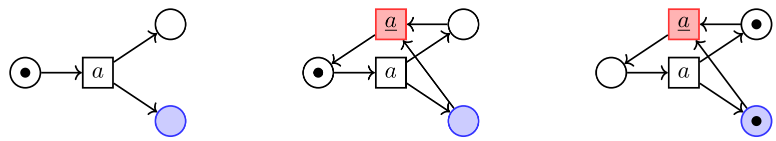


Figure 1. Encoding of $R = \langle\rangle \triangleright a.0$

actions in any possible order (interleaving/schedule). To address this limitation, subsequent research aimed to equip CCS with *true concurrent* semantics, adopting styles similar to Petri nets [Rei85] and Event Structures [Win88, NPW81]. It has been shown that every CCS process can be associated with a corresponding Petri net that can mimic its computations. Various flavors of Petri nets have been explored in the literature, including *occurrence* nets [Gol90], a variant of *Conditions/Events* nets [DNM88], and *flow* nets [BC94]. The works in [Win86] and [BC94] have additionally shown that the computation of a CCS process can be represented by event structures.

In the last decades, many efforts were made to endow computation models with reversible semantics [A+20, M+20]. In particular, two different models have been proposed for CCS: reversible CCS (RCCS) [DK04, Kri13] and CCS with communication keys (CCSK) [PU07]. Both of them incorporate a logging mechanism in the operational semantics of CCS that enables the undoing of computation steps. Moreover, it has been shown that they are isomorphic [LMM21] since they only differ on how they log information about past computations: while RCCS relies on some form of *memory/monitor*, CCSK uses *keys*. Previous approaches have also developed true concurrent semantics for reversible versions of CCS. For instance, it has been shown that CCSK can be associated with reversible bundle event structures [GPY18, GPY21]. Also configuration structures have been associated to RCCS [AC20]. Nonetheless, we still lack a Petri net model for reversible CCS processes. We may exploit some recent results that connect reversible occurrence nets with reversible event structures [MMU20, MMP+20, MMP21a, MMP24] to indirectly recover a Petri net model from the reversible bundle event structures defined in [GPY18]. However, we follow a different approach, which is more direct:

(1) We encode CCS processes into a mild generalization of occurrence nets, namely *unravel nets*, in the vein of Boudol and Castellani [BC94].
(2) We show that *unravel nets* can be made *causally-consistent* reversible by applying the approach in [MMP+20].
(3) We finally show that the reversible unravel nets derived by our encoding are an interpretation of RCCS terms.

An interesting aspect of the proposed encoding is that it highlights that all the information needed for reversing an RCCS process is already *encoded* in the structure of the net corresponding to the original CCS process, i.e., RCCS memories are represented by the structure of the net. Concretely, if an RCCS process R is a reachable process from a CCS process P with empty memory, then the encoding of R is retrieved from the encoding of P, what changes is the position of the markings. Consider the CCS process $P = a.0$ that executes a and then terminates. It can be encoded as the Petri net on the left in Figure 1 (the usage of the apparently redundant coloured places in the postset of a will be made clearer in Section 3).

The reversible version of P is $R = \langle\rangle \triangleright a.0$, where $\langle\rangle$ denotes an initially empty memory. According to RCCS semantics, R evolves to $R' = \langle *, a, 0\rangle \cdot \langle\rangle \triangleright 0$ by executing a. The memory

$\langle *, a, \mathsf{0}\rangle \cdot \langle\rangle$ in R' indicates that it can go back to the initial process R by undoing a. Note that the net corresponding to P (on the left) contains all the necessary information to reverse the action a; intuitively, the action a can be undone by firing it in the opposite direction (i.e., by consuming tokens from the postset and producing them in its preset), or equivalently, by executing a reversing transition $\underline{a}$ as depicted in the net shown in the middle of Figure 1. Furthermore, it is important to highlight that the net on the right of Figure 1 corresponds to the derivative R'. The coloured place carries the information stored in the memory of the derivative R'. Consequently, the encoding of a CCS term as a net already encompasses all the information required for its reversal, which stands in contrast to the additional memories needed in the case of RCCS. This observation provides a straightforward and nearly immediate true concurrent representation of RCCS processes, effectively capturing their reversible behaviour.

Organization of the paper. The paper is structured as follows: after establishing essential notation, we provide a brief overview of CCS and RCCS in Section 2. Next, in Section 3, we present a concise summary of Petri nets and introduce the concept of *unravel* nets, followed by their reversible counterpart. The encoding of CCS into unravel nets and the mapping of RCCS terms into reversible unravel nets, along with correspondence results, are described in Section 4. In the final section, we draw insightful conclusions and discuss potential avenues for future developments. Additionally, we present a practical implementation of the encoding and simulation of the execution in Haskell in Appendix A.

A preliminary version of this work has been published as [MMP21b]. In this version we have extended the scope and applicability of the proposed approach. We move from finite processes to infinite ones (i.e., recursive processes) by considering terms defined coinductively. Secondly, in this version we provide full and rigorous proofs of the key results. Finally, we provide a Haskell implementation of the encoding that allows for the simulation of the execution of encoded CCS processes. This practical implementation further exemplifies the feasibility and effectiveness of the approach.

Preliminaries. We recall some notation that we will use in the paper. We denote the set of natural numbers as $\mathbb{N}$. Let A be a set, a *multiset* of A is defined as a function $m : A \to \mathbb{N}$. The set of multisets of A is denoted by ∂A. We assume the usual operations on multisets, such as union $+$ and difference $-$. For multisets $m, m' \in \partial A$, we write $m \subseteq m'$ to indicate that $m(a) \leq m'(a)$ for all $a \in A$. Additionally, we define $[\![m]\!]$ as the multiset where $[\![m]\!](a) = 1$ if $m(a) > 0$, and $[\![m]\!](a) = 0$ otherwise. When a multiset m of A is a set, i.e., $m = [\![m]\!]$, we write $a \in m$ to denote that $m(a) \neq 0$. In this case, we often confuse the multiset m with the set $\{a \in A \mid m(a) \neq 0\}$ or a subset $X \subseteq A$ with the multiset $X(a) = 1$ if $a \in A$ and $X(a) = 0$ otherwise. We also employ standard set operations such as $\cap$, $\cup$, or $\setminus$, and, with a slight abuse of notation, write $\emptyset$ for the multiset m such that $[\![m]\!] = \emptyset$.

Given a relation $\mathcal{R}$, we indicate with $\mathcal{R}^*$ its reflexive and transitive closure.

2. CCS and reversible CCS

Let $\mathcal{A}$ be a set of actions, denoted as $a, b, c, \ldots$, and let $\overline{\mathcal{A}} = \{\overline{a} \mid a \in \mathcal{A}\}$ be the set of their corresponding co-actions. The set containing all possible actions is denoted by $\mathtt{Act} = \mathcal{A} \cup \overline{\mathcal{A}}$. We use α and β to represent elements from $\mathtt{Act}_\tau = \mathtt{Act} \cup \{\tau\}$, where τ is a symbol not present in Act, i.e., $\tau \notin$ Act, and denotes a *silent* action. We assume that for each $\alpha \in$ Act we have that $\bar{\bar{\alpha}} = \alpha$.

$$\text{(Actions)} \qquad \alpha \quad ::= a \mid \overline{a} \mid \tau$$

$$\text{(CCS Processes)} \qquad P, Q \quad ::=^{\mathtt{co}} \textstyle\sum_{i \in I} \alpha_i.P_i \mid (P \parallel Q) \mid P \backslash a$$

Figure 2. CCS Syntax

The syntax of CCS is presented in Figure 2. A prefix (or action) in CCS can take one of three forms: an input a, an output $\overline{a}$, or the silent action τ. A term of the form $\sum_{i \in I} \alpha_i.P_i$ represents a process that non-deterministically starts by selecting and performing some action α_i and then continues as P_i. We use $\mathbf{0}$, the idle process, when $I = \emptyset$ in place of $\sum_{i \in I} \alpha_i.P_i$. Similarly, we use $\alpha_i.P$ for a unitary sum where I is the singleton $\{i\}$. The term $P \parallel Q$ represents the parallel composition of processes P and Q. An action a can be restricted to be visible only inside process P, denoted as $P \backslash a$. Restriction is the only binder in CCS, where a is bound in $P \backslash a$. We addressed the representation of infinite processes by adopting an approach initiated by [CGP09]. Instead of fixing a syntactic representation of recursion, we simplified the treatment by employing infinite regular trees. Throughout this paper, in Figure 2 and beyond, we use the symbol $::=^{\mathtt{co}}$ to indicate that the productions should be interpreted *coinductively*. As a result, the set of processes is the greatest fixed point of the (monotonic) functor over sets defined by the grammar above [BDd22]. Consequently, a process is a potentially infinite, *regular* term coinductively generated by the grammar in Figure 2. A term is considered regular if it consists of finitely many *distinct* subterms. The language generated by the coinductive grammar is thus finitely representable either using the so-called μ notation [Pie02] or as solutions of finite sets of equations [Cou83]. For a more comprehensive treatment, interested readers are referred to [Cou83].

We represent the set of all CCS processes as $\mathcal{P}$. We denote the set of names of a process P as $\mathtt{n}(P)$, and we use $\mathtt{fn}(P)$ and $\mathtt{bn}(P)$ to represent the sets of free and bound names in P, respectively. (These functions can be straightforwardly defined by coinduction.)

Definition 2.1 (CCS Semantics)**.** The operational semantics of CCS is defined as the LTS $(\mathcal{P}, \mathtt{Act}_\tau, \rightarrow)$ where the transition relation $\rightarrow$ is the smallest relation induced by the rules in Figure 3.

Let us provide some comments on the rules presented in Figure 3. The act rule indicates that a non-deterministic choice proceeds by executing one of its prefixes α_z and transitions to the corresponding continuation P_z. The par-l and par-r rules allow the left and right processes of a parallel composition to independently execute an action while the other remains unchanged. The syn rule regulates synchronisation, allowing two processes in parallel to perform a handshake. Lastly, the hide rule restricts a certain action from being further propagated.

2.1. **Reversible CCS.** Reversible CCS (RCCS) [DK04, Kri13] is a reversible variant of CCS. In RCCS, processes are equipped with a *memory* that stores information about their past actions. The syntax of RCCS, shown in Figure 4, includes the same constructs as the original CCS formulation, but with the addition of reversible processes. A reversible process in RCCS can take one of the following forms: a *monitored* process $m \triangleright P$ where

$$\frac{z \in I}{\sum_{i\in I} \alpha_i.P_i \xrightarrow{\alpha_z} P_z} \text{(ACT)} \qquad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \text{(PAR-L)} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'} \text{(PAR-R)}$$

$$\frac{P \xrightarrow{\alpha} P' \qquad Q \xrightarrow{\bar{\alpha}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \text{(SYN)} \qquad \frac{P \xrightarrow{\alpha} P' \qquad \alpha \notin \{a, \bar{a}\}}{P\backslash a \xrightarrow{\alpha} P'\backslash a} \text{(R-RES)}$$

Figure 3. CCS semantics

(CCS Processes)	P, Q	$::=^{\mathtt{co}} \sum_{i\in I} \alpha_i.P_i \mid (P \parallel Q) \mid P\backslash a$
(RCCS Processes)	R, S	$::= m \triangleright P \mid (R \parallel S) \mid R\backslash a$
(Memories)	m	$::= \langle *, \alpha, Q\rangle \cdot m \mid \langle m, \alpha, Q\rangle \cdot m' \mid \langle 0\rangle \cdot m \mid \langle 1\rangle \cdot m \mid \langle\rangle$

Figure 4. RCCS syntax

m represents the memory, and P is a CCS process; the parallel composition $R \parallel S$ of the reversible processes R and S; and the restriction $R\backslash a$, where the action a is restricted to the process R. A *memory* is essentially a stack of events that encodes the history of actions previously performed by a process. The left-most element in the memory corresponds to the very last action executed by the monitored process. Memories in RCCS can contain three different kinds of events[1]: *partial* synchronisations $\langle *, \alpha, Q\rangle$, *full* synchronisations $\langle m, \alpha, Q\rangle$, and memory *splits* $\langle 0\rangle$ and $\langle 1\rangle$. In a synchronisation, whether partial or full, the action α and the process Q serve specific purposes in recording the selected action α of a choice and the discarded branches Q. The technical distinction between partial and full synchronisation will become evident when describing the semantics of RCCS. Events $\langle 0\rangle$ and $\langle 1\rangle$ represent the splitting of a monitored process into two parallel ones, respectively the left one ($\langle 0\rangle$) and the right one ($\langle 1\rangle$). The empty memory is represented by $\langle\rangle$. Let us note that in RCCS, memories also serve as unique process identifiers, and this will be handy when undoing a full synchronisation.

We define the following sets: the set $\mathcal{P}_R$ of all RCCS processes, the set $\mathcal{M}$ of all possible memories, and $\hat{\mathcal{M}} = \mathcal{M} \cup \mathcal{M}^2$, which includes individual as well as pairs of memories. We let $\hat{m}$ to range over the set $\hat{\mathcal{M}}$.

As for CCS, the only binder in RCCS is restriction, which applies at the level of both CCS and RCCS processes. Consequently, we extend the functions `n`, `fn`, and `bn` to RCCS processes and memories accordingly.

Definition 2.2. The operational semantics of RCCS is defined as a pair of LTSs sharing the same set of states and labels: a forward LTS $(\mathcal{P}_R, \hat{\mathcal{M}} \times \mathtt{Act}_\tau, \rightarrow)$ and a backward LTS $(\mathcal{P}_R, \hat{\mathcal{M}} \times \mathtt{Act}_\tau, \rightsquigarrow)$. Elements of the set $\hat{\mathcal{M}} \times \mathtt{Act}_\tau$ will be denoted as $m : \alpha$. The transition relations $\rightarrow$ and $\rightsquigarrow$ are the smallest relations induced by the rules in Figure 5 (left and right columns, respectively). Both relations make use of the structural congruence relation $\equiv$,

[1] In this paper, we adopt the original RCCS semantics with partial synchronisation. Later versions, such as [Kri13], employ communication keys to uniquely identify actions.

$$\text{(R-ACT)}\ \frac{}{m \triangleright \sum_{i\in I} \alpha_i.P_i \xrightarrow{m:\alpha_z} \langle *, \alpha_z^z, \sum_{i\in I\setminus\{z\}} \alpha_i.P_i\rangle \cdot m \triangleright P_z}$$

$$\frac{}{\langle *, \alpha_z^z, \sum_{i\in I\setminus\{z\}} \alpha_i.P_i\rangle \cdot m \triangleright P_z \overset{m:\alpha_z}{\rightsquigarrow} m \triangleright \sum_{i\in I} \alpha_i.P_i}\ \text{(R-ACT}^\bullet\text{)}$$

$$\text{(R-PAR-L)}\ \frac{R \xrightarrow{m:\alpha} R'}{R \parallel S \xrightarrow{m:\alpha} R' \parallel S} \qquad \frac{R \overset{m:\alpha}{\rightsquigarrow} R'}{R \parallel S \overset{m:\alpha}{\rightsquigarrow} R' \parallel S}\ \text{(R-PAR-L}^\bullet\text{)}$$

$$\text{(R-PAR-R)}\ \frac{S \xrightarrow{m:\alpha} S'}{R \parallel S \xrightarrow{m:\alpha} R \parallel S'} \qquad \frac{S \overset{m:\alpha}{\rightsquigarrow} S'}{R \parallel S \overset{m:\alpha}{\rightsquigarrow} R \parallel S'}\ \text{(R-PAR-R}^\bullet\text{)}$$

$$\text{(R-SYN)}\ \frac{R \xrightarrow{m_1:\alpha} R' \qquad S \xrightarrow{m_2:\bar{\alpha}} S'}{R \parallel S \xrightarrow{m_1,m_2:\tau} R'_{m_2@m_1} \parallel S'_{m_1@m_2}} \qquad \frac{R \overset{m_1:\alpha}{\rightsquigarrow} R' \qquad S \overset{m_2:\bar{\alpha}}{\rightsquigarrow} S'}{R_{m_2@m_1} \parallel S_{m_1@m_2} \overset{m_1,m_2:\tau}{\rightsquigarrow} R' \parallel S'}\ \text{(R-SYN}^\bullet\text{)}$$

$$\text{(R-RES)}\ \frac{R \xrightarrow{m:\alpha} R' \qquad \alpha \notin \{a, \bar{a}\}}{R\backslash a \xrightarrow{m:\alpha} R'\backslash a} \qquad \frac{R \overset{m:\alpha}{\rightsquigarrow} R' \qquad \alpha \notin \{a, \bar{a}\}}{R\backslash a \overset{m:\alpha}{\rightsquigarrow} R'\backslash a}\ \text{(R-RES}^\bullet\text{)}$$

$$\text{(R-EQUIV)}\ \frac{R \equiv R' \quad R' \xrightarrow{m:\alpha} S' \quad S' \equiv S}{R \xrightarrow{m:\alpha} S} \qquad \frac{R \equiv R' \quad R' \overset{m:\alpha}{\rightsquigarrow} S' \quad S' \equiv S}{R \overset{m:\alpha}{\rightsquigarrow} S}\ \text{(R-EQUIV}^\bullet\text{)}$$

FIGURE 5. RCCS semantics

which is the smallest congruence on RCCS processes containing the rules shown in Figure 6. We define $\hookrightarrow = \rightarrow \cup \rightsquigarrow$.

Let us provide some comments on the forward rules in Figure 5 (left column). Rule R-ACT allows a monitored process to perform a forward action α_z. Notably, the label of this transition pairs the executed action α_z with the memory m of the process. At this point, we are uncertain whether the performed action will synchronise with the context or not. Consequently, a partial synchronisation event of the form $\langle *, \alpha_z^z, \sum_{i\in I\setminus z} \alpha_i.P_i\rangle$ is added on top of the memory. The '$*$' in the partial synchronisation event will be replaced by a memory, let's say m_1, if the process eventually synchronises with another process monitored by m_1. Additionally, it is essential to note that the discarded process $\sum_{i\in I\setminus z} \alpha_i.P_i$ is recorded in the memory. Moreover, along with the prefix, we store its position 'z' within the sum. While this piece of information may be redundant for RCCS itself and was not present in the original semantics, it becomes useful when encoding an RCCS process into a net and when proving operational correspondence. This additional information enables a more straightforward representation of RCCS processes in a net-based setting and supports the validation of operational correspondence between the LTS and the net semantics. Importantly, it is worth

$$(\textsc{split}) \qquad m \triangleright (P \parallel Q) \equiv \langle 0 \rangle \cdot m \triangleright P \parallel \langle 1 \rangle \cdot m \triangleright Q$$

$$(\textsc{res}) \qquad m \triangleright P \backslash a \equiv (m \triangleright P) \backslash a \qquad \text{if } a \notin \texttt{fn}(m)$$

$$(\alpha) \qquad R \equiv S \qquad \text{if } R =_\alpha S$$

FIGURE 6. RCCS Structural laws

mentioning that this straightforward modification does not alter the original semantics of RCCS.

Rules R-PAR-L and R-PAR-R allows for the independent execution of an action in different components of a parallel composition. Rule R-SYN allows two parallel processes to synchronise. For synchronisation to occur, the action α in one process must match the co-action $\overline{\alpha}$ in the other process. Once this condition is met, the two partial synchronisations are updated to two full synchronisations using the operator '@'.

Definition 2.3. Let R be a monitored process, and let m_1 and m_2 be two memories. $R_{m_2@m_1}$ represents the process obtained from R by substituting all occurrences of $\langle *, \alpha, Q\rangle \cdot m_1$ with $\langle m_2, \alpha, Q\rangle \cdot m_1$.

Rule R-RES propagates actions through restriction, provided that the action is not on the restricted name. Rule R-EQUIV allows one to exploit the structural congruence defined in Figure 6. The structural rule SPLIT enables a monitored process with a top-level parallel composition to split into left and right branches, resulting in the duplication of the memory. The structural rule RES permits pushing restrictions outside monitored processes. Lastly, the structural rule α allows one to take advantage of α-conversion, denoted by $=_\alpha$.

Backward rules are reported in the right column of the Figure 5. As one can see, for each forward rule there exists a symmetrical backward one. Rule R-ACT$^\bullet$ allows a monitored process to undo its last action, which coincides with the event on top of the memory stack. As we can see, all the information is stored in the last performed event, hence the rule pops out the last event on the memory, and restores back the prefix corresponding to the event and the plus context. Rules R-PAR-L$^\bullet$ and R-PAR-R$^\bullet$ allow for the independent undoing of an action in different components of a parallel composition. Rule R-SYN$^\bullet$ allows for a de-synchronisation: that is, two parallel components which participated to a synchronisation, say, with labels α and $\overline{\alpha}$ can undo this synchronisation. Let us stress out that two processes, say R and S can undo a synchronisation along memories m_1 and m_2 only if they are in the form $R_{m_2@m_1}$ and $S_{m_1@m_2}$. Rules R-RES$^\bullet$ and R-EQUIV$^\bullet$ acts like their forward counterparts.

Definition 2.4. An RCCS process of the form $\langle\rangle \triangleright P$ is referred to as *initial*. Any process R derived from an initial process using the rules in Figure 5 is called *coherent*.

Example 2.5. Let $P = a.(b \parallel c) \parallel (\overline{a} \parallel d)$. Via an application of the SPLIT rule we obtain the following process

$$\langle\rangle \triangleright P \equiv \big(\langle 0\rangle \cdot \langle\rangle \triangleright a.(b \parallel c)\big) \parallel \big(\langle 1\rangle \cdot \langle\rangle \triangleright (\overline{a} \parallel d)\big) = R_1$$

and we can further apply SPLIT rule on the second monitored process of R_1 as follows:

$$R_1 \equiv \langle 0\rangle \cdot \langle\rangle \triangleright a.(b \parallel c) \parallel \langle 0\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright \overline{a} \parallel \langle 1\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright d = R_2$$

Now, in the process R_2 there are two monitored processes which can synchronise on a. That is

$$R_2 \xrightarrow{m_1,m_2:\tau} \langle m_1, a, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle\rangle \triangleright (b \parallel c) \parallel \big(\langle m_2, \overline{a}, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright \mathbf{0}\big) \parallel \big(\langle 1\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright d\big) = R_3$$

and by applying the Split rule on the left-most monitored process of R_3 we obtain

$$R_3 \equiv \big(\langle 0\rangle \cdot \langle m_1, a, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle\rangle \triangleright b\big) \parallel \big(\langle 1\rangle \cdot \langle m_1, a, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle\rangle \triangleright c\big) \parallel$$
$$\big(\langle m_2, \overline{a}, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright \mathbf{0}\big) \parallel \big(\langle 1\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright d\big)$$

where $m_1 = \langle 0\rangle \cdot \langle\rangle$ and $m_2 = \langle 1\rangle \cdot \langle\rangle$.

An important property of a fully reversible calculus is the so called Loop Lemma, stating that any action can be undone. Formally:

Lemma 2.6 (Loop Lemma [DK04]). *Let R be a coherent process. For any forward transition $R \xrightarrow{\hat{m}:\alpha} S$ there exists a backward transition $S \overset{\hat{m}:\alpha}{\rightsquigarrow} R$, and conversely.*

Corollary 2.7. *Let R be a coherent process. If $R \hookrightarrow^* R_1$ then $R_1 \hookrightarrow^* R$.*

RCCS is shown to be causal consistent, that is any step can be undone provided that its consequences are undone beforehand. A consequence of causal consistent reversibility, it that any process reached by mixing computations (e.g., forward and backward transitions) can be reached by only forward computations. That is:

Property 2.8. For any initial procces P, if $\langle\rangle \triangleright P \hookrightarrow^* R$ then $\langle\rangle \triangleright P \rightarrow^* R$.

The notion of context below will be useful in the following sections.

Definition 2.9. RCCS process context C and active contexts E are reversible processes with a hole "$\circ$", defined by the following grammar:

$$C \;::=^{\mathsf{co}} \circ \mid m \triangleright C \mid \alpha.C \mid \sum_{i\in I} P_i^C \mid C \mid (P \parallel C) \mid (C \parallel P) \mid C\backslash A$$
$$E \;::=^{\mathsf{co}} \circ \mid (R \parallel E) \mid (E \parallel R) \mid E\backslash A$$

where P^C can be either C or P.

3. Petri nets, Unravel Nets and Reversible Unravel Nets

3.1. **Petri nets.** We provide a brief overview of Petri nets, along with some related auxiliary notions.

Definition 3.1. A *Petri net* is a tuple $N = \langle S, T, F, \mathsf{m}\rangle$, where S is a set of *places*, T is a set of *transitions* (with $S \cap T = \emptyset$), $F \subseteq (S \times T) \cup (T \times S)$ is the *flow* relation, and $\mathsf{m} \in \partial S$ is the *initial marking*.

Petri nets are conventionally represented with transitions depicted as boxes, places as circles, and the flow relation indicated by directed arcs. The marking m, i.e., the state of the net, is depicted by drawing in the place s a number $\mathsf{m}(s)$ of '$\bullet$' symbols, also called *tokens*.

Given a net $N = \langle S, T, F, \mathsf{m}\rangle$ and $x \in S \cup T$, we define the following multisets: ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$. If x is a place then ${}^\bullet x$ and $x^\bullet$ are (multisets) of transitions; analogously, if $x \in T$ then ${}^\bullet x \in \partial S$ and $x^\bullet \in \partial S$. The sets ${}^\bullet x$ and $x^\bullet$ are respectively called the *pre* and *postset* of x. A transition $t \in T$ is enabled at a marking $\mathsf{m} \in \partial S$, denoted by $\mathsf{m}\,[t\rangle$, whenever ${}^\bullet t \subseteq \mathsf{m}$. A transition t enabled at a marking m can *fire* and its firing produces the marking $\mathsf{m}' = \mathsf{m} - {}^\bullet t + t^\bullet$. The firing of t at a marking m producing m' is denoted by $\mathsf{m}\,[t\rangle\,\mathsf{m}'$. We assume that each transition t of a net N is such that ${}^\bullet t \neq \emptyset$, meaning that no transition may fire *spontaneously*. Given a generic marking m (not necessarily the initial one), the *firing sequence* (shortened as fs) of $N = \langle S, T, F, \mathsf{m}\rangle$ starting at m_0 is defined as:

- m_0 is a firing sequence (of length 0), and
- if $\mathsf{m}_0\,[t_1\rangle\,\mathsf{m}_1 \cdots \mathsf{m}_{n-1}\,[t_n\rangle\,\mathsf{m}_n$ is a firing sequence and $\mathsf{m}_n\,[t\rangle\,\mathsf{m}'$, then also $\mathsf{m}_0\,[t_1\rangle\,\mathsf{m}_1 \cdots \mathsf{m}_{n-1}\,[t_n\rangle\,\mathsf{m}_n\,[t\rangle\,\mathsf{m}'$ is a firing sequence.

The set of firing sequences of a net $N = \langle S, T, F, \mathsf{m}\rangle$ starting at a marking m is denoted by $\mathcal{R}^N_{\mathsf{m}}$ and it is ranged over by σ. Given a fs $\sigma = \mathsf{m}_0\,[t_1\rangle\,\sigma'\,[t_n\rangle\,\mathsf{m}_n$, $start(\sigma)$ is the marking m_0, $lead(\sigma)$ is the marking m_n and $tail(\sigma)$ is the fs $\sigma'\,[t_n\rangle\,\mathsf{m}_n$. Given a net $N = \langle S, T, F, \mathsf{m}\rangle$, a marking m' is *reachable* iff there exists a fs $\sigma \in \mathcal{R}^N_{\mathsf{m}}$ such that $lead(\sigma)$ is m'. The set of reachable markings of N is $\mathcal{M}_N = \{lead(\sigma) \mid \sigma \in \mathcal{R}^N_{\mathsf{m}}\}$. Given a fs $\sigma = \mathsf{m}\,[t_1\rangle\,\mathsf{m}_1 \cdots \mathsf{m}_{n-1}\,[t_n\rangle\,\mathsf{m}'$, we write $X_\sigma = \sum_{i=1}^{n} t_i$ for the multiset of transitions associated to fs, which we call an *execution* of the net and we write $\mathbb{E}(N) = \{X_\sigma \in \partial T \mid \sigma \in \mathcal{R}^N_{\mathsf{m}}\}$ for the set of the executions of N. Observe that an execution simply says which transitions (and the relative number of occurrences of them) has been executed, not their (partial) ordering. Given a fs $\sigma = \mathsf{m}\,[t_1\rangle\,\mathsf{m}_1 \cdots \mathsf{m}_{n-1}\,[t_n\rangle\,\mathsf{m}_n \cdots$, with ρ_σ we denote the sequence $t_1 t_2 \cdots t_n \cdots$.

Definition 3.2. A net $N = \langle S, T, F, \mathsf{m}\rangle$ is said to be *safe* if each marking $\mathsf{m} \in \mathcal{M}_N$ is such that $\mathsf{m} = [\![\mathsf{m}]\!]$.

The notion of subnet will be handy in the following. A subnet is obtained by restricting places and transitions, and correspondingly the flow relation and the initial marking.

Definition 3.3. Let $N = \langle S, T, F, \mathsf{m}\rangle$ be a Petri net and let $T' \subseteq T$ be a subset of transitions and $S' = {}^\bullet T' \cup T'^\bullet$. Then, the subnet generated by T' $N|_{T'} = \langle S', T', F', \mathsf{m}'\rangle$, where F' is the restriction of F to S' and T', and m' is the multiset on S' obtained by m restricting to the places in S'.

3.2. **Unravel nets.** To define *unravel nets* we need the notion of *causal net*, i.e., a net representing how the various transitions are related and all of them can be executed in a firing sequence.

Definition 3.4. A safe Petri net $N = \langle S, T, F, \mathsf{m}\rangle$ is a *causal net* (CA for short) when $\forall s \in S.\ |{}^\bullet s| \leq 1$ and $|s^\bullet| \leq 1$, F^* is acyclic, $T \in \mathbb{E}(N)$, and $\forall s \in S\ {}^\bullet s = \emptyset \Rightarrow \mathsf{m}(s) = 1$.

Requiring that $T \in \mathbb{E}(N)$ implies that all the transition can be executed whereas F^* acyclic means that dependencies among transitions are settled. Observe that causal net has no isolated and unmarked places as $\forall s \in S\ {}^\bullet s = \emptyset \Rightarrow \mathsf{m}(s) = 1$.

Definition 3.5. An *unravel net* (UN for short) $N = \langle S, T, F, \mathsf{m}\rangle$ is a safe net such that

(1) for each execution $X \in \mathbb{E}(N)$ the subnet $N|_X$ is a CA, and
(2) $\forall t, t' \in T.\ {}^\bullet t = {}^\bullet t' \wedge t^\bullet = t'^\bullet \Rightarrow t = t'$.

Unravel nets describe the dependencies among transitions in the executions of a concurrent and distributed device and are similar to *flow nets* [Bou90, BC94]. Flow nets are safe nets in which, for every possible firing sequence, each place can be marked only once. The first condition in Definition 3.5 implies that the subnet consisting of the transitions executed by the firing sequence is a causal net. The second condition, which states that when two transitions have identical pre- and postsets they are the same transition, serves the purpose of ruling out the possibility of having two different transitions that are indistinguishable because they consume and produce the same tokens (places). Similar to flow nets, UN also adhere to the rule that each place can be marked only once. However, unlike flow nets, the requirement that two transitions with the same preset and postset are the same transition (economy efficiency) stipulated by the second condition is integral to its definition. Moreover, flow nets were initially introduced alongside flow event structure [Bou90], a concept which we do not consider in this paper. Lastly, as the process algebra we consider cannot have terms with unguarded choices, the requirement that outgoing arcs are denumerable is always fulfilled and therefore we do not require it explicitly. In an UN, two transitions t and t' are conflicting if they never appear together in an execution, *i.e.*, $\forall X \in \mathbb{E}(N).\ \{t, t'\} \not\subseteq X$, as formally stated below. Given a place s of an unravel net, if ${}^\bullet s$ contains two or more transitions, then they are in conflict.

Proposition 3.6. *Let $N = \langle S, T, F, \mathsf{m}\rangle$ be an UN and $s \in S$ be a place such that $|{}^\bullet s| > 1$. Then $\forall t, t' \in {}^\bullet s.\ \forall X \in \mathbb{E}(N)$, if $t \in X$ and $t' \in X$ then $t = t'$.*

Proof. Take a place $s \in S$ such that $|{}^\bullet s| > 1$ and take $t, t' \in {}^\bullet s$. Assume that there is an execution $X \in \mathbb{E}(N)$ such that X contains both t and t'. As $N|_X$ is a causal net we have that it is acyclic and therefore t must be equal to t' as both produce a token in s. □

It is worth noting that the classical notion of an *occurrence net* [NPW81, Win86] is, in fact, a specific type of UN. In this context, the conflict relation is *inherited* throughout the transitive closure of the flow relation and can be inferred directly from the structure of the net itself. A further evidence that unravel nets generalize occurrence nets is implied also by the fact that flow nets generalize occurrence nets as well [Bou90].

Definition 3.7. An unravel net $N = \langle S, T, F, \mathsf{m}\rangle$ is *complete* whenever $\forall t \in T.\ \exists s_t \in S.\ {}^\bullet s_t = \{t\} \ \wedge\ s_t{}^\bullet = \emptyset$, and $|t^\bullet| > 1$. We use $\mathcal{K}_T$ to denote the subset of S of such places and we call the places in $\mathcal{K}_T$ *key*-places.

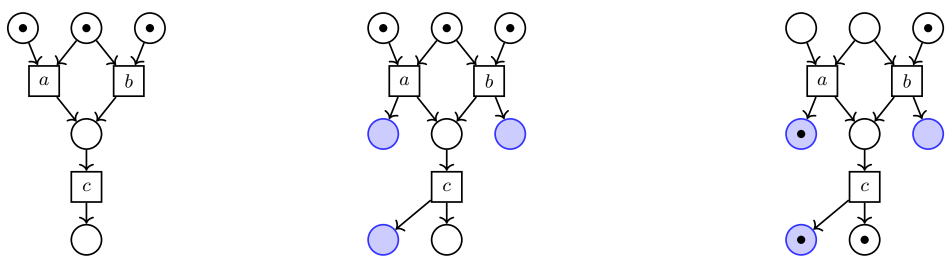
We choose the term *key-places* to denote the places within the set $\mathcal{K}_T$, as they resemble the communication keys in CCKS [PU07]. These keys serve as unique markers used to indicate partial or full synchronisation. Thus, in a complete UN, the execution of a transition t is signalled by the marked place s_t. Given an UN N, it can be turned easily into a complete one by adding for each transition the suitable place, without changing the executions of the net, thus we consider complete UNs only. Completeness comes handy when defining the reversible counterpart of an UN. In a complete UN $N = \langle S, T, F, \mathsf{m}\rangle$, it is easy to see that $|\mathcal{K}_T| = |T|$.

Proposition 3.8. *Let $N = \langle S, T, F, \mathsf{m}\rangle$ be a complete UN and $\mathcal{K}_T$ the key-places. Then, $\mathbb{E}(N) = \mathbb{E}(N')$ where $N' = \langle S', T, F', \mathsf{m}\rangle$ and $S' = S \setminus \mathcal{K}_T$ and $F' = F \cap ((S' \times T) \cup (T \times S'))$.*

The key-places do not play any role in the firing of transitions in a UN.

Example 3.9. Consider the nets in Figure 7. The net N is an unravel net, that has the two maximal executions delineated by the following sequences: first a, followed by c, or first b, then c. The net N is not complete due to the absence of key-places associated with transitions a, b, and c. Transitions a and b lack key-places because each place in their postsets possesses an outgoing arc. Transition c also lacks a key-place because it has just one place in its postset. The net N' is derived from N by augmenting each transition with a key-place, rendering N' a complete net. These key-places serve the purpose of recording executed transitions. Consequently, in executions such as first a and then c, tokens are placed in the key-places corresponding to a and c.

In a complete net, certain key-places can be determined unambiguously, such as those associated with transitions a and b in Figure 7B. However, for others, the selection is somewhat arbitrary, as one place is chosen among alternative options. This is exemplified by the key-place linked to transition c in Figure 7B



(A) N (B) N' (C) N'

FIGURE 7. An UN N, its complete version N' and the net N' after the execution of a and b

3.3. **Reversible unravel nets.** The definition of *reversible unravel nets* builds upon that of the *reversible occurrence nets* of [MMP+20], extending the notion just as unravel nets generalise upon occurrence nets.

Definition 3.10. A *reversible unravel net* (rUN for short) is a quintuple $N = \langle S, T, U, F, \mathsf{m}\rangle$ such that

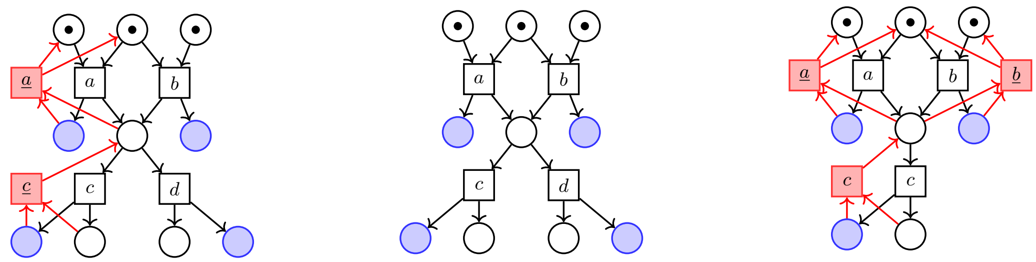
(1) $U \subseteq T$ and $\forall u \in U.\ \exists!\ t \in T \setminus U$ such that ${}^\bullet u = t^\bullet$ and $u^\bullet = {}^\bullet t$, and
(2) $N|_{T\setminus U}$ is a complete unravel net and $\langle S, T, F, \mathsf{m}\rangle$ is a safe one.

The transitions in U are the reversing ones; hence, we often say that a reversible unravel net N is *reversible with respect to* U. A reversing transition u is associated with a unique non-reversing transition t (condition 1) and its effects are intended to *undo* t. This fact ensures the existence of an injective mapping $h : U \to T \setminus U$, which consequently implies that each reversible transition is accompanied by precisely one corresponding reversing transition. The second condition stipulates that when disregarding all reversing transitions, the resulting subnet is indeed a complete unravel net and the net itself is a safe net.

Along the lines of [MMP+20], we can prove that the set of reachable markings of a reversible unravel net is not influenced by performing a reversing transition.

Proposition 3.11. *Let* $N = \langle S, T, U, F, \mathsf{m}\rangle$ *be an* rUN. *Then* $\mathcal{M}_N = \mathcal{M}_{N|_{T\setminus U}}$.

Proof. Clearly $\mathcal{M}_{N|_{T\setminus U}} \subseteq \mathcal{M}_N$. For the other inclusion, we first observe that if $\mathsf{m}\,[t\rangle$ then $t \in T \setminus U$ as none of the transitions in U is enabled at the initial marking. Consider now an fs $\sigma\,[u\rangle\,m$, with $u \in U$, and w.l.o.g. assume that all the transitions in σ belong to $T \setminus U$,



(A) N (B) $N|_{T\setminus U}$ (C) $\overleftarrow{N'}$

Figure 8. An rUN N with reversing transitions $U = \{\underline{a}, \underline{c}\}$, the UN $N|_{T\setminus U}$ and the complete $\overleftarrow{N'}$ associated to the net N' in Figure 7B

i.e., $X_\sigma \subseteq T \setminus U$. We construct an fs leading to m which does not contain any transition in U. As $\sigma\,[u\rangle$ we have that ${}^\bullet u \subseteq \mathit{lead}(\sigma)$ and this implies that the transition $h(u) \in X_\sigma$. We can then write σ as $\sigma'\,[h(u)\rangle\,\sigma''$ and none of the transitions in σ'' uses the tokens produced by $h(u)$ as $N|_{X_\sigma}$ is a subnet of $N|_{T\setminus U}$, which is a complete UN. Therefore we have that the transitions in the fs $\mathit{lead}(\sigma')\,[h(u)\rangle\,\sigma''$ can be rearranged in a fs $\sigma'''\,[h(u)\rangle\,\mathit{lead}(\sigma)$. Observing that the effects of firing u at $\mathit{lead}(\sigma)$ are producing the tokens in places ${}^\bullet h(u)$ we have that the fs we are looking for is obtained executing the transitions in σ' followed by the ones in σ''' and the reached marking is precisely $\mathit{lead}(\sigma)$. Hence also $\mathcal{M}_N \subseteq \mathcal{M}_{N|_{T\setminus U}}$ holds. □

A consequence of this fact is that each marking can be reached by using just *forward transition.*

Given an unravel net and a subset of transitions to be reversed, it is straightforward to obtain a reversible unravel net.

Proposition 3.12. *Let* $N = \langle S, T, F, \mathsf{m}\rangle$ *be a complete unravel net and let* $U \subseteq T$ *be the set of transitions to be reversed. Define* $\overleftarrow{N}^U = \langle S', T', U', F', \mathsf{m}'\rangle$ *where* $S = S'$, $U' = U \times \{\mathtt{r}\}$, $T' = (T \times \{\mathtt{f}\}) \cup U'$,

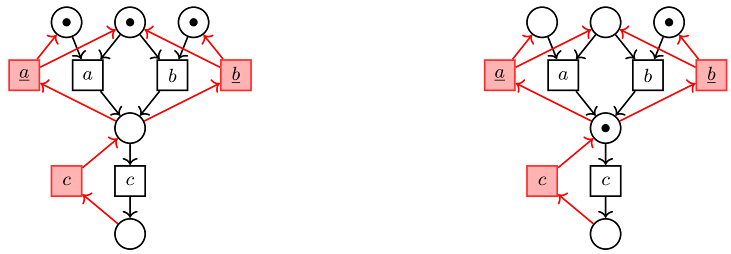
$$\begin{aligned} F' = \;& \{(s,(t,\mathtt{f})) \mid (s,t) \in F\} \cup \{((t,\mathtt{f}),s) \mid (t,s) \in F\} \cup \\ & \{(s,(t,\mathtt{r})) \mid (t,s) \in F\} \cup \{((t,\mathtt{r}),s) \mid (s,t) \in F\} \end{aligned}$$

and $\mathsf{m}' = \mathsf{m}$*. Then* $\overleftarrow{N}^U$ *is a reversible unravel net.*

Proof. We check the conditions of Definition 3.10. The first condition is satisfied as we observe that for each transition in $(t,\mathtt{r}) \in U'$, there exists a unique corresponding transition $(t,\mathtt{f}) \in T \times \{\mathtt{f}\}$; moreover, ${}^\bullet(t,\mathtt{r}) = (t,\mathtt{f})^\bullet$ and $(t,\mathtt{r})^\bullet = {}^\bullet(t,\mathtt{f})$. The second one depends on the fact that N is a complete UN. Finally N is, up to the renaming of transitions, equal to $\overleftarrow{N}^U|_{U'}$, which is a complete unravel net. Finally, $\overleftarrow{N}^U$ is trivially safe as N is safe. □

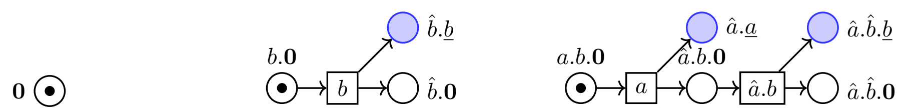
The construction above simply adds as many events (transitions) as transitions to be reversed in U. The preset of each added event is the postset of the corresponding event to be reversed, and its postset is the preset of the event to be reversed. We write $\overleftarrow{N}$ instead of $\overleftarrow{N}^T$ when $N = \langle S, T, F, \mathsf{m}\rangle$, i.e., when every transition is reversible. The reversible unravel net obtained by reversing every transition is depicted in Figure 8C.

To clarify the crucial role played by key-places, consider the UN N depicted Figure 7A. Simply adding the reversing transitions in accordance with Proposition 3.12 would yield the



(A) N (B) N after executing a

Figure 9. An rUN N with reversing transitions $U = \{\underline{a}, \underline{c}\}$ and the net N after the firing of the transition a.



(A) $\mathcal{N}(\mathbf{0})$ (B) $\mathcal{N}(b.\mathbf{0})$ (C) $\mathcal{N}(a.b.\mathbf{0})$

Figure 10. Example of nets corresponding to CCS processes

net shown in Figure 9A. However, this net is not an rUN, as the net obtained by removing the reversing transitions is not complete. Now, consider the marking after firing a, as depicted in Figure 9B. With this marking, the reversing transition $\underline{b}$ is enabled and can be executed, contradicting the expectation that a transition can only be reversed if it has been previously executed. The inclusion of the key-places for transitions a, b and c resolves this problem.

4. CCS processes as unravel nets

4.1. **Encoding of CCS processes.** We now recall the encoding of CCS terms into Petri nets due to Boudol and Castellani [BC94]. It is worth noting that the original encoding was on *proved terms* instead of plain CCS. The difference between proved terms and CCS is that in a proved term the labels carry the position of the process which did the action. Hence, we will use *decorated* versions of labels. For instance, $\hat{a}.b$ denotes an event b that has been preceded by the occurrence of a in the term $a.b$. Analogously, labels carry also information about the syntactical structure of a term, actions corresponding to subterms of a choice and of a parallel composition are also decorated with an index i that indicates the subterm that performs the action. An interesting aspect of this encoding is that these information is reflected in the name of the places and the transitions of the nets, which simplifies the formulation of the behavioural correspondence of a term and its associated net. We write $\ell(\_)$ for the function that removes decorations for a name, e.g., $\ell(\hat{a}.\hat{b}.c) = c$.

We are now in place to define and comment the encoding of a CCS term into a net. The encoding is inductively defined on the structure of the CCS process. For a CCS process P, its encoded net is $\mathcal{N}(P) = \langle S_P, T_P, F_P, \mathsf{m}_P \rangle$. The net corresponding to the inactive process $\mathbf{0}$, is just a net with just one marked place and with no transition, that is:

Definition 4.1. The net $\mathcal{N}(\mathbf{0}) = \langle \{\mathbf{0}\}, \emptyset, \emptyset, \{\mathbf{0}\} \rangle$ is the net associated to $\mathbf{0}$ and it is called *zero*.

To ease notation in the constructions we are going to present, we adopt the following conventions: let $X \subseteq S \cup T$ be a set of places and transitions, we write $\hat{\alpha}.X$ for the set $\{\hat{\alpha}.x \mid x \in X\}$ containing the *decorated* versions of places and transitions in X. Analogously we lift this notation to relations: if R is a binary relation on $(S \cup T)$, then $\hat{\alpha}.R = \{(\hat{\alpha}.x, \hat{\alpha}.y) \mid (x, y) \in R\}$ is a binary relation on $(\alpha.S \cup \alpha.T)$.

The net $\mathcal{N}(\alpha.P)$ corresponding to a process $\alpha.P$ extends $\mathcal{N}(P)$ with two extra places $\alpha.P$ and $\hat{a}.\underline{\alpha}$ and one transition α. The place $\alpha.P$ stands for the process that executes the prefix α and follows as P. The place $\hat{a}.\underline{\alpha}$ is not in the original encoding of [BC94]; we have add it to ensure that the obtained net is complete, which is essential for the definition of the reversible net. This will become clearer when commenting the encoding of the parallel composition. It should be noted that this addition does not interfere with the behaviour of the net, since all added places are final. Also a new transition, named α is created and added to the net, and the flow relation is updated accordingly. We use colours to indicate the name of places that serve as key places. The input and output places of the added transitions vary depending on the CCS operator under consideration.

Figures 10A, 10B and 10C report respectively the encodings of the inactive process, of the process $b.\mathbf{0}$ and $a.b.\mathbf{0}$. Moreover the aforementioned figures systematically show how the prefixing operator is rendered into Petri nets. As a matter of fact, the net $a.b.\mathbf{0}$ is built starting from the net corresponding to $b.\mathbf{0}$ by adding the prefix a. We note that also the label of transitions is affected by appending the label of the new prefix at the beginning. This is rendered in Figure 10C where the transition mimicking the action b is labeled as $\hat{a}.b$ indicating that an a was done before b. In what follows we will often omit such representation from figures.

Definition 4.2. Let P a CCS process and $\mathcal{N}(P) = \langle S_P, T_P, F_P, \mathsf{m}_P\rangle$ be the associated net. Then $\mathcal{N}(\alpha.P)$ is the net $\langle S_{\alpha.P}, T_{\alpha.P}, F_{\alpha.P}, \mathsf{m}_{\alpha.P}\rangle$ where

$$\begin{array}{rcl} S_{\alpha.P} & = & \{\alpha.P, \hat{\alpha}.\underline{\alpha}\} \cup \hat{\alpha}.S_P \\ T_{\alpha.P} & = & \{\alpha\} \cup \hat{\alpha}.T_p \\ F_{\alpha.P} & = & \{(\alpha.P, \alpha), (\alpha, \hat{\alpha}.\underline{\alpha})\} \cup \{(\alpha, \hat{\alpha}.b) \mid b \in \mathsf{m}_P\} \cup \hat{\alpha}.F_P \\ \mathsf{m}_{\alpha.P} & = & \{\alpha.P\} \end{array}$$

The set of *key*-places of $\mathcal{N}(\alpha.P)$ is $\hat{\alpha}.\mathcal{K}_{T_P} \cup \{\hat{\alpha}.\underline{\alpha}\}$, where $\mathcal{K}_{T_P}$ are the *key*-places of $\mathcal{N}(P)$.

As we have done for prefixes, for a set X of transitions and places we write $\|_{\mathsf{i}}X$ for $\{\|_{\mathsf{i}}x \mid x \in X\}$, which straightforwardly lifts to relations. We do the same with $+_{\mathsf{i}}$ and $\backslash_{\mathsf{a}}$, which are the decorations for the sum and the restriction.

The encoding of parallel goes along the line of the prefixing one. Also in this case we have to decorate the places (and transitions) with the position of the term in the syntax tree. To this end, each branch of the parallel is decorated with $\|_{\mathsf{i}}$ with i being the i-th position. Regarding the transitions, we have to add all the possible synchronisations among the processes in parallel. This is why, along with the transitions of the branches (properly decorated with $\|_{\mathsf{i}}$) we have to add extra transitions to indicate the possible synchronisation. Naturally a synchronisation is possible when one label is the co-label of the other transition. Figure 11A shows the net corresponding to the process $a.b \parallel \overline{a}.c$. As we can see, the encoding builds upon the encoding of $a.b$ and $\overline{a}.c$, by (i) adding to all the places and transitions whether the branch is the left one or the right one and (ii) adding an extra transition and place for the only possible synchronisation. We add an extra place (in line with the prefixes) to mark the fact that a synchronisation has taken place. Let us note that the extra places $\underline{a}$, $\underline{\overline{a}}$ and $\underline{\tau}$ are used to understand whether the two prefixes have done a partial synchronisation or they contributed to do a synchronisation. Suppose, for example, that the net had not such places, and suppose that we have two tokens in the places $\|_0\hat{a}.b$ and $\|_1\hat{\bar{a}}.b$. Now, how can we understand whether these two tokens are the result of the firing sequence a,$\bar{a}$ or they are the result of the τ transition? It is impossible, but by using the aforementioned extra-places, which are instrumental to tell if a single prefix has executed, we can distinguish the τ from the sequence $a\bar{a}$ and then reverse accordingly.

Definition 4.3. Let $\mathcal{N}(P_1)$ and $\mathcal{N}(P_2)$ be the nets associated to the processes P_1 and P_2. Then $\mathcal{N}(P_1\|P_2)$ is the net $\langle S_{P_1\|P_2}, T_{P_1\|P_2}, F_{P_1\|P_2}, \mathsf{m}_{P_1\|P_2}\rangle$ where

$$\begin{array}{rcl}
S_{P_1\|P_2} &=& \|_0 S_{P_1} \cup \|_1 S_{P_2} \cup \{s_{\{t,t'\}} \mid t \in T_{P_1} \wedge t' \in T_{P_2} \wedge \overline{\ell(t)} = \ell(t')\}\\
T_{P_1\|P_2} &=& \|_0 T_{P_1} \cup \|_1 T_{P_2} \cup \{\{t,t'\} \mid t \in T_{P_1} \wedge t' \in T_{P_2} \wedge \overline{\ell(t)} = \ell(t')\}\\
F_{P_1\|P_2} &=& \|_0 F_{P_1} \cup \|_1 F_{P_2}\\
&& \cup\{(\{t,t'\}, s_{\{t,t'\}}) \mid t \in T_{P_1} \wedge\ t' \in T_{P_2}\ \wedge\ \overline{\ell(t)} = \ell(t')\}\\
&& \cup\{(\|_i s, \{t_1,t_2\}) \mid (s,t_i) \in F_{P_i}\} \cup \{(\{t_1,t_2\}, \|_i s) \mid (t_i, s) \in F_{P_i}\ \wedge\ s \notin \mathcal{K}_{T_{P_i}}\}\\
\mathsf{m}_{P_1\|P_2} &=& \|_0 \mathsf{m}_{P_1} \cup \|_1 \mathsf{m}_{P_2}
\end{array}$$

The *key*-places of the resulting net are the following.

$$\|_0\mathcal{K}_{T_{P_1}} \cup \|_1\mathcal{K}_{T_{P_2}} \cup \{s_{\{t,t'\}} \mid t \in T_{P_1} \wedge t' \in T_{P_2} \wedge \overline{\ell(t)} = \ell(t')\}$$

They are obtained by properly renaming the ones arising from the encoding of the branches and those corresponding to the synchronisations of the components.

The encoding of the choice operator is similar to the parallel one. The only difference is that we do not have to deal with possible synchronisations since the branches of a choice are mutually exclusive. Figure 11B illustrates the net corresponding to the process $a.b + \bar{a}.c$. As in the previous examples, the net is built upon the subnets representing $a.b$ and $\bar{a}.c$.

Definition 4.4. Let $\mathcal{N}(P_i)$ be the net associated to the processes P_i for $i \in I$. Then $+_{i\in I}P_i$ is the net $\langle S_{+_{i\in I}P_i}, T_{+_{i\in I}P_i}, F_{+_{i\in I}P_i}, \mathsf{m}_{+_{i\in I}P_i}\rangle$ where:

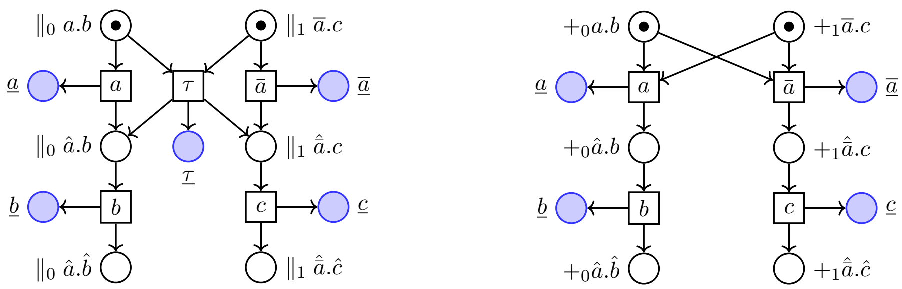
$$\begin{array}{rcl}
S_{+_{i\in I}P_i} &=& \cup_{i\in I} +_i S_{P_i}\\
T_{+_{i\in I}P_i} &=& \cup_{i\in I} +_i T_{P_i}\\
F_{+_{i\in I}P_i} &=& \{(+_i x, +_i y) \mid (x,y) \in F_{P_i}\} \cup \{(+_j s, +_i t) \mid s \in \mathsf{m}_{P_j} \wedge {}^\bullet t \in \mathsf{m}_{P_i} \wedge i \neq j\}\\
\mathsf{m}_{+_{i\in I}P_i} &=& \cup_{i\in I} +_i \mathsf{m}_{P_i}.
\end{array}$$

In this case the *key*-places of $+_{i\in I}P_i$ are just the union of all *key*-places after the suitable renaming, i.e., $\cup_{i\in I} +_i \mathcal{K}_{T_{P_i}}$.

We write T^a for the set all transitions in T labelled by a or $\bar{a}$, i.e., $\{t \in T \mid \ell(t) = a \vee \ell(t) = \bar{a}\}$. The encoding of the hiding operator simply removes all transitions whose labels corresponds to actions performed over the restricted name and the key-places associated to these transitions.

Definition 4.5. Let P a CCS process and $\mathcal{N}(P) = \langle S_P, T_P, F_P, \mathsf{m}_P\rangle$ be the associated net. Then $\mathcal{N}(P \setminus a)$ is the net $\langle S_{P\setminus a}, T_{P\setminus a}, F_{P\setminus a}, \mathsf{m}_{P\setminus a}\rangle$ where

$$\begin{array}{rcl}
S_{P\setminus a} &=& \setminus_a(S_P \setminus \mathcal{K}_{T^a})\\
T_{P\setminus a} &=& \setminus_a(T_P \setminus T^a)\\
F_{P\setminus a} &=& \{(\setminus_a s, \setminus_a t) \mid s \in S_P \setminus \mathcal{K}_{T^a}\ \wedge\ (s,t) \in F_P\ \wedge\ t \notin T^a\} \cup\\
&& \{(\setminus_a t, \setminus_a s) \mid s \in S_P \setminus \mathcal{K}_{T^a}\ \wedge\ (t,s) \in F_P\ \wedge\ t \notin T^a\}\\
\mathsf{m}_{P\setminus a} &=& \setminus_a \mathsf{m}_P
\end{array}$$



(A) $\mathcal{N}(a.b \parallel \overline{a}.c)$ (B) $\mathcal{N}(a.b + \overline{a}.c)$

FIGURE 11. Example of nets corresponding to CCS parallel and choice operator. We omit the trailing $\mathbf{0}$

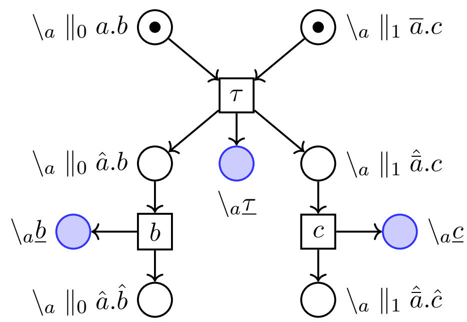


FIGURE 12. The net $\mathcal{N}((a.b \parallel \overline{a}.c) \setminus a)$

In this case, as the number of *firable* transitions decreases, a corresponding decrease is observed in the number of *key*-places. Hence, $\mathcal{K}_{\backslash_{\mathsf{a}}(T_P \setminus T^a)} = \backslash_{\mathsf{a}}(\mathcal{K}_{T_P} \setminus \mathcal{K}_{T^a})$. Figure 12 shows the net corresponding to the CCS process $(a.b \parallel \overline{a}.c) \setminus a$. Observe that certain transitions are removed (those labeled with the restricted action), along with their associated key-places, although this is not strictly necessary. In fact, after the removal of the transitions, the respective places remain isolated because the only connected transitions have been removed.

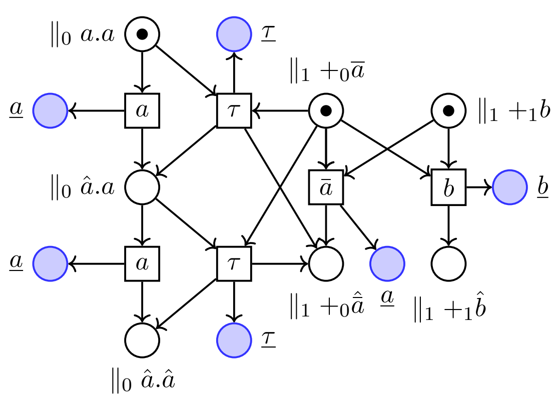


FIGURE 13. A complex example: $\mathcal{N}(a.a \parallel \overline{a} + b)$

In Figure 13, a more complex example is depicted, illustrating the net corresponding to the process $a.a \parallel \overline{a} + b$. In this case, the process on the right of the parallel composition can synchronise with the one on the left one in two different occasions. This is why there are two different transitions representing the synchronisations. However, due to the nature of the process on the right-hand side being a choice, there is a possibility that the right branch of that choice gets executed, thereby preventing the synchronisation from occurring. As the right branch of the parallel constitutes a choice between two options, the encoding designates these branches as '$\|_1 +_0$' and '$\|_1 +_0$' respectively. These labels serve to identify the left and right branches of the choice, which is situated within the right branch of the parallel operator.

The following proposition is instrumental for the main correspondence result.

Proposition 4.6. *The nets defined in Definitions 4.1 to 4.5 are* complete *unravel nets.*

Proof. By induction on the structure of a CCS process. Clearly the net $\mathcal{N}(\mathbf{0})$ is an unravel net and it is trivially complete because it has no transition. Assume now that $\mathcal{N}(P) = \langle S_P, T_P, F_P, \mathsf{m}_P \rangle$ associated with the CCS process P is a complete UN. Also $\mathcal{N}(\alpha.P) = \langle S_{\alpha.P}, T_{\alpha.P}, F_{\alpha.P}, \mathsf{m}_{\alpha.P} \rangle$ is an UN as it is obtained by adding a new transition α that precedes all transitions in T_P. Moreover, a new key-place $\hat{\alpha}.\underline{\alpha}$ is added for such transition. Assuming now that $\mathcal{N}(P_1)$ and $\mathcal{N}(P_2)$ are the two complete UNs associated with P_1 and P_2. The net $\mathcal{N}(P_1 \| P_2)$ is an UN as the two components, when *synchronise*, have the effect of the local changes beside the key-places. For each synchronising transition $\{t, t'\}$, a corresponding key-place $s_{\{t,t'\}}$ exists, rendering the net complete. Similarly, $+_{i \in I} P_i$ is a complete unravel net, as each $\mathcal{N}(P_i)$ is a complete unravel net. The additional flow arcs ensure that only transitions of a specific component are executed. Lastly, $\mathcal{N}(P \setminus a)$ is complete because the elimination of transitions does not add any new behaviour. □

4.2. **Encoding of RCCS processes.** We are now at the point where we can define the network that corresponds to an RCCS process. So far, our focus has been on encoding CCS processes into nets. Since RCCS is built upon CCS processes, our encoding of RCCS naturally builds upon the encoding of CCS. To do so, we first introduce the concept of ancestor, i.e., the initial process from which an RCCS process is derived. Notably, in the context of our discussion involving coherent RCCS processes (as defined in Definition 2.4), an RCCS process invariably possesses an ancestor.

The ancestor $\rho(R)$ of an RCCS process R can be calculated through syntactical analysis of R, as all information about its past is stored within memories. The sole instance in which a process must wait for its counterpart is during a memory fork, denoted as $\langle 1 \rangle$ or $\langle 2 \rangle$.

Definition 4.7. Given a coherent RCCS process R, its ancestor $\rho(R)$ is derived by using the inference rules of Figure 14. The rules use the pre-congruence relation $\preceq$ defined as $\equiv$ (see Figure 6) with the exception that rule SPLIT can be only applied from right to left.

Example 4.8. Consider the RCCS term R below:

$$\begin{aligned} R = \ & \langle 0 \rangle \cdot \langle m_2, a^1, \mathbf{0} \rangle \cdot \langle 0 \rangle \cdot \langle \rangle \triangleright b \ \parallel \ \langle 1 \rangle \cdot \langle m_2, a^1, \mathbf{0} \rangle \cdot \langle 0 \rangle \cdot \langle \rangle \triangleright c \ \parallel \\ & \langle m_1, \overline{a}^1, \mathbf{0} \rangle \cdot \langle 0 \rangle \cdot \langle 1 \rangle \cdot \langle \rangle \triangleright \mathbf{0} \ \parallel \ \langle 1 \rangle \cdot \langle 1 \rangle \cdot \langle \rangle \triangleright d \end{aligned}$$

$$\langle \_, \alpha_z^z, \sum_{i \in I\setminus\{z\}} \alpha_i.P_i\rangle \cdot m \triangleright P \rightharpoondown m \triangleright \sum_{i\in I} \alpha_i.P_i \ (\textsc{Act})$$

$$\frac{R \preceq R' \qquad R' \rightharpoondown S' \qquad S' \preceq S}{R \rightharpoondown S}\ (\textsc{Pre}) \qquad \frac{R \rightharpoondown R'}{R \parallel S \rightharpoondown R' \parallel S}\ (\textsc{Par})$$

$$\frac{R \rightharpoondown R'}{R\backslash a \rightharpoondown R'\backslash a}\ (\textsc{Res}) \qquad \frac{R \rightharpoondown^* \langle\rangle \triangleright P}{\rho(R) = P}\ (\textsc{Init})$$

Figure 14. Ancestor inference rules

with $m_1 = \langle 0\rangle \cdot \langle\rangle$ and $m_2 = \langle 0\rangle \cdot \langle 1\rangle \cdot \langle\rangle$ By applying the inference rules in Figure 14, we compute its ancestor as follows:

$$\begin{aligned}
R &\rightharpoondown \langle m_2, a^1, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle\rangle \triangleright (b \parallel c) \parallel && (\textsc{Act})\\
&\quad \langle m_1, \overline{a}^1, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright \mathbf{0} \ \parallel\ \langle 1\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright d \\
&\rightharpoondown \langle 0\rangle \cdot \langle\rangle \triangleright a.(b \parallel c) \parallel \langle m_1, \overline{a}^1, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright \mathbf{0} \parallel\ \langle 1\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright d && (\textsc{Act})\\
&\rightharpoondown \langle 0\rangle \cdot \langle\rangle \triangleright a.(b \parallel c) \parallel \langle 0\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright \overline{a} \parallel\ \langle 1\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright d && (\textsc{Pre})\\
&\rightharpoondown \langle 0\rangle \cdot \langle\rangle \triangleright a.(b \parallel c) \parallel \langle 1\rangle \cdot \langle\rangle \triangleright (\overline{a} \parallel d) && (\textsc{Pre})\\
&\rightharpoondown \langle\rangle \triangleright (a.(b \parallel c) \parallel \overline{a} \parallel d)
\end{aligned}$$

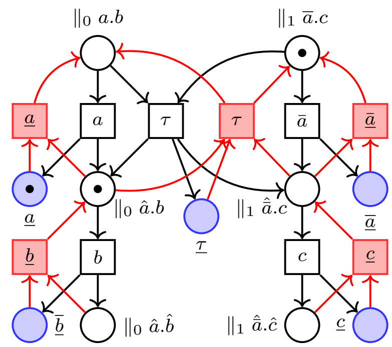
that is $\rho(R) = a.(b \parallel c) \parallel \overline{a} \parallel d$. Let us note that reversing a synchronisation is achieved by applying the Act rule twice—each monitored process can undo its respective part of the synchronisation. This is possible due to the coherence of processes. Essentially, upon encountering a synchronisation event, a process possesses adequate information to revert to its previous local state. Conversely, when encountering a split event, the process must await its siblings (as per the Pre rule) to reconstruct the parallel process.

Lemma 4.9. *For any coherent RCCS process R its ancestor $\rho(R)$ exists and it is unique.*

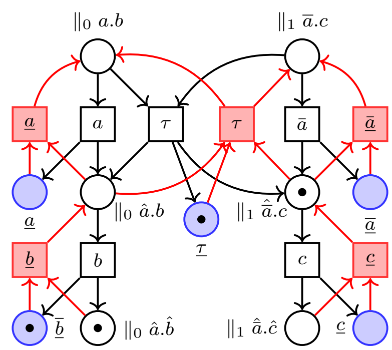
Proof. Since R is a coherent process then there exists a CCS process P such that $\langle\rangle \triangleright P \hookrightarrow^* R$. By Property 2.8 we have that $\langle\rangle \triangleright P \rightarrow^* R$, and by applying Corollary 2.7 we obtain that $R \rightsquigarrow^* \langle\rangle \triangleright P$. The proof is then by induction on the number n of reductions contained in $\rightsquigarrow^*$ and by noticing that for each application of $\rightsquigarrow$ there exists a corresponding rule of $\rightharpoondown$. □

There is a tight correspondence between RCCS memories and transitions/places names. That is, a memory contains all the information to recover the path from the root to the process itself. To this end, we introduce the function $\texttt{path}(\cdot)$, which is inductively defined as follows

$$\begin{aligned}
&\texttt{path}(m \cdot \langle m', \alpha^i, \mathbf{0}\rangle) = \texttt{path}(m \cdot \langle *, \alpha^i, \mathbf{0}\rangle) = \hat{\alpha}.\texttt{path}(m)\\
&\texttt{path}(m \cdot \langle m', \alpha^i, Q\rangle) = \texttt{path}(m \cdot \langle *, \alpha^i, Q\rangle) = +_{\mathsf{i}}\hat{\alpha}.\texttt{path}(m)\\
&\texttt{path}(m \cdot \langle i\rangle) = \parallel_{\mathsf{i}} \texttt{path}(m)\\
&\texttt{path}(\langle\rangle) = \epsilon
\end{aligned}$$



(A) $\overleftarrow{\mathcal{N}(R_1)}$



(B) $\overleftarrow{\mathcal{N}(R_2)}$

FIGURE 15. Example of nets corresponding to RCCS process R_1 and R_2

Example 4.10. Let us consider the RCCS processes R_1 and R_2 defined below

$$R_1 = \langle *, a^1, \mathbf{0} \rangle \cdot \langle 0 \rangle \cdot \langle \rangle \triangleright b \parallel \langle 1 \rangle \cdot \langle \rangle \triangleright \overline{a}.c$$
$$R_2 = \langle *, b^1, \mathbf{0} \rangle \cdot \langle m_2, a^1, \mathbf{0} \rangle \cdot \langle 0 \rangle \cdot \langle \rangle \triangleright \mathbf{0} \parallel \langle m_1, \overline{a}^1, \mathbf{0} \rangle \cdot \langle 1 \rangle \cdot \langle \rangle \triangleright c$$

with $m_i = \langle i \rangle \cdot \langle \rangle$. Their corresponding nets are shown in Figure 15.

We have that the path of the left process is $\texttt{path}(\langle *, a^1, \mathbf{0} \rangle \cdot \langle 0 \rangle \cdot \langle \rangle) = \|_0 \hat{a}$, while the path of the right process is $\texttt{path}(\langle 1 \rangle \cdot \langle \rangle) = \|_1$.

The encoding of an RCCS process should yield an equivalent net to that of its ancestor, with the only potential distinction being the marking – indicating the specific locations where tokens are placed. And such positions are inferred from the information stored in memories. Following the intuitions in Section 4, we will treat names of places and transitions as strings. When we write ϕX, where X is a set of strings and $\phi \in \{\|_\mathsf{i}, +_\mathsf{i}, \hat{\alpha}, \backslash_\mathsf{a}\}$, we are indicating the set $\{\phi x \mid x \in X\}$. Also, we will indicate with $\tilde{\phi}$ the sequence $\phi_1 \cdots \phi_n$ with

$\phi_i \in \{\|_{\mathsf{j}}, +_{\mathsf{j}}, \hat{\alpha}, \backslash_{\mathsf{a}}\}$. Then the *marking* function $\mu(\cdot)$ is inductively defined as follows:

$$\begin{aligned}
\mu(R \parallel S) &= \mu(R) \bowtie \mu(S)\\
\mu(R\backslash a) &= \backslash_{\mathsf{a}}\mu(R)\\
\mu(m \cdot \langle m_1, \alpha^i, \mathbf{0}\rangle \cdot \langle\rangle \triangleright P) &= \{\alpha, m_1\} \cup \hat{\alpha}.\mu(m \cdot \langle\rangle \triangleright P)\\
\mu(m \cdot \langle m_1, \alpha^i, Q\rangle \cdot \langle\rangle \triangleright P) &= \{+_{\mathsf{i}}\alpha, m_1\} \cup +_{\mathsf{i}}\hat{\alpha}.\mu(m \cdot \langle\rangle \triangleright P)\\
\mu(m \cdot \langle *, \alpha^i, \mathbf{0}\rangle \cdot \langle\rangle \triangleright P) &= \{\hat{\alpha}.\underline{\alpha}\} \cup \hat{\alpha}.\mu(m \cdot \langle\rangle \triangleright P)\\
\mu(m \cdot \langle *, \alpha^i, Q\rangle \cdot \langle\rangle \triangleright P) &= \{+_{\mathsf{i}}\hat{\alpha}.\underline{\alpha}\} \cup +_{\mathsf{i}}\hat{\alpha}.\mu(m \cdot \langle\rangle \triangleright P)\\
\mu(m \cdot \langle i\rangle \cdot \langle\rangle \triangleright P) &= \|_{\mathsf{i}}\, \mu(m \cdot \langle\rangle \triangleright P)\\
\mu(\langle\rangle \triangleright P) &= \{P\}
\end{aligned}$$

where $\bowtie$ is defined as the usual set union on single element, and as the merge on pairs of the form $\{t_1, m_2\}$ $\{t_2, m_1\}$ where $\{t_1, m_2\} \bowtie \{t_2, m_1\} = s_{\{t_1,t_2\}}$ if $\ell(t_1) = \overline{\ell(t_2)}$ and $t_i = \mathtt{path}(m_i)\alpha_i$ with $\alpha_i = \ell(t_i)$, where $s_{\{t_1,t_2\}}$ is the key place of the synchronisation between transitions t_1 and t_2.

Example 4.11. Let us consider the RCCS processes R_1 and R_2 of Example 4.10. The marking of the process R_1 is

$$\begin{aligned}
&\mu(\langle *, a^1, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle\rangle \triangleright b \parallel \langle 1\rangle \cdot \langle\rangle \triangleright \bar{a}.c)\\
&= (\mu(\langle *, a^1, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle\rangle \triangleright b)) \bowtie (\|_1\mu(\langle\rangle \triangleright \bar{a}.c))\\
&= (\|_0\mu(\langle *, a^1, \mathbf{0}\rangle \cdot \langle\rangle \triangleright b)) \bowtie (\{\|_1\bar{a}.c\})\\
&= (\{\|_0\hat{a}.\underline{a}\} \cup \|_0\hat{a}.\mu(\langle\rangle \triangleright b)) \bowtie \{\|_1\bar{a}.c\}\\
&= \{\|_0\hat{a}.\underline{a}, \|_0\hat{a}.b\} \bowtie \{\|_1\bar{a}.c\}\\
&= \{\|_0\hat{a}.\underline{a}, \|_0\hat{a}.b, \|_1\bar{a}.c\}
\end{aligned}$$

and the marking of the process R_2 is

$$\begin{aligned}
&\mu(\langle *, b^1, \mathbf{0}\rangle \cdot \langle m_2, a^1, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle\rangle \triangleright \mathbf{0} \parallel \langle m_1, \overline{a}^1, \mathbf{0}\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright c)\\
&= (\mu(\langle *, b^1, \mathbf{0}\rangle \cdot \langle m_2, a^1, \mathbf{0}\rangle \cdot \langle 0\rangle \cdot \langle\rangle \triangleright \mathbf{0})) \bowtie (\mu(\langle m_1, \overline{a}^1, \mathbf{0}\rangle \cdot \langle 1\rangle \cdot \langle\rangle \triangleright c))\\
&= (\|_0\mu(\langle *, b^1, \mathbf{0}\rangle \cdot \langle m_2, a^1, \mathbf{0}\rangle \cdot \langle\rangle \triangleright \mathbf{0})) \bowtie (\|_1\mu(\langle m_1, \overline{a}^1, \mathbf{0}\rangle \cdot \langle\rangle \triangleright c))\\
&= \|_0(\{a, m_2\}, \hat{a}.\mu(\langle *, b^1, \mathbf{0}\rangle \cdot \langle\rangle \triangleright \mathbf{0})) \bowtie (\|_1\{\{\overline{a}, m_1\}, \hat{\overline{a}}.\mu(\langle\rangle \triangleright c)\})\\
&= \|_0(\{a, m_2\}, \hat{a}.\{\underline{b}, b\}) \bowtie (\|_1\{\{\overline{a}, m_1\}, \hat{\overline{a}}.c\})\\
&= \{\{\|_0 a, m_2\}, \|_0\hat{a}.\underline{b}, \|_0\hat{a}.\hat{b}\} \bowtie \{\{\|_1\overline{a}, m_1\}, \|_1\hat{\overline{a}}.c\}\\
&= \{\{\|_0 a, \|_1\overline{a}\}, \|_0\hat{a}.\underline{b}, \|_0\hat{a}.\hat{b}, \|_1\hat{\overline{a}}.c\}
\end{aligned}$$

We are now in place to define a property that relates the definitions of $\mu(\cdot)$ and $\mathtt{path}(\cdot)$ with RCCS processes.

Property 4.12. Let $R = m \triangleright \sum_{i\in I} \alpha_i.P_i$ be a RCCS process. For any $z \in I$ such that $R \xrightarrow{m:\alpha_z} \langle *, \alpha_z^z, \sum_{i\in I\setminus\{z\}} \alpha_i.P_i\rangle \cdot m \triangleright P_z$ we have that

$$\mu(\langle *, \alpha_z^z, \sum_{i\in I\setminus\{z\}} \alpha_i.P_i\rangle \cdot m \triangleright P_z) = \mu(R) \setminus \{\mathtt{path}(m)+_{\mathsf{z}}\alpha_z.P_z\}$$
$$\cup\ \{\mathtt{path}(m)+_{\mathsf{z}}\hat{\alpha}_z.\underline{\alpha_z}, \mathtt{path}(m)+_{\mathsf{z}}\alpha_z.P_z\}$$

Proof. The proof is by induction on the size of m. The base case with $m = \langle\rangle$ trivially holds. In the inductive case we have $m = m_1 \cdot e \cdot \langle\rangle$ where e can be $\langle i\rangle$, $\langle *, \beta^i, Q\rangle$ or $\langle m_2, \beta^i, Q\rangle$. We will show the first two cases, with the third being similar to the second one. We have that

$$S_0 = m_1 \cdot \langle\rangle \triangleright \sum_{i\in I} \alpha_i.P_i \xrightarrow{m_1\cdot\langle\rangle:\alpha_z} \langle *, \alpha_z^z, \sum_{i\in I\setminus\{z\}} \alpha_i.P_i\rangle \cdot m_1 \cdot \langle\rangle \triangleright P_z = R_0$$

and by applying inductive hypothesis (on a shorter memory) we have that

$$\mu(S_0) = \mu(R_0) \setminus \{\mathtt{path}(m_1)+_{\mathsf{z}}\alpha_z.P_z\} \cup \{\mathtt{path}(m_1)+_{\mathsf{z}}\hat{\alpha}_z.\underline{\alpha_z}, \mathtt{path}(m_1)+_{\mathsf{z}}\alpha_z.P_z\} \tag{4.1}$$

We proceed by case analysis.

$e = \langle i\rangle$: let us note that $\mathtt{path}(m) = \|_{\mathsf{i}}\mathtt{path}(m_1)$, and that $\mu(R) = \|_{\mathsf{i}}\mu(R_0)$ and $\mu(S) = \|_{\mathsf{i}}\mu(S_0)$. Thanks to Eq. (4.1) we know the form of $\mu(S_0)$, hence

$$\begin{aligned}\mu(S) &= \|_{\mathsf{i}}\mu(S_0)\\ &= \|_{\mathsf{i}}\mu(R_0) \setminus \{\|_{\mathsf{i}}\mathtt{path}(m_1)+_{\mathsf{z}}\alpha_z.P_z\} \cup \{\|_{\mathsf{i}}\mathtt{path}(m_1)+_{\mathsf{z}}\hat{\alpha}_z.\underline{\alpha_z}, \|_{\mathsf{i}}\mathtt{path}(m_1)+_{\mathsf{z}}\alpha_z.P_z\}\\ &= \mu(R) \setminus \{\mathtt{path}(m)+_{\mathsf{z}}\alpha_z.P_z\} \cup \{\mathtt{path}(m)+_{\mathsf{z}}\hat{\alpha}_z.\underline{\alpha_z}, \mathtt{path}(m)+_{\mathsf{z}}\alpha_z.P_z\}\end{aligned}$$

as desired.

$e = \langle *, \beta^i, Q\rangle$: let us note that $\mathtt{path}(m) = +_{\mathsf{i}}\hat{\beta}.\mathtt{path}(m_1)$, and that $\mu(R) = +_{\mathsf{i}}\hat{\beta}.\mu(R_0) \cup \{+_{\mathsf{i}}\hat{\beta}.\underline{\beta}\}$, and $\mu(S) = +_{\mathsf{i}}\hat{\beta}.\mu(S_0) \cup \{+_{\mathsf{i}}\hat{\beta}.\underline{\beta}\}$. Thanks to Eq. (4.1) we know the form of $\mu(S_0)$, hence

$$\begin{aligned}\mu(S) &= +_{\mathsf{i}}\hat{\beta}.\mu(S_0) \cup \{+_{\mathsf{i}}\hat{\beta}.\underline{\beta}\}\\ &= +_{\mathsf{i}}\hat{\beta}.\{\mu(R_0) \setminus \{\mathtt{path}(m_1)+_{\mathsf{z}}\alpha_z.P_z\}\\ &\quad \cup\ \{\mathtt{path}(m_1)+_{\mathsf{z}}\hat{\alpha}_z.\underline{\alpha_z}, \mathtt{path}(m_1)+_{\mathsf{z}}\alpha_z.P_z\}\}\ \cup\ \{+_{\mathsf{i}}\hat{\beta}.\underline{\beta}\}\\ &= +_{\mathsf{i}}\hat{\beta}.\mu(R_0) \setminus \{+_{\mathsf{i}}\hat{\beta}.\mathtt{path}(m_1)+_{\mathsf{z}}\alpha_z.P_z\}\\ &\quad \cup \{+_{\mathsf{i}}\hat{\beta}.\mathtt{path}(m_1)+_{\mathsf{z}}\hat{\alpha}_z.\underline{\alpha_z}, +_{\mathsf{i}}\hat{\beta}.\mathtt{path}(m_1)+_{\mathsf{z}}\alpha_z.P_z\}\} \cup \{+_{\mathsf{i}}\hat{\beta}.\underline{\beta}\}\\ &= +_{\mathsf{i}}\hat{\beta}.\mu(R_0) \setminus \{\mathtt{path}(m)+_{\mathsf{z}}\alpha_z.P_z\}\\ &\quad \cup \{\mathtt{path}(m)+_{\mathsf{z}}\hat{\alpha}_z.\underline{\alpha_z}, \mathtt{path}(m)+_{\mathsf{z}}\alpha_z.P_z\}\} \cup \{+_{\mathsf{i}}\hat{\beta}.\underline{\beta}\}\\ &= \mu(R) \setminus \{\mathtt{path}(m)+_{\mathsf{z}}\alpha_z.P_z\} \cup \{\mathtt{path}(m)+_{\mathsf{z}}\hat{\alpha}_z.\underline{\alpha_z}, \mathtt{path}(m)+_{\mathsf{z}}\alpha_z.P_z\}\}\end{aligned}$$

as desired. □

As a consequence, we have the following corollary.

Corollary 4.13. *Let $R = m \triangleright \alpha.P$ be a RCCS process. For any $z \in I$ such that $R \xrightarrow{m:\alpha_z} \langle *, \alpha, \mathbf{0}\rangle \cdot m \triangleright P$ we have that*

$$\mu(\langle *, \alpha, \mathbf{0}\rangle \cdot m \triangleright P) = \mu(R) \setminus \{\mathtt{path}(m)\alpha.P\} \cup \{\mathtt{path}(m)\hat{\alpha}.\underline{\alpha}, \mathtt{path}(m)\alpha.P\}$$

We are now ready to formalise the reversible net corresponding to an RCCS process.

Definition 4.14. Let R be an RCCS term with $\rho(R) = P$. Then $\overleftarrow{\mathcal{N}(R)}$ is the net $\langle S, T, F, \mu(R)\rangle$ where $\mathcal{N}(P) = \langle S, T, F, \mathsf{m}\rangle$.

Note that the reversible net corresponding to a coherent RCCS process R retains identical places, transitions, and flow relationships as the ancestor of R. The sole divergence lies in the marking, which is derived through the utilisation of the computational history stored within the memories of R. The following Proposition is a consequence of the Proposition 4.6, Lemma 4.9 and of the definition of $\mu(\cdot)$.

Proposition 4.15. *Let R be an RCCS term with $\rho(R) = P$. Then $\overleftarrow{\mathcal{N}(R)}$ is a reversible unravel net.*

4.3. **Correctness result.** We prove the correctness of our encoding in terms of a behavioural equivalence. To this aim we reformulate the definition of *forward and reverse bisimilarity* [PU07], initially stated for CCSK, to cope with RCCS terms and Petri nets.

Definition 4.16 (Forward and reverse bisimulation)**.** Let R a coherent RCCS process and $N = \langle S, T, F, \mathsf{m}\rangle$ an rUN. The relation $\mathcal{R}$ is a forward reverse bisimulation if whenever $(R, N) \in \mathcal{R}$:

(1) if $R \xrightarrow{m:\alpha} R'$ then there exist $t \in T$ and m' such that $\mathsf{m}\,[t\rangle\,\mathsf{m}'$, $t = (\mathtt{path}(m)\alpha, \mathtt{f})$ and $(R', \langle S, T, F, \mathsf{m}'\rangle) \in \mathcal{R}$;
(2) if $R \overset{m:\alpha}{\rightsquigarrow} R'$ then there exist $t \in T$ and m' such that $\mathsf{m}\,[t\rangle\,\mathsf{m}'$, $t = (\mathtt{path}(m)\alpha, \mathtt{r})$ and $(R', \langle S, T, F, \mathsf{m}'\rangle) \in \mathcal{R}$;
(3) if $R \xrightarrow{m_1,m_2:\tau} R'$ then there exist $(t_1, \mathtt{f}), (t_2, \mathtt{f}) \in T$ and m' such that $\mathsf{m}\,[(\{t_1, t_2\}, \mathtt{f})\rangle\,\mathsf{m}'$, $\overline{\ell(t_1)} = \ell(t_2)$, $\mathtt{path}(m_i) < t_i$ for $i \in \{1, 2\}$ and $(R', \langle S, T, F, \mathsf{m}'\rangle) \in \mathcal{R}$;
(4) if $R \overset{m_1,m_2:\tau}{\rightsquigarrow} R'$ then there exist $(t_1, \mathtt{r}), (t_2, \mathtt{r}) \in T$ and m' such that $\mathsf{m}\,[(\{t_1, t_2\}, \mathtt{r})\rangle\,\mathsf{m}'$, $\overline{\ell(t_1)} = \ell(t_2)$, $\mathtt{path}(m_i) < t_i$ for $i \in \{1, 2\}$ and $(R', \langle S, T, F, \mathsf{m}'\rangle) \in \mathcal{R}$;
(5) if $\mathsf{m}\,[t\rangle\,\mathsf{m}'$ with $t = (\mathtt{path}(m)\alpha, \mathtt{f})$ then there exists R, R' such that $\mu(R) = \mathsf{m}$, $\mu(R') = \mathsf{m}'$, $R \xrightarrow{m:\alpha} R'$ and $(R', \langle S, T, F, \mathsf{m}'\rangle) \in \mathcal{R}$;
(6) if $\mathsf{m}\,[t\rangle\,\mathsf{m}'$ with $t = (\mathtt{path}(m)\alpha, \mathtt{r})$ then there exists R, R' such that $\mu(R) = \mathsf{m}$, $\mu(R') = \mathsf{m}'$, $R \overset{m:\alpha}{\rightsquigarrow} R'$ and $(R', \langle S, T, F, \mathsf{m}'\rangle) \in \mathcal{R}$;
(7) if $\mathsf{m}\,[(\{t_1, t_2\}, \mathtt{f})\rangle\,\mathsf{m}'$ with $\overline{\ell(t_1)} = \ell(t_2)$ and $\mathtt{path}(m_i)\alpha_i = t_i$ with $\ell(t_i) = \alpha_i$ for $i \in \{1, 2\}$ then there exists R, R' such that $\mu(R) = \mathsf{m}$, $\mu(R') = \mathsf{m}'$, $R \xrightarrow{m_1,m_2:\tau} R'$ and $(R', \langle S, T, F, \mathsf{m}'\rangle) \in \mathcal{R}$;
(8) if $\mathsf{m}\,[(\{t_1, t_2\}, \mathtt{r})\rangle\,\mathsf{m}'$ with $\overline{\ell(t_1)} = \ell(t_2)$ and $\mathtt{path}(m_i)\alpha_i = t_i$ with $\ell(t_i) = \alpha_i$ for $i \in \{1, 2\}$ then there exists R, R' such that $\mu(R) = \mathsf{m}$, $\mu(R') = \mathsf{m}'$, $R \overset{m_1,m_2:\tau}{\rightsquigarrow} R'$ and $(R', \langle S, T, F, \mathsf{m}'\rangle) \in \mathcal{R}$.

The largest forward reverse bisimulation is called forward reverse bisimilarity, denoted with $\sim_{FR}$.

We first prove that two coherent RCCS processes which are structurally congruent are encoded within the *same* rUN. Subsequently, we demonstrate the equivalence between a step taken in the process algebra and the firing of an appropriate transition in the corresponding network, and vice versa.

Lemma 4.17 (Preservation)**.** *Let R_1 and R_2 be two coherent RCCS processes. If $R_1 \equiv R_2$ then $\overleftarrow{\mathcal{N}(R_1)}$ and $\overleftarrow{\mathcal{N}(R_2)}$ are isomorphic and have the same marking up to places renaming.*

Proof. Since $\equiv$ is defined on monitored processes, then the only axiom which changes the structure of the ancestor process is the α-renaming. Hence R_1 and R_2 have the same ancestor, say P, up to α-renaming. It is easy to see that the two generated nets have the same places, transitions and flow relation up to renaming, hence they are isomorphic. We just have to check whether the initial markings are the same. The proof follows by induction and case analysis on the last applied axiom of $\equiv$:

Split: If the last applied rule is (Split), w.l.o.g. we can assume $R_1 = m \triangleright (P_1 \parallel P_2)$ and $R_2 = \langle 0\rangle \cdot m \triangleright P_1 \parallel \langle 1\rangle \cdot m \triangleright P_2$. We need to show that $\mu(R_1) = \mu(R_2)$. By looking at the definition of $\mu(\cdot)$ we have that

$$\mu(R_1) = \mathsf{m} \text{ and}$$

and

$$\mu(R_2) = \mu(\langle 0\rangle \cdot m \triangleright P_1) \bowtie \mu(\langle 1\rangle \cdot m \triangleright P_2)$$

Also, by definition of $\mu(\cdot)$ we have that $\mathsf{m} = \mathsf{m}_0 \cup \mathtt{path}(m)\mathsf{m}_{P_1\parallel P_2}$, where $\mathsf{m}_{P_1\parallel P_2}$ is the initial marking of the net encoding $(P_1 \parallel P_2)$. Hence, we can divide this marking into the marking of P_1 and the marking of P_2 as follows:

$$\mathsf{m} = \mathsf{m}_0 \cup \mathtt{path}(m)\|_0\mathsf{m}_{P_1} \cup \mathtt{path}(m)\|_1\mathsf{m}_{P_2}$$

Also, we have that:

$$\mu(\langle 0\rangle \cdot m \triangleright P_1) = \mathsf{m}_a \cup \mathtt{path}(\langle 0\rangle \cdot m)\mathsf{m}_{P_1} = \mathsf{m}_0^a \cup \mathtt{path}(m)\|_0\mathsf{m}_{P_1}$$
$$\mu(\langle 1\rangle \cdot m \triangleright P_2) = \mathsf{m}_b \cup \mathtt{path}(\langle 1\rangle \cdot m)\mathsf{m}_{P_2} = \mathsf{m}_0^b \cup \mathtt{path}(m)\|_1\mathsf{m}_{P_2}$$

Were $\mathsf{m}_a = \mathsf{m}_b = \mathsf{m}_0$, since it is the marking derived from the information contained into the memory m. Now, it is simple to conclude, since $\mathsf{m}_a \bowtie \mathsf{m}_b = \mathsf{m}_0 \bowtie \mathsf{m}_0 = \mathsf{m}_0$, as they are the same marking and since we are considering reachable processes it is impossible for a process to synchronise in the future with itself, hence $\bowtie$ acts as the normal set union.

Res: this case is a simplified version of the previous one.

α: Suppose $\mu(R_1) = \mathsf{m} \cup \mathsf{m}'$ where m' is the markings containing the bound action which will be converted by the last application of $\equiv$. By inductive hypothesis we also have that $\mu(R_2) = \mathsf{m} \cup \alpha(\mathsf{m}')$ where the α-conversion is applied only to those names which contains the bound action, that is m'. We have that the two nets have the same marking up to some renaming, as desired. □

Lemma 4.18 (Soundness)**.** *Let R_1 be an RCCS coherent process and $\overleftarrow{\mathcal{N}(R_1)} = \langle S, T, F, \mu(R_1)\rangle$ its corresponding* rUN*. If $R_1 \xrightarrow{\hat{m}:\alpha} R_2$ then*

- *$\overleftarrow{\mathcal{N}(R_2)} = \langle S, T, F, \mu(R_2)\rangle$; and*
- *there exists $t \in T$ such that $\mu(R_1)\,[t\rangle\, \mu(R_2)$; and*
- *for some $d \in \{\mathtt{f}, \mathtt{r}\}$ either*
  - *$\hat{m} = m$ and $t = (\mathtt{path}(m)\alpha, d)$; or*
  - *$\hat{m} = m_1, m_2$ and $\alpha = \tau$ and there exist two transitions $(t_1, d), (t_2, d) \in T$ with $\overline{\ell(t_1)} = \ell(t_2)$ and $\mathtt{path}(m_i)\alpha_i = t_i$ with $\ell(t_i) = \alpha_i$ for $i \in \{1, 2\}$, and $t = (\{t_1, t_2\}, d)$.*

Proof. As R_1 is a coherent process then it has an ancestor $\rho(R_1)$, say P which is unique (thanks to Lemma 4.9), which is the same ancestor of R_2, as R_2 is reached by R_1 with one reduction step. Therefore $\overleftarrow{\mathcal{N}(R_1)}$ and $\overleftarrow{\mathcal{N}(R_2)}$ have the same places, transitions and flow relation, the only difference being the marking. We show that for each move in the process algebra a corresponding firing of a transition $t \in T$ exists such that $(\mu(R_1) \setminus {}^\bullet t) \cup t^\bullet = \mu(R_2)$.

We have two cases: either the process synchronises with the context or it performs a τ (or a reversing of any of them). Both cases are similar, so we will focus on the first one. We proceed by induction on the derivation $R_1 \xrightarrow{m:\alpha} R_2$ with a case analysis on the last applied rule. The base cases correspond to the application of either R-ACT or R-ACT$^\bullet$.

R-ACT: Consider the application of the rule R-ACT. We have

$$R_1 = m \triangleright \sum_{i\in I} \alpha_i.Q_i \xrightarrow{m:\alpha_z} \langle *, \alpha_z^z, \sum_{i\in I\setminus\{z\}} \alpha_i.Q_i\rangle \cdot m \triangleright P_z = R_2$$

We first consider the case where $|I| = 1$. Hence we have

$$R_1 = m \triangleright \alpha.Q \xrightarrow{m:\alpha} \langle *, \alpha, \mathbf{0}\rangle \cdot m \triangleright Q = R_2$$

The marking corresponding to R_1 in the net $\overleftarrow{\mathcal{N}(P)} = \langle S, T, F, \mathsf{m}\rangle$ is $\mu(R_1) = \mu(m \triangleright \alpha.Q)$ and thanks to Corollary 4.13 the marking of R_2 is

$$\begin{aligned}\mu(R_2) &= \mu(\langle *, \alpha, \mathbf{0}\rangle \cdot m \triangleright Q)\\ &= \mu(m \triangleright \alpha.Q) \setminus \{\mathtt{path}(m)\alpha.Q\} \cup \{\mathtt{path}(m)\hat{\alpha}.\underline{\alpha}\} \cup \{\mathtt{path}(m)\hat{\alpha}.\mathsf{m}_Q\}\end{aligned}$$

By construction (see Definition 4.2), the net $\overleftarrow{\mathcal{N}(P)}$ contains a transition $t \in T$ such that $t = (\mathtt{path}(m)\alpha, \mathtt{f})$, with ${}^\bullet t = \{\mathtt{path}(m)\alpha.Q\}$ and $t^\bullet = \{\mathtt{path}(m)\hat{\alpha}.\underline{\alpha}\} \cup \{\mathtt{path}(m)\hat{\alpha}.\mathsf{m}_Q\}$. The thesis follows by observing that such transition is enabled at $\mu(R_1)$ because $\{\mathtt{path}(m)\alpha.Q\} \in \mu(R_1)$ by definition of $\mu(\cdot)$, and $\mu(R_1)\,[t\rangle\,\mu(R_2)$.

Consider now the case with $|I| > 1$.

$$R_1 = m \triangleright \sum_{i\in I} \alpha_i.Q_i \xrightarrow{m:\alpha_z} \langle *, \alpha_z^z, \sum_{i\in I\setminus\{z\}} \alpha_i.Q_i\rangle \cdot m \triangleright P_z = R_2$$

The marking corresponding to R_1 in the net $\overleftarrow{\mathcal{N}(P)} = \langle S, T, F, \mathsf{m}\rangle$ is

$$\begin{aligned}\mu(R_1) &= \mu(m \triangleright \sum_{i\in I} \alpha_i.Q_i)\\ &= \bigcup_{i\in I} \mu(m \triangleright \alpha_i.Q_i)\end{aligned}$$

and it contains the marked places $\{\mathtt{path}(m)+_{\mathsf{i}}\alpha_i.Q_i \mid i \in I\}$. Again by construction, the net $\overleftarrow{\mathcal{N}(P)}$ contains a transition $t \in T$ such that $t = (\mathtt{path}(m)+_{\mathsf{z}}\alpha_z, \mathtt{f})$, with $z \in I$, ${}^\bullet t = \{\mathtt{path}(m)+_{\mathsf{i}}\alpha_i.Q_i \mid i \in I\}$ and $t^\bullet = \{\mathtt{path}(m)+_{\mathsf{z}}\hat{\alpha}_z.\underline{\alpha_z}\} \cup \{\mathtt{path}(m)+_{\mathsf{z}}\hat{\alpha}_z.\mathsf{m}_{Q_z}\}$ and again $\mu(R_1)\,[t\rangle\,\mu(R_2)$ where $\mu(R_2)$ is the marking

$$\mu(m \triangleright \sum_{i\in I} \alpha_i.Q_i) \setminus \{\mathtt{path}(m)+_{\mathsf{i}}\alpha_i.Q_i \mid i \in I\} \cup \{\mathtt{path}(m)+_{\mathsf{z}}\hat{\alpha}_z.\underline{\alpha_z}\} \cup \{\mathtt{path}(m)+_{\mathsf{z}}\hat{\alpha}_z.\mathsf{m}_{Q_z}\}$$

R-ACT$^\bullet$: The case in which (R-ACT$^\bullet$) is used is similar. Assume

$$R_1 = \langle *, \alpha_z^z, \sum_{i\in I\setminus\{z\}} \alpha_i.Q_i\rangle \cdot m \triangleright Q_z \overset{m:\alpha_z}{\rightsquigarrow} m \triangleright \sum_{i\in I} \alpha_i.Q_i = R_2$$

and again take $|I|=1$.Then $\mu(R_1)=\mu(\langle *,\alpha,\mathbf{0}\rangle\cdot m\triangleright Q)=\mu(m\triangleright\alpha.Q)\setminus\{\mathtt{path}(m)\alpha.Q\}\cup\{\mathtt{path}(m)\hat{\alpha}.\underline{\alpha}\}\cup\{\mathtt{path}(m)\hat{\alpha}.\mathsf{m}_Q\}$. The transition $t=(\mathtt{path}(m)a,\mathtt{r})$ in $\overleftarrow{\mathcal{N}(P)}$ is enabled at $\mu(R_1)$ as it is the reverse of $(\mathtt{path}(m)\alpha,\mathtt{f})$ and its execution leads to the marking $\mu(R_2)=\mu(m\triangleright\alpha.Q)$ as required.

The case with $|I|>1$ follows the same argument of the forward one.

In the inductive case we have to do a case analysis on the last applied rule. We have (L-PAR), (R-SYCH), (R-RES) and (R-EQUIV) and their reversible variants. The most representative cases are (R-SYCH) and (R-EQUIV).

R-EQUIV: Consider the application of the rule (R-EQUIV). It follows by induction and by applying Lemma 4.17.

R-EQUIV$^\bullet$: The application of the rule (R-EQUIV$^\bullet$) follows the same argument of the previous case.

R-SYNCH: For the (R-SYCH) case, let us suppose $R_0=R_0^1\parallel R_0^2$. We have that $R_0^1\parallel R_0^2\xrightarrow{m_1,m_2:\tau}R^1_{1_{m_1@m_2}}\parallel R^2_{1_{m_2@m_1}}$ with $R_0^i\xrightarrow{m_i:\alpha_i}R_1^i$ and $\alpha_1=\overline{\alpha_2}$. By applying the inductive hypothesis on the derivations $R_0^i\xrightarrow{m_i:\alpha_i}R_1^i$ we have that there exists two transitions t_1 and t_2 such that $\mathsf{m}^{\mathsf{i}}_{\mathsf{r}_0}[t_i\rangle\,\mathsf{m}^{\mathsf{i}}_{\mathsf{r}_1}$, $(\mathtt{path}(m_i)\alpha_i,\mathtt{f})=t_i$, $\mu(R_0^i)=\mathsf{m}^{\mathsf{i}}_{\mathsf{r}_0}$ and $\mu(R_1^i)=\mathsf{m}^{\mathsf{i}}_{\mathsf{r}_1}$. We can desume that ${}^\bullet t_1\cap{}^\bullet t_2=\emptyset$, since they are enabled on different markings. Also, by definition we have that $\mathsf{m}_{\mathsf{r}_0}=\mu(R_0)=\mu(R_0^1)\bowtie\mu(R_0^2)$. Let us note that the operator $\bowtie$ acts on places which corresponds to past synchronisations, hence it does not affect ${}^\bullet t_i$, that is ${}^\bullet t_i\in\mathsf{m}_{\mathsf{r}_0}$. Since $\alpha_1=\overline{\alpha_2}$ then by Definition 4.3 in the net there exists a transition $t_\tau=(\{\mathtt{path}(m_1)\alpha_1,\mathtt{path}(m_2)\alpha_2\},\mathtt{f})$ where the preset and postset are respectively ${}^\bullet t_\tau={}^\bullet t_1\cup{}^\bullet t_2$ and $t_\tau{}^\bullet=(t_1{}^\bullet\setminus\{\mathtt{path}(m_1)\hat{\alpha}_1.\underline{\alpha_1}\})\cup(t_2{}^\bullet\setminus\{\mathtt{path}(m_2)\hat{\alpha}_2.\underline{\alpha_2}\})\cup\{s_{\{\mathtt{path}(m_1)\alpha_1,\mathtt{path}(m_2)\alpha_2\}}\}$. Hence we have that $\mathsf{m}_{\mathsf{r}_0}[t_\tau\rangle\,(\mathsf{m}_{\mathsf{r}}\setminus{}^\bullet t_\tau)\cup t_\tau{}^\bullet$. By definition we have that $\{\mathtt{path}(m_1)\alpha_1,m_2\}\in\mu(R_1^1)$ and $\{\mathtt{path}(m_2)\alpha_2,m_1\}\in\mu(R_1^2)$ and that $\{\mathtt{path}(m_1)\alpha_1,\mathtt{path}(m_2)\alpha_2\}\in\mu(R_1^1)\bowtie\mu(R_1^2)$. Also let us note that the $m_i@m_j$ operation just replace the $*$ on top of the memory m_i with m_j, which is similar to the $\bowtie$ operator. Hence $\mu(R_1^1)\bowtie\mu(R_1^2)=\mu(R^1_{1_{m_1@m_2}}\parallel R^2_{1_{m_2@m_1}})=\mathsf{m}_{\mathsf{r}_2}$, as desired.

R-SYCH$^\bullet$: this case is analogous to (R-ACT$^\bullet$). ☐

Lemma 4.19 (Completeness)**.** *Let R_1 be an RCCS coherent process and let $\overleftarrow{\mathcal{N}(R_1)}=\langle S,T,F,\mu(R_1)\rangle$ be the corresponding* rUN*. If $\mu(R_1)\,[t\rangle\,\mathsf{m}'$, then there exists R_2 s.t. one of the following holds:*

- $t=(\mathtt{path}(m)\alpha,d)$ *and* $R_1\xhookrightarrow{m:\alpha}R_2$ *and* $\overleftarrow{\mathcal{N}(R_2)}=\langle S,T,F,\mathsf{m}'\rangle$;
- $t=(\{t_1,t_2\},d)$ *such that* $\overline{\ell(t_1)}=\ell(t_2)$, *with* $t_i=(\mathtt{path}(m_i)\alpha_i,d)$, $\alpha_i\in\{\ell(t_1),\ell(t_2)\}$ *for* $i=1,2$ *and* $R_1\xhookrightarrow{m_1,m_2:\tau}R_2$ *with* $\overleftarrow{\mathcal{N}(R_2)}=\langle S,T,F,\mathsf{m}'\rangle$

with $d\in\{\mathtt{f},\mathtt{r}\}$.

Proof. If $\mu(R_1)\,[t\rangle\,\mathsf{m}'$, then $\mu(R_1)=\mathsf{m}_0\cup{}^\bullet t$ and $\mathsf{m}'=(\mathsf{m}_0\setminus{}^\bullet t)\cup t^\bullet$. The encoding of $\mathcal{N}(\cdot)$ is such that each transition or place name has a unique form, which corresponds to a path of a CCS term, and the transitions in $\overleftarrow{\mathcal{N}(\cdot)}$ are of the form (t,d), where t is the transition name of the CCS term and $d\in\{\mathtt{f},\mathtt{r}\}$ is the *direction*, either forward or reverse. That is from the transition name (t,d) we can isolate the RCCS term which can mimic the action.

If the transition (t,d) is not a synchronisation, that is (t,d) is not of the form $(\{t_1,t_2\},d)$, then we can assume w.l.o.g. that $t=(\tilde{\phi}\alpha,d)$ with $\tilde{\phi}$ being a sequence of $\phi\in\{\|_{\mathsf{i}},+_{\mathsf{i}},\hat{\alpha},\setminus_{\mathsf{a}}\}$.

Suppose d is $\mathtt{f}$. If the last decoration in $\tilde{\phi}$ has the form $+_{\mathsf{j}}$, that is $\tilde{\phi} = \tilde{\phi}'+_{\mathsf{j}}$ this means there exists in the net a set of transition $T' = \{t_i = (\tilde{\phi}\beta_i, \mathtt{f}) \mid (t_i, \mathtt{f}) \in T\}$. Now, let assume that the ancestor of R_1 is P, we have that $P = C[\sum_{i\in I} \beta_i.Q_i]$ where there exists an index $j \in I$ such that $\beta_j = \alpha$ and α is the action mimicked by the transition $(t, \mathtt{f})$ and the right position of the hole in the context is calculated using $\tilde{\phi}$. Also, since the transition is enabled in the net, then also $R_1 = E[(m \triangleright \sum_{i\in I} \beta_i.Q_i)\backslash A]$ where $E[\cdot]$ is an active context. Hence, we have that

$$E[(m \triangleright \sum_{i\in I} \beta_i.Q_i)\backslash A] \xrightarrow{m:\beta^j} E[(\langle *, \beta^j, \sum_{i\in I\backslash\{j\}} \rangle \cdot m \triangleright Q_j)\backslash A] = R_2$$

By definition 4.14 we have $\mathsf{m} = \mu(R_1)$, and by definition 4.4 ${}^{\bullet}t = \{\tilde{\phi}\beta_i.Q_i \mid i \in I\}$ and $t^{\bullet} = \{\tilde{\phi}\hat{\beta}_j.\underline{\beta_j}\} \cup \tilde{\phi}.\{\beta_j.Q_j\}$. Also

$$\mu(R_1) = \mu(E[\mathbf{0}]) \bowtie \mu((m \triangleright \sum_{i\in I} \beta_i.Q_i)\backslash A) = \mathsf{m} \cup \mathsf{m}_1$$
$$\mu(R_2) = \mu(E[\mathbf{0}]) \bowtie \mu((\langle *, \beta^j, \sum_{i\in I\backslash\{j\}} \rangle \cdot m \triangleright Q_j)\backslash A) = \mathsf{m} \cup \mathsf{m}_2$$

where m_1 and m_2 are the results of applying the eventual synchronisation $\bowtie$ respectively on $\mu((m \triangleright \sum_{i\in I} \beta_i.Q_i)\backslash A)$ and $\mu((\langle *, \beta^j, \sum_{i\in I\backslash\{j\}}\rangle \cdot m \triangleright Q_j)\backslash A)$. Moreover, we can separate from m_1 and m_2 the key places, that is the places whose name terminates with $\hat{\alpha}.\underline{\alpha}$ or with $s_{\{t_1,t_2\}}$. Be m_i^k such markings then we have:

$$\mu(R_1) = \mathsf{m} \cup \mathsf{m}_1 = \mathsf{m} \cup \mathsf{m}_1^k \cup \mathsf{m}_1'$$
$$\mu(R_2) = \mathsf{m} \cup \mathsf{m}_2 = \mathsf{m} \cup \mathsf{m}_2^k \cup \mathsf{m}_2'$$

By definition of $\mu(\cdot)$ we have that

$$\mathsf{m}_1' = \{\mathtt{path}(m).+_{\mathsf{i}}\beta_i.Q_i \mid i \in I\}$$
$$\mathsf{m}_2' = \{\mathtt{path}((\langle *, \beta^j, \sum_{i\in I\backslash\{j\}} \rangle \cdot m).\hat{\beta}_j.\underline{\beta_j}\} \cup \{\mathtt{path}((\langle *, \beta^j, \sum_{i\in I\backslash\{j\}} \rangle \cdot m).\hat{\beta}_j.Q_j\}$$

It is easy to check that $\tilde{\phi} = \mathtt{path}(m)+_{\mathsf{i}}$ and $\tilde{\phi} = \mathtt{path}((\langle *, \beta^j, \sum_{i\in I\backslash\{j\}}\rangle \cdot m)$. And we are done.

The cases of synchonisation and backward transitions are similar. ☐

We can now state our main result in terms of bisimulation:

Theorem 4.20. *Let R be an RCCS process and let $P = \rho(R)$ be its ancestor, then*

$$\langle\rangle \triangleright P \sim_{FR} \overleftarrow{\mathcal{N}(P)}$$

Proof. It is sufficient to show that

$$\mathcal{R} = \{(R, \langle S, T, F, \mu(R)\rangle) \mid \rho(R) = P,\ \overleftarrow{\mathcal{N}(P)} = \langle S, T, F, \mathsf{m}\rangle\}$$

is a forward and reverse bisimulation. It is easy to check that all the conditions of Definition 4.16 are matched by Lemmas 4.18 and 4.19. ☐

5. Conclusions and future works

On the line of previous research we have equipped a reversible process calculus with a non sequential semantics by using one of the classical encoding of process calculi into nets. What comes out from the encoding is that the machinery to reverse a process was already present in the encoding. Other approaches to address true concurrency in reversible calculi have been explored, for instance [Aub22, Aub24], where a proved semantics [DP92] for CCSK is given. This requires to revisit the LTS of CCSK in order to add extra information in the labels, about the process which contributed to an action, and then to derive a true-concurrent notion. Our approach directly compiles RCCS into a truly concurrent model, and hence we do not need to modify the lts of RCCS. Hence we exploit the natively truly concurrent semantics of Petri nets in order to retrieve a truly concurrent semantics of RCCS. Also our approach accounts for infinite behaviours, while [Aub22, Aub24] do not.

The current results applies to RCCS, but we do believe that the same encoding could be used to model CCSK processes. As a matter of fact, in CCSK the information is stored directly in the process and executed prefixes are marked with communications keys and in our encoding it is signalled by a token in *key*-places. For example if we take the process $P = a.Q$ in CCSK the process evolves in $a[i].Q$ where the forward behaviour of the process is Q while the backward behaviour is represented by the marked prefix $a[i]$. The same mechanisms applies to synchronisations. If we take the process $a.b.\mathbf{0} \parallel \overline{a}.\mathbf{0}$ the process can make a synchronisation followed by the b action and evolves to $a[i].b[j].\mathbf{0} \parallel \overline{a}[i].\mathbf{0}$. In this way, the synchronisation on a cannot be undone if first the action b is undone. By looking on how history information is kept into CCSK processes, it is clear that there is a tight correspondence between the marked prefixes, the key-places and $\hat{\cdot}$ decorations we have used in unravel nets. Also in CCSK the process structure does not change, and the marking of the reversible net would correspond to the marked prefixes. This seems to bring a more straightforward encoding of CCSK into Petri Nets, where the marking can be easily retrieved from a CCSK term. Having the two encodings into Petri Net would allow us for cross-fertilization results, in line with [LMM21]. The whole encoding and the machinery connected to it is left for future work.

Our result relies on unravel nets, that are able to represent *or*-causality. The consequence is that the same event may have different pasts. Unravel nets are naturally related to *bundle* event structures [Lan92, LBK97], where the dependencies are represented using *bundles*, namely finite subsets of conflicting events, and the bundle relation is usually written as $X \mapsto e$. Starting from an unravel net $\langle S, T, F, \mathsf{m}\rangle$, and considering the transition $t \in T$, the bundles representing the dependencies are ${}^{\bullet}s \mapsto t$ for each $s \in {}^{\bullet}t$, and the conflict relation can be easily inferred by the semantic one definable on the unravel net. This result relies on the fact that in any unravel net, for each place s, the transitions in ${}^{\bullet}s$ are pairwise conflicting. The *reversible* bundle structures add to the bundle relation (defined also on the reversing events) a prevention relation, and the intuition behind this relation is the usual one: some events, possibly depending on the one to be reversed, are still present and they *prevent* that event to be reversed. The problem here is that in an unravel net, differently from occurrence nets, is not so easy to determine which transitions depend on the happening of a specific one, thus potentially preventing it from being reversed. An idea would be to consider all the transitions in $s^{\bullet}$ for each $s \in t^{\bullet}$, but it has to be carefully checked if this is enough. Thus, which is the proper "reversible bundle event structure" corresponding to the reversible unravel nets has to be answered, though it is likely that the conditions to be posed on the prevention relations will be similar to the ones considered in [GPY18, GPY21]. Once that also this step is done, we will have the full correspondence between reversible processes calculi and non sequential models.

Another future works idea would be to move from reversible CCS to reversible π-calculus [LMS16, CKV13] by relying on the results of [BG09]. In [BG09] a truly concurrent semantics of π-calculus is given in form of Petri nets with inhibitor arcs. We could exploit our previous results on reversibility and Petri nets with inhibitor arcs [MMP21a, MMP23, MMP24] to obtain a truly concurrent semantics for reversible π-calculus in Petri nets with inhibitor arcs. Alternatively we could exploit the encoding of reversible π-calculus into rigid families (based on configuration structures), given in [CKV16], and bring it to Petri nets.

References

[A+20] Bogdan Aman et al. Foundations of reversible computation. In *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, volume 12070 of *LNCS*, pages 1–40. Springer, 2020. doi:10.1007/978-3-030-47361-7_1.

[AC20] Clément Aubert and Ioana Cristescu. How reversibility can solve traditional questions: The example of hereditary history-preserving bisimulation. In *31st International Conference on Concurrency Theory, CONCUR 2020*, volume 171 of *LIPIcs*, pages 7:1–7:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CONCUR.2020.7.

[Aub22] Clément Aubert. Concurrencies in reversible concurrent calculi. In *Reversible Computation - 14th International Conference, RC 2022, Urbino, Italy, July 5-6, 2022, Proceedings*, volume 13354 of *Lecture Notes in Computer Science*, pages 146–163. Springer, 2022. doi:10.1007/978-3-031-09005-9_10.

[Aub24] Clément Aubert. The correctness of concurrencies in (reversible) concurrent calculi. *J. Log. Algebraic Methods Program.*, 136:100924, 2024. doi:10.1016/J.JLAMP.2023.100924.

[BC94] Gérard Boudol and Ilaria Castellani. Flow models of distributed computations: Three equivalent semantics for CCS. *Information and Computation*, 114(2):247–314, 1994. doi:10.1006/inco.1994.1088.

[BDd22] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Open compliance in multiparty sessions. In *Formal Aspects of Component Software - 18th International Conference, FACS 2022, Virtual Event, November 10-11, 2022, Proceedings*, pages 222–243, 2022. doi:10.1007/978-3-031-20872-0_13.

[BG09] Nadia Busi and Roberto Gorrieri. Distributed semantics for the pi-calculus based on Petri nets with inhibitor arcs. *J. Log. Algebraic Methods Program.*, 78(3):138–162, 2009. doi:10.1016/j.jlap.2008.08.002.

[Bou90] Gérard Boudol. Flow event structures and flow nets. In *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings*, volume 469 of *LNCS*, pages 62–95. Springer, 1990. doi:10.1007/3-540-53479-2_4.

[CGP09] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(5):1–61, 2009.

[CKV13] Ioana Cristescu, Jean Krivine, and Daniele Varacca. A compositional semantics for the reversible p-calculus. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 388–397. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.45.

[CKV16] Ioana Cristescu, Jean Krivine, and Daniele Varacca. Rigid families for the reversible π-calculus. In *Reversible Computation - 8th International Conference, RC 2016*, volume 9720, pages 3–19. Springer, 2016. doi:10.1007/978-3-319-40578-0_1.

[Cou83] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, 1983. doi:10.1016/0304-3975(83)90059-2.

[DK04] Vincent Danos and Jean Krivine. Reversible communicating systems. In *CONCUR 2004 - Concurrency Theory*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004.

[DNM88] Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. A distributed operational semantics for CCS based on condition/event systems. *Acta Informatica*, 26(1/2):59–91, 1988. doi:10.1007/BF02915446.

[DP92] Pierpaolo Degano and Corrado Priami. Proved trees. In *Automata, Languages and Programming, 19th International Colloquium, ICALP92*, volume 623 of *Lecture Notes in Computer Science*, pages 629–640. Springer, 1992. doi:10.1007/3-540-55719-9_110.

[Gol90] Ursula Goltz. CCS and Petri Nets. In *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings*, volume 469 of *LNCS*, pages 334–357. Springer, 1990. doi:10.1007/3-540-53479-2_14.

[GPY18] Eva Graversen, Iain Phillips, and Nobuko Yoshida. Event structure semantics of (controlled) reversible CCS. In *Reversible Computation RC 2018*, volume 11106 of *LNCS*, pages 102–122. Springer, 2018. doi:10.1007/978-3-319-99498-7_7.

[GPY21] Eva Graversen, Iain Phillips, and Nobuko Yoshida. Event structure semantics of (controlled) reversible CCS. *Journal of Logical and Algebraic Methods in Programming*, 2021. doi:10.1016/j.jlamp.2021.100686.

[Kri13] Jean Krivine. A verification technique for reversible process algebra. In *Reversible Computation RC 2012. Revised Papers*, volume 7581 of *LNCS*. Springer, 2013. doi:10.1007/978-3-642-36315-3.

[Lan92] Rom Langerak. Bundle event structures: a non-interleaving semantics for LOTOS. In *Formal Description Techniques, V, Proceedings of the IFIP TC6/WG6.1 FORTE 92*, volume C-10 of *IFIP Transactions*, pages 331–346. North-Holland, 1992.

[LBK97] Rom Langerak, Ed Brinksma, and Joost-Pieter Katoen. Causal ambiguity and partial orders in event structures. In *CONCUR '97: Concurrency Theory*, volume 1243 of *LNCS*, pages 317–331. Springer, 1997. doi:10.1007/3-540-63141-0_22.

[LMM21] Ivan Lanese, Doriana Medic, and Claudio Antares Mezzina. Static versus dynamic reversibility in CCS. *Acta Informatica*, 58(1):1–34, 2021. doi:10.1007/s00236-019-00346-6.

[LMS16] Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversibility in the higher-order π-calculus. *Theor. Comput. Sci.*, 625:25–84, 2016. doi:10.1016/j.tcs.2016.02.019.

[M+20] Claudio Antares Mezzina et al. Software and reversible systems: A survey of recent activities. In *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, volume 12070 of *LNCS*, pages 41–59. Springer, 2020. doi:10.1007/978-3-030-47361-7_2.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980. doi:10.1007/3-540-10235-3.

[MMP+20] Hernán Melgratti, Claudio Antares Mezzina, Iain Phillips, G. Michele Pinna, and Irek Ulidowski. Reversible occurrence nets and causal reversible prime event structures. In *Reversible Computation RC 2020*, volume 12227 of *LNCS*, pages 35–53. Springer, 2020. doi:10.1007/978-3-030-52482-1_2.

[MMP21a] Hernán Melgratti, Claudio Antares Mezzina, and G. Michele Pinna. A distributed operational view of reversible prime event structures. In *Proceedings of the 36rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470623.

[MMP21b] Hernán Melgratti, Claudio Antares Mezzina, and G. Michele Pinna. Towards a truly concurrent semantics for reversible CCS. In *Reversible Computation - 13th International Conference, RC 2021*, volume 12805 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2021. doi:10.1007/978-3-030-79837-6_7.

[MMP23] Hernán Melgratti, Claudio Antares Mezzina, and G. Michele Pinna. Relating reversible Petri nets and reversible event structures, categorically. In *Formal Techniques for Distributed Objects, Components, and Systems - 43rd IFIP WG 6.1 International Conference, FORTE 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023*, volume 13910 of *Lecture Notes in Computer Science*, pages 206–223. Springer, 2023. doi:10.1007/978-3-031-35355-0_13.

[MMP24] Hernán Melgratti, Claudio Antares Mezzina, and G. Michele Pinna. A reversible perspective on petri nets and event structures. *ACM Trans. Comput. Logic*, aug 2024. Just Accepted. doi:10.1145/3686154.

[MMU20] Hernán Melgratti, Claudio Antares Mezzina, and Irek Ulidowski. Reversing place transition nets. *Log. Methods Comput. Sci.*, 16(4), 2020. URL: https://lmcs.episciences.org/6843.

[NPW81] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoretical Computer Science*, 13:85–108, 1981. doi:10.1016/0304-3975(81)90112-2.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[PU07] Iain Phillips and Irek Ulidowski. Reversing algebraic process calculi. *J. Log. Algebraic Methods Program.*, 73(1-2):70–96, 2007. doi:10.1016/j.jlap.2006.11.002.

[Rei85] Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985. doi:10.1007/978-3-642-69968-9.

[Rei99] Claus Reinke. Haskell-Coloured Petri Nets. In *Implementation of Functional Languages, 11th International Workshop, IFL'99*, volume 1868 of *Lecture Notes in Computer Science*, pages 165–180. Springer, 1999. doi:10.1007/10722298_10.

[Win86] Glynn Winskel. Event Structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 325–392. Springer, 1986. doi:10.1007/3-540-17906-2_31.

[Win88] Glynn Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, volume 354 of *Lecture Notes in Computer Science*, pages 364–397. Springer, 1988.

Appendix A. Implementation

We describe an effective implementation of the proposed encoding in Haskell[2]. The intent of this section is to provide a practical evidence that the coinductive approach to describe infinite behaviours is effective, rather than providing a fully fledged tool. We are aware there exist other Petri net implementations in Haskel (see for example [Rei99]), but a comparison with such tools is out of the scope of this section.

A.1. **Representation of infinite nets.** When working with an infinite data structure, a pivotal aspect is devising an efficient strategy to traverse the pertinent section of the structure. In our specific scenario, we prioritise the capability to identify and execute enabled transitions within a (potentially infinite) net. Therefore, our main objective is to identify those transitions that are enabled at a given marking. For this purpose, we adopt a representation of infinite nets that facilitates obtaining a truncated version of the net that contains all the enabled transitions in given marking. To maintain simplicity, we avoid explicitly representing the flow relation as a set of pairs. Instead, we associate each transition with its preset (input places) and postset (output places). Consequently, we rely on the following instrumental datatype to represent transitions.

```haskell
-- Each transition consists of a name, a preset and a postset
data Transition t s = Transition
  { trName :: t
  , trPre  :: [s]
  , trPost :: [s]
  }
```

The parameters `t` and `s` represent the types of the names of transitions and places, respectively. In this representation, a transition is defined by its name and two lists of places, corresponding to its pre and postset.

[2]The code can be accessed at https://github.com/hmelgra/reversible-ccs-as-nets.

Then, the datatype for nets is as follows:

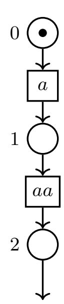
```
data Net s t = Net
  { netPlaces        :: [s] →  [s]
  , netTransitions   :: [s] →  [Transition t s]
  , netMarking       :: [s]
  }
```

The components of a net include a marking, denoted as `netMarking`, which is essentially a set of places. Additionally, there are two functions, `netPlaces` and `netTransitions`, which map every marking to a set of places and transitions, respectively, of a truncated, finite version of the net. This truncated net includes all the transitions from the potentially infinite net that are enabled in the given marking.

Example A.1. Consider the infinite net N depicted in Figure 16A. One potential Haskell definition for N could be `nAt [0]`, utilising the function `nAt` given in Figure 16B. This function takes a marking of type `[Int]` and returns a net with place names represented as integers and transitions as strings, i.e., of type `Net Int String`. The net's definition relies on the functions `p` and `t`, which determine the truncation of the net corresponding to a given marking. It is important to note that, for a specific marking `m`, any enabled transition t in `m` should satisfy the conditions ${}^{\bullet}t = \{i-1\}$ and $t^{\bullet} = \{i\}$, where $0 < i \leq m$, and m is the maximum integer in `m`. Therefore, the function `p`, which maps markings to sets of places, is defined as follows:

- For the empty marking, it returns an empty set of places since no transitions are enabled in the empty marking.
- For a non-empty marking `m`, it generates a list containing all integers in the range from 0 to the maximum value in `m` plus one.

Similarly, the function `t` creates a list of transitions, encompassing all those among the places in `p m`. These transitions are defined in such a way that `[i - 1]` represents its preset, and `[i]` represents its postset. The name of the i-th transition is denoted by i occurrences of `'a'`.



(A) N

```
nAt :: [Int] →  Net Int String
nAt = Net p t
  where
    p [] = []
    p m = [0 .. maximum m + 1]

    t [] = []
    t m  = [Transition (replicate i 'a') [i - 1] [i] |
            i ←  [1 .. maximum m + 1]]
```

(B) Haskell code

FIGURE 16. The Haskell representation of a simple infinite net N

The auxiliary functions `places` and `transitions`, defined below, allow us to respectively retrieve the set of places and transitions from the truncation of net for its marking. For instance, `places (nAt [0])` returns `[0,1]`, and `places (nAt [1,3])` gives `[0,1,2,3,4]`.

```
places :: Net s t → [s]
places n = netPlaces n $ netMarking n

transitions :: Net s t → [Transition t s ]
transitions n = netTransitions n $ netMarking n
```

Analogously, we rely on `isTransition :: Eq t ⇒ t → Net s t → Bool` to check that a given transition appears in the truncation of the net `n`. Then, the following predicate `isEnabled` allows to check if a given transition is enabled on a net.

```
isEnabled :: (Eq s, Eq t) ⇒ t → Net s t → Bool
isEnabled t n
  | isTransition t n  = all  (‘elem‘ netMarking n) (pre t n)
  | otherwise = False
```

The guard `isTransition t n` simply checks that `t` appears in the truncation of the net `n`. In such case, a transition `t` is enabled if all elements in its preset appear in the marking of the net. Otherwise, the transition is not enabled.

The firing of a transition straightforwardly changes the marking of the net as expected, i.e., by removing the preset of the transition and by adding the postset of the transition

```
fire :: (Eq s, Eq t) ⇒ t → Net s t → Net s t
fire t n@(Net ps ts m)
  | isEnabled t n = Net ps ts ((m  L.\\ pre t n) ++ post t n)
  | otherwise = error "transition not enabled"
```

Note that the firing generates an error if the transition is not enabled.

A.2. **Representing CCS processes.** The datatype for representing CCS actions is straightforwardly defined as follows:

```
data Action a
  = In a    -- Input
  | Out a   -- Output
```

Note that the datatype is parametric with respect to the type `a` of action names. The binary predicate `dual` (shown below) tests whether a two actions are dual, i.e., one is an input and the other is an output performed over the same channel.

```
-- It checks whether a pair of actions are duals
dual :: Eq a ⇒ Action a → Action a → Bool
dual (Out x) (In y) = x == y
dual (In x) (Out y) = x == y
```

The datatype for representing CCS processes is as follow.

```
{- Syntax for infinite CCS Processes -}
data CCS a
  = (Action a) :. (CCS a)   -- Prefix
  | (CCS a) :| (CCS a)      -- Parallel
  | (CCS a) :+ (CCS a)      -- Choice
  | (CCS a) :\ a            -- Restriction
  | Nil                     -- Ended process
```

```
  | Var VarName            -- Process variable
  | Rec VarName (CCS a)    -- Recursive process
```

The constructors are straightforward. For instance, `CCS Char` stands for the type of CCS processes whose channel names are characters. Then, the process $a.a \parallel \overline{a} + b$ in Figure 13 is defined as

```
ccs :: CCS Char
ccs = (In 'a' :. Out 'a' :. Nil) :| ((Out 'a' :. Nil) :+ (In 'b' :. Nil))
```

We highlight that the datatype `CCS` includes constructors for the finite definition of infinite processes, i.e., `Var` for a process variable and `Rec` for a recursive definition. This choice is down to the facts that (i) our encoding uses CCS processes as the names of the elements of the generated nets; and (ii) the operational semantics of nets is defined under the assumption that names can be effectively compared (see details below). In order to have an equality test for infinite terms, we opted for a finite representation. Hence, the infinite CCS process consisting of an infinite sequence of inputs over the channel a can be defined as follows

```
ccs' :: CCS Char
ccs' = Rec (VarName "X") (In 'a' :. Var (VarName "X") )
```

Process variables in CCS are now represented using the following type:

```
newtype VarName = VarName String deriving (Eq, Ord)
```

This new type aims to enhance the parsing of strings into `Ccs` instances by implementing the `Read` class (Details are omitted as they are non-essential for the translation process).

This new type has been introduced in order to facilitate the parsing of strings to CCS instances, by providing an instances of the class `Read`.

When dealing with the finite representation of infinite processes, we need the usual unfolding operation, which is defined in terms of the substitution of a process variable by a process. Substitution is given by the following function

```
subs :: CCS a     -- ^ process over which substitution is applied
     → String     -- ^ process variable to be substituted
     → CCS a      -- ^ replacement term
     → CCS a
```

whose defining equations are standard and therefore omitted.

The unfold function is as follows.

```
unfold :: CCS a → CCS a
unfold (Rec (VarName x) p) = subs p x (Rec (VarName x) p)
unfold p                   = p
```

The function `unfold` will be used in the definition of the encoding.

Despite we rely on the finite representation of CCS processes, we remark that the implementation of the encoding associates **infinite** nets to recursive CCS processes.

A.3. **Implementation of the encoding.** According to the encoding introduced in Section 4, the names of the places and transitions of the obtained nets are (possibly) decorated CCS processes. We rely on the following datatypes introducing constructors for the names of places and transitions.

```
{-- Place's names --}
data PlaceNames a
  = Proc (CCS a)                          -- CCS process
  | PKey (Action a)                       -- key for an action
  | PPref (Action a) (PlaceNames a)       -- prefixed by an executed action
  | PParLeft (PlaceNames a)               -- on the left of a parallel operator
  | PParRight (PlaceNames a)              -- on the right of a parallel operator
  | PSync (TransNames a) (TransNames a)   -- key for synchronisation
  | PPlusLeft (PlaceNames a)              -- on the left of a sum operator
  | PPlusRight (PlaceNames a)             -- on the right of sum operator
  | PRest (PlaceNames a) a                -- under restriction
  deriving (Eq, Ord)

{-- Transition's names --}
data TransNames a
  = Act (Action a)                        -- CCS process
  | TPref (Action a) (TransNames a)       -- prefixed by an executed action
  | TParLeft (TransNames a)               -- on the left of a parallel operator
  | TParRight (TransNames a)              -- on the right of a parallel operator
  | TSync (TransNames a) (TransNames a)   -- a synchronisation
  | TPlusLeft (TransNames a)              -- on the left of a sum operator
  | TPlusRight (TransNames a)             -- on the right of sum operator
  | TRest (TransNames a) a                -- under restriction
  deriving (Eq, Ord)
```

The above definitions are in one-to-one correspondence with the names introduced by the encoding of the previous Section, and self-explanatory.

We will use the predicate `isKey` on place's names that determines if a place name is a key, i.e., either `PKey` or `PSync` (its omitted definition is straightforward).

```
isKey :: PlaceNames a → Bool
```

The following function

```
label :: TransNames a → Action a
```

allows us to recover the label associated with a transition.

Then, the encoding function is given by

```
enc :: (Eq t) ⇒ CCS t → Net (PlaceNames t) (TransNames t)
```

We now illustrate some of its representative defining equations. According to Definition 4.1, the encoding of the process 0 (here represented by `Nil`) produces a net consisting of just one marked place. We name that place `Proc Nil`, i.e., the CCS process 0.

```
enc Nil = Net (const [Proc Nil]) (const []) [Proc Nil]
```

The fact that the net is defined in terms of the constant functions `const [Proc Nil]` and `const []` reflect that every finite truncation, independently from the given marking, consists of just one place `Proc Nil` and none transition. The marking `[Proc Nil]` assigns one token to the unique place.

The encoding of a prefixed process follows Definition 4.2. Hence, the encoding of $\alpha.P$ (written `a :.p` in the implementation) is built on top of the encoding of P, i.e., the names of the places and the transitions appearing in the encoding of P are decorated with the prefix $\hat{\alpha}$. We use `PPref a` for decorating a place name with the past of action `a` and similarly `TPref a` for a transition name. The following function (whose defining equations are omitted because are uninteresting) is in charge of applying renamings to a net.

```
rename :: (s -> s') -> (t -> t') -> (s' -> Maybe s) -> Net s t -> Net s' t'
```

The first and second parameter correspond respectively to the renaming of places and transitions. The third one is instrumental for mapping a marking on the decorated names to a marking of the encoding of P, which is needed for computing a truncation. Then, the equation for the encoding of `a :.p` is as follows.

```
1 enc (a :. p) = Net s t [Proc (a :. p)]
2   where
3     Net aSp aTp amp = rename (PPref a) (TPref a) (unwrapPref a) $ enc p
4     s m = if null m then [] else [Proc (a :. p), PKey a] ++ aSp m
5     t m = if null m then [] else
6             Transition (Act a) [Proc (a :. p)] (PKey a : amp) : aTp m
```

Note that line 3 introduces the net corresponding to the encoding of `p`, with its element suitable renamed. Then, the places and transitions of the (truncations of the) net are given by the defining equations of `s` and `t`. Besides the fact that they are empty for empty markings, their definitions mimic Definition 4.2. The encoding of `p` is extended with two places, one for the process (i.e., `Proc (a :. p)`) and one for the key (i.e, `PKey a`), and one transition of name `Act a`, whose preset is `Proc (a :. p)` and whose poset corresponds to the initial marking of the encoding of `p`, i.e., `amp`, and the new key `PKey a`.

As for the illustrated cases, the remaining equations follow the corresponding definitions in Section 4.

A.4. **Reversing nets.** Reversible nets, are implemented as nets with tagged transitions: the tag `Fwd` stands for forward transitions and `Bwd` are for reversing transitions. The corresponding data type is as follows.

```
data Directed a
  = Fwd a
  | Bwd a deriving (Eq, Ord)
```

Then, the following function `rev` takes a net and generates its reversible version.

```
rev :: Net s t -> Net s (Directed t)
rev (Net s t m) = Net s t' m
  where
    t' = foldr reverse [] . t

    reverse (Transition x y z) =
      (Transition (Fwd x) y z :) . (Transition (Bwd x) z y :)
```

Consider the network, denoted as `Net s t m`, which is translated into a new net with the same sets of places and markings, represented as `Net s t' m`. The set of transitions `t'` in the new net is obtained by applying the following transformations to each transition `Transition x y z` from the original set `t`:

- Add a forward transition, denoted as `Transition (Fwd x) y z`, to tag each transition in `t` as forward.
- Add the corresponding reversing transition, denoted as `Transition (Rev x) y z`, to maintain the bidirectional nature of the net.

A.5. **Simulation.** The concepts introduced in the previous sections can now be effectively utilised to simulate the behavior of reversible CCS processes. To illustrate this, let us consider the definition of the infinite CCS process `ccs` below.

```
ccs1 :: CCS Int
ccs1 = Rec (VarName "X") (In 1 :. Var (VarName "X"))

ccs2 :: CCS Int
ccs2 = Rec (VarName "X") ((Out 2 :. Var (VarName "X")) :+ (Out 1 :. Nil))

ccs :: CCS Int
ccs = (ccs1 :| ccs2) :\ 1
```

This process is defined as the parallel composition of two infinite processes, where the shared name `1` is restricted.

To obtain the corresponding reversible net, we apply the encoding followed by the reversing function, represented as `rev(enc ccs)`.

Using the functions that determine the enabled transitions of net and compute the firing of transitions, we can seamlessly implement a simulation function to replicate the behavior of the process.

```
simulate :: (Show s, Show t, Ord t, Eq s) ⇒ Net s t → IO ()
```

Then, the evaluation of

```
simulate $ rev (enc ccs)
```

shows the set of enabled transitions of the obtained net, which are as follows.

```
Enabled transitions:
1) → (|r:+l:2!)\1
2) → (|l:1?∗|r:+r:1!)\1
```

The name `(|r:+l:2!)\1` of the first transition indicates that it corresponds to the output performed on channel `2` by the left branch (i.e., `+l:`) of the right hand of the parallel composition (i.e., `|r:`). Similarly, the symbol ∗ in the name `|l:1?∗|r:+r:1!)\1` indicates that the transition corresponds to a synchronisation between the input performed on channel `1` by the left hand side of the parallel composition (i.e., `|l:`) and the output on channel `1` performed by the right branch of the right hand side of the parallel composition (i.e., `|r:+r:`).

At this point, any of the two transitions can be fired. After firing the first one, the obtained set of enabled transitions is the following.

```
Enabled transitions:
1) → (|r:+l:^2!.+l:2!)\1
2) → (|l:1?∗|r:+l:^2!.+r:1!)\1
3) ← (|r:+l:2!)\1
```

The first two transitions mirror the ones originally enabled; however, their names indicate that actions on the right-hand side of the parallel composition causally depend on the preceding performed action (the prefix +`l:^2!`).

In addition to these two forward transitions, there is one reversing transition that undoes the previously executed action.