On the Encodability of Reversible Process Calculi

Ivan Lanese
Olas Team, University of Bologna/INRIA, Italy

Claudio Antares Mezzina
University of Bari Aldo Moro, Italy

Iain Phillips
Imperial College London, UK

Irek Ulidowski
University of Leicester, UK
AGH University of Kraków, Poland

Shoji Yuen
Nagoya University, Japan

Abstract

Reversibility, allowing one to execute a program not only forwards as usual, but also backwards, has emerged as a fundamental concept in computing, with applications ranging from debugging and fault tolerance to biological and quantum systems. CCSK, a reversible extension of CCS, is a paradigmatic model of reversible concurrent computation. In this paper, we investigate the encodability of CCSK into classical forward-only concurrent models. We establish a separation theorem showing that there is no basic, success-sensitive encoding of CCSK into CCS or the π-calculus, highlighting the strong impact of reversibility on expressive power. We then present an encoding of CCSK processes with only top-level parallel composition into the internal π-calculus, correct up to strong bisimilarity. We also identify a fundamental limitation: no parallel-preserving encoding of CCSK (with arbitrary parallel composition) into the π-calculus can be correct up to strong bisimilarity. Finally, we provide a parallel-preserving encoding correct under a weaker behavioural correspondence: weak mutual simulation. Our findings extend the literature of encodability results to reversible process calculi.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Reversible computation, Process calculi, Encodings, Impossibility results

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2026.41

Related Version *Full Version with Proofs*: `https://arxiv.org/abs/2606.25916` [10]

Funding *Ivan Lanese*: partial support of the French ANR project SmartCloud ANR-23-CE25-0012.
Claudio Antares Mezzina: partial support of the MSCA SE project QCOMICAL (Grant Agreement ID: 101182520) and of the Japan Society for the Promotion of Science (JSPS) through the JSPS Invitation Fellowship for Research in Japan, grant no. S25016.
Irek Ulidowski: partial support of the AY2024 International PI Invitation Program, IAR Nagoya University, and of the Japan Society for the Promotion of Science (JSPS) through the JSPS Invitation Fellowship for Research in Japan, grant no. S21050.

Acknowledgements We thank the anonymous referees for their helpful comments and suggestions.

1 Introduction

Reversibility has emerged as a fundamental concept in the theory of computation. In concurrent systems, reversibility requires that every action can be undone in a causally consistent way [4], thereby restoring past states without breaking causal dependencies. This paradigm has proved useful in debugging [14, 5, 13], where reversing erroneous computations is more natural than replaying from scratch; in biochemical modelling [28, 15, 8], given

that many biochemical reactions are inherently reversible; in fault tolerance, where recovery requires controlled rollback to consistent states [33, 18]; and in quantum computing, where reversibility is forced by the laws of physics.

Several reversible extensions of process calculi have been proposed [4, 27, 11, 3]. Among them, CCSK [26, 27] enriches CCS with keys that record causal dependencies, supporting both forward and backward execution steps. Its operational semantics preserves the familiar flavour of CCS while enabling precise reversal of computations, including synchronisations.

A natural and fundamental question arises: *can reversibility be encoded into classical forward process calculi*?

The expressive power of forward process calculi has been extensively studied. For instance, Palamidessi showed that the synchronous π-calculus cannot be encoded into its asynchronous counterpart [20]. Gorla developed a uniform framework for separation and encodability results, identifying robust criteria for valid encodings [6]. More recently, Pugliese and Tiezzi introduced *replacement freeness* as a general criterion for proving separation results between calculi [29]. Further information on encodings can be found in a survey on the topic [21]. However, the role of reversibility in this landscape remains poorly understood. Can reversible calculi such as CCSK be faithfully encoded in forward ones like CCS or the π-calculus?

While broadly speaking the answer should be positive, as shown, e.g., by the encoding in [11], our deeper investigation shows a number of subtleties that need to be considered in order to define such encodings, and that encoding reversibility is quite demanding in terms of the requirements on, e.g., the target calculus.

In more detail, this paper gives the first systematic account of the expressive power of CCSK. Our contributions are as follows:

Separation theorem: We prove (Theorem 4.18) that there exists no basic, success-sensitive encoding of CCSK into CCS or the π-calculus (with recursion and without matching). The proof adapts techniques from Pugliese-Tiezzi [29] and Gorla [6], and relies only on a fragment of CCSK (nil, prefixing, and top-level parallel composition), showing that the result is robust. This establishes that reversibility in CCS is not eliminable: it requires expressive power which goes beyond that provided by classical CCS.

Encoding into π: We define (Figure 6) a parallel preserving encoding (Definition 5.8) of CCSK processes with top-level parallel composition only into the π-calculus with internal mobility [30]. The encoding exploits π names to represent CCSK keys and it is correct up to strong bisimilarity. This demonstrates how far CCSK can be captured within π, while making precise where the correspondence breaks down.

Limits of behavioural correspondence: We show (Theorem 5.10) that no encoding of CCSK into the π-calculus that is parallel preserving can be correct up to strong bisimilarity. This identifies a sharp boundary: top-level parallelism can be handled, but general parallelism exposes an inherent mismatch between reversible and forward concurrency.

Encoding general parallel composition: we refine (Definition 6.2) the encoding of CCSK into the π-calculus mentioned above to cope not only with top-level parallel composition, but also with parallel composition at lower levels. This requires the definition of a multiparty protocol to ensure that all descendants of a process have reversed before reversing it; hence the encoding is correct only up to *weak* mutual simulation.

Our results situate CCSK (and in general reversibility in CCS) in the landscape of encodability results between process calculi. They reveal that reversibility requires expressive power not available in classical (forward) CCS, for two main conceptual reasons: (i) the encoding cannot be compositional since the encoding of a term depends on its past, as it can go back to such past; (ii) the encoding of parallel composition in a calculus with binary synchronisation such

as CCS or the π-calculus requires a complex protocol to ensure all the children of a process are back to their initial state in order to enable backward execution of the parent process. These results are not dependent on features which are specific to CCSK, but on general aspects of reversibility; hence, similar results should hold for other reversible calculi as well. While we give some indications in this direction, we leave a more detailed analysis of this topic for future work.

Along with the theoretical interpretation of our results, the encoding techniques suggest systematic ways of translating causal histories into explicit control structures, which could inspire runtime mechanisms for reversible execution. In particular, using communication keys as explicit channels in the π-calculus highlights how one might represent rollback information as first-class messages, paving the way for implementations in distributed settings.

Structure of the paper. Sections 2 and 3 briefly recall CCSK and the π-calculus. Section 4 presents a separation result. In Section 5 we present an encoding of CCSK into the π-calculus with top-level parallel composition only, correct up to strong bisimilarity. Section 6 handles lower-level parallel composition, using the weaker notion of mutual simulation. The final section discusses related work and concludes the paper.

2 CCSK: syntax and operational semantics

In this section we recall the syntax and semantics of CCSK [26].

Let $\mathcal{N} = \{a, b, c, \ldots\}$ be a set of *names* (also called channels), and let $\overline{\mathcal{N}} = \{\overline{a} \mid a \in \mathcal{N}\}$ be the set of their corresponding *co-names*. The set of all *actions* is $\mathsf{Act} = \mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}$, where $\mathcal{N}$ and $\overline{\mathcal{N}}$ contain, respectively, input and output actions, and τ denotes the silent action. We let α range over the set $\mathcal{N} \cup \overline{\mathcal{N}}$, while μ ranges over the set Act. We say two prefixes such as a and $\overline{a}$ are *complementary*. The syntax of CCSK is given below. The set of CCSK terms is Proc, and we shall refer to terms as *processes*. We let P, Q and their primed or subscripted versions range over processes.

$$P \quad ::= \Sigma_I \rho_i.P_i \mid P \mid Q \mid \nu a.P \qquad \rho \quad ::= \mu \mid \mu[k]$$

We use n-ary guarded choice $\Sigma_I \rho_i.P_i$, indexed by a set I, to denote a process which may execute (or may already have executed, see below) any of the prefixes ρ_i and then behave as P_i. This will simplify our encoding while syntactically ensuring that choice is always guarded. We assume $\Sigma_I \rho_i.P_i = \mathbf{0}$ (the nil process, which does nothing) when $I = \emptyset$. We may write just $\rho.P$ when I is a singleton, and $\rho_1.P_1 + \Sigma_{I\setminus\{1\}} \rho_i.P_i$ to highlight a single branch, assuming guarded choice to be associative and commutative. Prefix ρ can be either μ (representing an action to be executed) or $\mu[k]$ (representing an action which has already been executed). Here k is a *key*, namely an identifier for the action execution. We denote with $\mathcal{K}$ the set of all keys. Synchronising actions have the same key, to ensure that they are undone together. This will become clearer when discussing the operational semantics. Process $P \mid Q$ is the parallel composition of P and Q, and name a is bound in the restriction $\nu a.P$.

▶ **Definition 2.1** (Process keys). *The set of keys of a CCSK process P, written $\mathsf{k}(P)$, is inductively defined as follows:*

$$\mathsf{k}(\mu.P) = \mathsf{k}(\mathbf{0}) = \emptyset \qquad \mathsf{k}(\mu[k].P) = \{k\} \cup \mathsf{k}(P) \qquad \mathsf{k}(P \mid Q) = \mathsf{k}(P) \cup \mathsf{k}(Q)$$
$$\mathsf{k}(\Sigma_I \rho_i.P_i) = \bigcup_I \mathsf{k}(\rho_i.P_i) \qquad \mathsf{k}(\nu a.P) = \mathsf{k}(P)$$

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 \qquad P =_\alpha Q \implies P \equiv Q$$

Figure 1 CCSK structural congruence rules.

Contexts C are obtained by extending with a hole $\bullet$ the syntax of processes. We consider contexts with a single occurrence of $\bullet$. We write $C[P]$ for the process obtained by replacing the $\bullet$ in C with process P.

We define $\equiv$ as the smallest congruence relation closed under the rules of Figure 1, where $P =_\alpha Q$ indicates two processes equivalent modulo α-conversion of bound names. The forward semantics of CCSK is given by the labelled transition system $(\mathsf{Proc}, \mathsf{Act} \times \mathcal{K}, \rightharpoonup)$, where $\rightharpoonup \subseteq \mathsf{Proc} \times (\mathsf{Act} \times \mathcal{K}) \times \mathsf{Proc}$ is the smallest relation closed under the forward rules of Figure 2, and under structural congruence. Correspondingly, the backward semantics

$$\text{(Act1)}\ \frac{\mathsf{std}(Q)}{\mu.Q \overset{\mu[k]}{\rightharpoonup} \mu[k].Q} \qquad \text{(Act1}^\bullet\text{)}\ \frac{\mathsf{std}(Q)}{\mu[k].Q \overset{\mu[k]}{\rightsquigarrow} \mu.Q}$$

$$\text{(Act2)}\ \frac{P \overset{\mu[h]}{\rightharpoonup} P' \qquad h \neq k}{\alpha[k].P \overset{\mu[h]}{\rightharpoonup} \alpha[k].P'} \qquad \text{(Act2}^\bullet\text{)}\ \frac{P' \overset{\mu[h]}{\rightsquigarrow} P \qquad h \neq k}{\alpha[k].P' \overset{\mu[h]}{\rightsquigarrow} \alpha[k].P}$$

$$\text{(Sum)}\ \frac{P_i \overset{\mu[k]}{\rightharpoonup} P_i' \quad \forall j \neq i.\ (P_j' = P_j \wedge \mathsf{std}(P_j))}{\Sigma_I P_i \overset{\mu[k]}{\rightharpoonup} \Sigma_I P_i'} \qquad \text{(Sum}^\bullet\text{)}\ \frac{P_i' \overset{\mu[k]}{\rightsquigarrow} P_i \quad \forall j \neq i.\ (P_j = P_j' \wedge \mathsf{std}(P_j'))}{\Sigma_I P_i' \overset{\mu[h]}{\rightsquigarrow} \Sigma_I P_i}$$

$$\text{(Par)}\ \frac{P \overset{\mu[k]}{\rightharpoonup} P' \qquad k \notin \mathsf{k}(Q)}{P \mid Q \overset{\mu[k]}{\rightharpoonup} P' \mid Q} \qquad \text{(Par}^\bullet\text{)}\ \frac{P' \overset{\mu[k]}{\rightsquigarrow} P \qquad k \notin \mathsf{k}(Q)}{P' \mid Q \overset{\mu[k]}{\rightsquigarrow} P \mid Q}$$

$$\text{(Syn)}\ \frac{P \overset{\alpha[k]}{\rightharpoonup} P' \qquad Q \overset{\overline{\alpha}[k]}{\rightharpoonup} Q'}{P \mid Q \overset{\tau[k]}{\rightharpoonup} P' \mid Q'} \qquad \text{(Syn}^\bullet\text{)}\ \frac{P' \overset{\alpha[k]}{\rightsquigarrow} P \qquad Q' \overset{\overline{\alpha}[k]}{\rightsquigarrow} Q}{P' \mid Q' \overset{\tau[k]}{\rightsquigarrow} P \mid Q}$$

$$\text{(Res)}\ \frac{P \overset{\mu[k]}{\rightharpoonup} P' \qquad \mu \notin \{a, \overline{a}\}}{\nu a.P \overset{\mu[k]}{\rightharpoonup} \nu a.P'} \qquad \text{(Res}^\bullet\text{)}\ \frac{P' \overset{\mu[k]}{\rightsquigarrow} P \qquad \mu \notin \{a, \overline{a}\}}{\nu a.P' \overset{\mu[k]}{\rightsquigarrow} \nu a.P}$$

$$\text{(Str)}\ \frac{P \equiv P' \quad P' \overset{\mu[k]}{\rightharpoonup} Q' \quad Q' \equiv Q}{P \overset{\mu[k]}{\rightharpoonup} Q} \qquad \text{(Str}^\bullet\text{)}\ \frac{P \equiv P' \quad P' \overset{\mu[k]}{\rightsquigarrow} Q' \quad Q' \equiv Q}{P \overset{\mu[k]}{\rightsquigarrow} Q}$$

Figure 2 CCSK labelled transition rules.

of a CCSK process is the smallest relation $\rightsquigarrow$ closed under the backward rules of Figure 2, which are symmetric to the forward ones. The semantics of CCSK is the union of the two relations, denoted by $\rightarrow$. Notably, each rule executing some action is paired with a dual rule undoing the same action. To this end, executed actions are not dropped from the process as in classical process calculi, but equipped with a key, thus denoting that they belong to the *past* of the process. This is visible, e.g., in rule (Act1). Due to this, execution can occur under executed prefixes, cf. rule (Act2). Rule (Syn) ensures that synchronising actions have the same key; thus rule (Syn$^\bullet$) is needed to ensure they are undone together. Note that rule (Par$^\bullet$) is not applicable in this case, due to the side condition $k \notin \mathsf{k}(Q)$.

▶ **Definition 2.2** (Guarded and top-level sub-processes). *Given a CCSK process P, a sub-process Q is* guarded*, or* lower level*, in P if it occurs within a summation $\sum_I \rho_i.P_i$; otherwise we say it is* top level*. Process P has only top-level parallel composition if every sub-process of the form $Q_1 \mid Q_2$ is top level.*

▶ **Definition 2.3** (Standard process). *A CCSK process P is* standard*, written $\mathsf{std}(P)$, if it contains no keys, that is $\mathsf{k}(P) = \emptyset$.*

Standard processes are CCS processes. Not all the processes generated by the grammar are meaningful. To cope with this issue we define what is a reachable process.

▶ **Definition 2.4** (Reachable process). *A CCSK process P is* reachable *if it can be derived from a standard process using the rules of Figure 2.*

From now on we will restrict attention to reachable processes.

Next we recall two definitions, which will come in handy once we define our encoding in Section 5.

▶ **Definition 2.5** (Free and bound keys). *A key k is* bound *in a process P iff it occurs either twice, attached to complementary prefixes, or once, attached to a τ prefix. A key k is* free *if it occurs once, attached to a non-τ prefix. We will indicate with $\mathsf{fk}(P)$ the set of free keys of P.*

Thanks to [12, Prop. 3.5] keys are either free or bound.

▶ **Definition 2.6.** *Let $\mathsf{toStd}(\cdot)$ be a forgetful map on CCSK processes defined as follows:*

$$\mathsf{toStd}(P) = P \text{ if } \mathsf{std}(P) \qquad \mathsf{toStd}(\mu[k].P) = \mu.\mathsf{toStd}(P)$$
$$\mathsf{toStd}(\Sigma_I \rho_i.P_i) = \Sigma_I \mathsf{toStd}(\rho_i.P_i) \qquad \mathsf{toStd}(P \mid Q) = \mathsf{toStd}(P) \mid \mathsf{toStd}(Q)$$
$$\mathsf{toStd}(\nu a.P) = \nu a.\mathsf{toStd}(P)$$

Intuitively, function $\mathsf{toStd}(\cdot)$ removes all keys from a process, hence undoing all its actions.

▶ **Example 2.7** (CCSK computation). Consider the CCSK process $a.(b.\mathbf{0} + c.\mathbf{0})$. Executing action a then b is represented by transitions

$$a.(b.\mathbf{0} + c.\mathbf{0}) \xrightarrow{a[k]} a[k].(b.\mathbf{0} + c.\mathbf{0}) \xrightarrow{b[h]} a[k].(b[h].\mathbf{0} + c.\mathbf{0}).$$

Executed actions are not dropped from the terms, as in CCS, but decorated with keys. Also, unused branches of the choice (here $c.\mathbf{0}$) are kept, to allow their future execution. After a, b have been done, we can undo the action $b[h]$ and then choose to explore c instead:

$$a[k].(b[h].\mathbf{0} + c.\mathbf{0}) \overset{b[h]}{\rightsquigarrow} a[k].(b.\mathbf{0} + c.\mathbf{0}) \xrightarrow{c[l]} a[k].(b.\mathbf{0} + c[l].\mathbf{0}) \quad ⌟$$

3 π-calculus: syntax and operational semantics

In this section we present the syntax and the early semantics of the π-calculus [31]. We let R, S and their primed versions range over processes and let Proc_π be the set of all processes, and $\mathcal{N}_\pi$ the set of all channel names. Also, sometimes we will refer to elements of $\mathcal{N}_\pi$ as names.

The syntax of π-calculus processes is reported below:

$$\begin{array}{lll} R & ::= \Sigma_I \pi_i.R_i \mid R \mid S \mid \nu a.R \mid \mathtt{rec}\ X.R \mid X \\ \pi & ::= a(x) \mid \overline{a}\langle b\rangle \mid \tau \end{array}$$

As for CCSK, we consider n-ary guarded choice $\Sigma_I \pi_i.R_i$, but sometimes also consider prefixes $\pi_i.R_i$ in isolation to allow for more modular definitions.

Bound and free names of a process R (written respectively $\mathrm{bn}(R)$ and $\mathrm{fn}(R)$) are defined by saying that name n is bound in $a(n).R$ and $\nu n.R$, other kinds of occurrences defining free names. We indicate with $\tilde{x}$ a non-empty sequence of names $x_1, \cdots, x_n$, and by abuse of notation we write $\nu\tilde{x}$, instead of $\nu x_1.\cdots.\nu x_n$.

The recursion operator $\mathtt{rec}\ X.R$ binds occurrences of recursion variable X in R. Every occurrence of X in R must be *guarded*, i.e. within a summation $\Sigma_I \pi_i.R_i$. Within a process every recursion variable must be bound.

$$\text{(In)}\ \ a(x).R \xrightarrow[\pi]{av} R\{v/x\} \qquad \text{(Out)}\ \ \overline{a}\langle v\rangle.R \xrightarrow[\pi]{\overline{a}\langle v\rangle} R \qquad \text{(Tau)}\ \ \tau.R \xrightarrow[\pi]{\tau} R$$

$$\text{(Sum)}\ \ \frac{j \in I \qquad R_j \xrightarrow[\pi]{\mu_j} R'_j}{\Sigma_I R_i \xrightarrow[\pi]{\mu_j} R'_j}$$

$$\text{(Par-L)}\ \ \frac{R \xrightarrow[\pi]{\mu} R' \qquad \mathrm{bn}(\mu) \cap \mathrm{fn}(S) = \emptyset}{R \mid S \xrightarrow[\pi]{\mu} R' \mid S} \qquad \text{(Com-L)}\ \ \frac{R \xrightarrow[\pi]{\overline{a}\langle v\rangle} R' \qquad S \xrightarrow[\pi]{av} S'}{R \mid S \xrightarrow[\pi]{\tau} R' \mid S'} \qquad \text{(Res)}\ \ \frac{R \xrightarrow[\pi]{\mu} R' \qquad a \notin \mathrm{n}(\mu)}{\nu a.R \xrightarrow[\pi]{\mu} \nu a.R'}$$

$$\text{(Open)}\ \ \frac{R \xrightarrow[\pi]{\overline{a}\langle b\rangle} R'}{\nu b.R \xrightarrow[\pi]{\overline{a}(b)} R'} \qquad \text{(Close-L)}\ \ \frac{R \xrightarrow[\pi]{\overline{a}(b)} R' \qquad S \xrightarrow[\pi]{ab} S' \qquad b \notin \mathrm{fn}(S)}{R \mid S \xrightarrow[\pi]{\tau} \nu b.(R' \mid S')} \qquad \text{(Rec)}\ \ \frac{R\{\mathtt{rec}\ X.R/X\} \xrightarrow[\pi]{\mu} R'}{\mathtt{rec}\ X.R \xrightarrow[\pi]{\mu} R'}$$

$$\text{(Str)}\ \ \frac{R \equiv_\pi R' \qquad R' \xrightarrow[\pi]{\mu} S' \qquad S' \equiv_\pi S}{R \xrightarrow[\pi]{\mu} S}$$

Figure 3 π-calculus early labelled transition system.

The semantics of the π-calculus is given by the labelled transition system $(\mathsf{Proc}_\pi, \mathsf{Act}_\pi, \xrightarrow[\pi]{})$, where $\xrightarrow[\pi]{} \subseteq \mathsf{Proc}_\pi \times \mathsf{Act}_\pi \times \mathsf{Proc}_\pi$ is the smallest transition relation closed under the rules of Figure 3. The set of actions Act_π is generated by the following grammar:

$$\mu \quad ::= ab \mid \overline{a}\langle b\rangle \mid \overline{a}(b) \mid \tau$$

In the grammar, ab represents an input with subject a and object b; $\overline{a}\langle b\rangle$ represents an output with subject a and object b; $\overline{a}(b)$ is a shorthand for $\nu b\,\overline{a}\langle b\rangle$ and represents the output

$$R =_\alpha S \implies R \equiv_\pi S$$

$$R \mid \mathbf{0} \equiv_\pi R \qquad R \mid S \equiv_\pi S \mid R \qquad R \mid (S \mid T) \equiv_\pi (R \mid S) \mid T$$

$$R + \mathbf{0} \equiv_\pi R \qquad R + S \equiv_\pi S + R \qquad R + (S + T) \equiv_\pi (R + S) + T$$

$$\nu a.\mathbf{0} \equiv_\pi \mathbf{0} \qquad \nu a.\nu b.R \equiv_\pi \nu b.\nu a.R \qquad \nu a.(R \mid S) \equiv_\pi R \mid \nu a.S \text{ if } a \notin \mathrm{fn}(P)$$

Figure 4 π-calculus structural congruence.

of a bound name; and τ denotes an internal step. The bound and free names of an action μ, written respectively $\mathrm{fn}(\mu)$ and $\mathrm{bn}(\mu)$, are defined by saying that the name b is bound in $\overline{a}(b)$, while other occurrences of names are free. We also set $\mathrm{n}(\mu) = \mathrm{fn}(\mu) \cup \mathrm{bn}(\mu)$.

We define $\equiv_\pi$ as the smallest congruence relation closed under the rules of Figure 4, where $R =_\alpha S$ indicates two processes equivalent modulo α-conversion. Relation $\underset{\pi}{\Rightarrow}$ is the transitive and reflexive closure of $\underset{\pi}{\xrightarrow{\tau}}$, and relation $\underset{\pi}{\overset{\mu}{\Rightarrow}}$ is defined as $\underset{\pi}{\Rightarrow}\underset{\pi}{\xrightarrow{\mu}}\underset{\pi}{\Rightarrow}$.

In our development, we will focus on a subcalculus of the π-calculus, called the internal π-calculus [30]. In the internal π-calculus, only *new* names can be sent as outputs. In other words, output prefix $\overline{a}\langle b\rangle$ can only occur immediately inside a restriction name b, namely as a term $\nu b.\overline{a}\langle b\rangle$. This implies that each synchronisation exchanges a different fresh name. As we will see, our encoding will use the internal π-calculus of order two, where communicated names, beyond being bound, can then only be used for synchronisation (like CCS names), and not for sending further names (cf. [30, Definition 6.2]). In this case we will drop the object from the prefix or the label.

4 Separation result

We give conditions under which CCSK cannot be encoded into CCS or the π-calculus, adapting definitions and results from [29, 6].

We start by recalling the definitions of replacement freeness and basic encodings from [29]. These are defined for general process calculi which have notions of reduction and barbs, which we shall instantiate for CCSK, CCS and the π-calculus.

▶ **Definition 4.1.** *A* (process) calculus $\mathbb{C}$ *has a set of processes, ranged over by* $P, \ldots$. *These are generated using process operators, giving rise to contexts as usual. Processes use* names *as communication channels and possibly for input parameters and output values. Name-binding operators delimit the scope of names, and names may be either free or bound. A process with no free names is* closed. *Calculus* $\mathbb{C}$ *also has a notion of* reduction $P \to P'$, *and* barb $P \downarrow \alpha$, *meaning that we can observe barb* α *at* P. *Weak reduction* $\Rightarrow$ *is the reflexive and transitive closure of* $\to$, *and* $P \Downarrow \alpha$ *means that there is* P' *such that* $P \Rightarrow P' \downarrow \alpha$.

▶ **Definition 4.2** (Visibility [29])**.** *A process* P *is* visible, *written* $P \Downarrow$, *if* $P \Downarrow \alpha$ *for some* α; *otherwise* P *is* invisible.

▶ **Definition 4.3** (Replacement freeness [29])**.** *A calculus* $\mathbb{C}$ *is* strongly replacement free (strongly RF) *if for every single-hole context* C, *invisible process* I *and process* P *in* $\mathbb{C}$, *we have* $C[I] \Downarrow$ *implies* $C[P] \Downarrow$. *A calculus* $\mathbb{C}$ *is* replacement free (RF) *if for every single-hole context* C, closed *invisible process* I *and process* P *in* $\mathbb{C}$, *we have* $C[I] \Downarrow$ *implies* $C[P] \Downarrow$. *Furthermore,* $\mathbb{C}$ *is* weakly RF *if it is RF but not strongly RF.*

▶ **Definition 4.4** (Basic encoding [29, Def. 3.1]). *An encoding* $[\![\cdot]\!]$ *of* $\mathbb{C}_1$ *into* $\mathbb{C}_2$ *is* basic *if:*

1. $[\![\cdot]\!]$ *is* compositional*: for every* k*-ary operator* op *in* $\mathbb{C}_1$ *there is a* k*-hole context* C_{op} *in* $\mathbb{C}_2$ *such that* $\forall P_1, \ldots, P_k \in \mathbb{C}_1$ *we have* $[\![\mathsf{op}(P_1, \ldots, P_k)]\!] = C_{\mathsf{op}}[\,[\![P_1]\!], \ldots, [\![P_k]\!]\,]$.
2. $[\![\cdot]\!]$ *is* interaction sensitive*: for all processes* P *in* $\mathbb{C}_1$ *we have* $P \Downarrow$ *iff* $[\![P]\!] \Downarrow$.

If an encoding is compositional then for any $\mathbb{C}_1$ context $C_1[\bullet]$ there is a $\mathbb{C}_2$ context $C_2[\bullet]$ such that for all $\mathbb{C}_1$ processes P we have $[\![C_1[P]]\!] = C_2[\,[\![P]\!]\,]$ [29, Lemma 3.1].

▶ **Proposition 4.5** ([29, Thm. 3.1]). *There exists no basic encoding from a non-strongly RF calculus to a strongly RF one.* ◀

Proposition 4.5 is used in [29] to show that the π-calculus extended with polyadic synchronisation and/or matching cannot be encoded into the basic π-calculus.

We now instantiate reduction and barbs for CCS, the π-calculus and CCSK.

▶ **Definition 4.6** (Reduction and barbs for CCS). *Let* $P \to P'$ *iff* $P \xrightarrow{\tau} P'$. *Strong barb:* $P \downarrow a$ *if* $P \xrightarrow{a} P'$ *for some* P'. $P \downarrow \overline{a}$ *if* $P \xrightarrow{\overline{a}} P'$ *for some* P'.

▶ **Definition 4.7** (Reduction and barbs for the π-calculus). *Let* $R \to R'$ *iff* $R \xrightarrow[\pi]{\tau} R'$. *Strong barb:* $R \downarrow a$ *if* R *can perform an input transition with subject* a*;* $R \downarrow \overline{a}$ *if* R *can perform an output transition with subject* a.

In CCSK we allow both forward and backward barbs.

▶ **Definition 4.8** (Reduction and barbs for CCSK). *(cf. [12]) Let* $P \to P'$ *iff either* $P \overset{\tau[m]}{\rightharpoonup} P'$ *or* $P \overset{\tau[m]}{\rightharpoondown} P'$*, for some key* m. *Strong barb:* $P \downarrow \mu$ *if* $P \overset{\mu[m]}{\rightharpoonup} P'$ *for* $\mu \neq \tau$ *and some* m *and* P'. $P \downarrow \underline{\mu}$ *if* $P \overset{\mu[m]}{\rightharpoondown} P'$ *for* $\mu \neq \tau$ *and some* m *and* P'.

The next lemma is instrumental in proving that CCS and the π-calculus are strongly RF.

▶ **Lemma 4.9.** *In CCS or the* π*-calculus, let* $C[\bullet]$ *be a context, let* I *be invisible and let* P *be any process, and let* α *be any barb (so that* $\alpha = a$ *or* $\alpha = \overline{a}$ *for some name* a*). If* $C[I] \Downarrow \alpha$ *then* $C[P] \Downarrow \alpha$.

▶ **Proposition 4.10** ([29]). *CCS and the* π*-calculus are strongly RF.*

The analogue of Lemma 4.9 does not hold for CCSK, as the following example shows.

▶ **Example 4.11.** Let $C[\bullet] = a[m].\bullet$. Let $I = \mathbf{0}$ and $P = b[n]$. Then $C[I] \Downarrow \underline{a}$ but $C[P] \Downarrow \underline{b}$ and not $C[P] \Downarrow \underline{a}$. ⌟

Nevertheless, we can show that CCSK is strongly RF using different methods.

▶ **Proposition 4.12.** *CCSK is strongly RF.*

Since CCSK, CCS and the π-calculus are all strongly RF, we need to adapt the definitions of replacement freeness and basic encodings to obtain an impossibility result for encoding CCSK into the π-calculus.

We therefore modify Definition 4.3 by considering success rather than visibility. We suppose ✓ be a particular barb that can be used to record the success of a computation. In CCSK, ✓ belongs to $\mathcal{N}$, the set of names, and is a forward barb; similarly for CCS. In the π-calculus, ✓ belongs to $\mathcal{N}_\pi$, the set of channel names, and is an input barb.

▶ **Definition 4.13** (Success replacement freeness). *A calculus* $\mathbb{C}$ *is* strongly success replacement free (strongly SuRF) *if for every single-hole context* C*, invisible process* I *and process* P *in* $\mathbb{C}$*, we have* $C[I] \Downarrow \checkmark$ *implies* $C[P] \Downarrow \checkmark$*. A calculus* $\mathbb{C}$ *is* success replacement free (SuRF) *if for every single-hole context* C*,* closed *invisible process* I *and process* P *in* $\mathbb{C}$*, we have* $C[I] \Downarrow \checkmark$ *implies* $C[P] \Downarrow \checkmark$*. Furthermore,* $\mathbb{C}$ *is* weakly SuRF *if it is SuRF but not strongly SuRF.*

▶ **Definition 4.14** ([6, Property 5]). *An encoding* $[\![\cdot]\!]$ *of* $\mathbb{C}_1$ *into* $\mathbb{C}_2$ *is* success sensitive *if for all processes* P *in* $\mathbb{C}_1$ *we have* $P \Downarrow \checkmark$ *iff* $[\![P]\!] \Downarrow \checkmark$*.*

We can obtain an analogue of Proposition 4.5:

▶ **Proposition 4.15.** *There exists no basic, success-sensitive encoding from a non-strongly SuRF calculus to a strongly SuRF one.*

▶ **Proposition 4.16.** *CCS and the* π*-calculus are strongly SuRF.*

▶ **Proposition 4.17.** *CCSK is not SuRF.*

Proof. We consider the context $C[\bullet] = (a[m].\bullet \mid \bar{a}[m].\mathbf{0}) \mid a.\checkmark.\mathbf{0}$, closed invisible process $I = \mathbf{0}$ and process $P = b[n].\mathbf{0}$. Then $C[I] \xrightarrow{\tau[m]} (a.\mathbf{0} \mid \bar{a}.\mathbf{0}) \mid a.\checkmark.\mathbf{0} \xrightarrow{\tau[k]} (a.\mathbf{0} \mid \bar{a}[k].\mathbf{0}) \mid a[k].\checkmark.\mathbf{0} \downarrow \checkmark$, so that $C[I] \Downarrow \checkmark$. However $C[P]$ cannot perform any reduction, so that $C[P] \Downarrow \checkmark$ fails to hold. ◀

The proof of Proposition 4.17 uses a limited fragment of CCSK: just $\mathbf{0}$, prefixing and top-level parallel composition. Actually, if we allow backward barbs as $\checkmark$, then we could use just $C[\bullet] = \checkmark[m].\bullet$ (cf. Example 4.11). Alternatively, with forward barbs and τ prefixes we could use $C[\bullet] = \checkmark.\mathbf{0} + \tau[m].\bullet$, featuring choice instead of parallel composition. This shows that the result is robust. Indeed, analogous results can be proved for many reversible calculi, obviously including calculi that can embed CCSK such as revTPL [1], but also others such as reversible broadcast CCS [17]. This shows that this separation result is more a feature of the reversibility mechanism (indeed, all examples above rely on backward actions) than of the details of the calculus. As a consequence, also the main result below is robust.

▶ **Theorem 4.18.** *There is no basic, success-sensitive encoding from CCSK to CCS or the* π*-calculus.*

Considering the counterexample in the proof of Proposition 4.17, we can rephrase Theorem 4.18 to state that there is no basic, success-sensitive encoding from CCSK with only top-level parallel composition to CCS or the π-calculus. We shall present an encoding from CCSK with top-level parallel composition to the π-calculus (see Figure 6). It must therefore fail to satisfy at least one of compositionality, interaction sensitivity and success sensitivity. Indeed, it is not compositional but it does satisfy the other two properties.

▶ Remark 4.19. Since the counterexample in the proof of Proposition 4.17 does not use summation, we can revise the definition of compositionality and of basic encoding to make no requirement on summation, and show that Theorem 4.18 still holds with this weaker notion of basic encoding.

See Figure 5 for a summary of the classification of calculi according to RF and SuRF. CPG is CCS with Priority Guards [25], which is shown to be not RF in [29, Prop. 6.2].

strongly RF	not RF
CCS, π, CCSK	CPG

strongly SuRF	not SuRF
CCS, π	CCSK

Figure 5 Classification of calculi according to RF and SuRF.

5 Encoding CCSK into the π-calculus

The previous section shows that there is no basic, success-sensitive encoding of CCSK into CCS or the π-calculus. Now we present an encoding of a subset of CCSK (with only top-level parallel), into the π-calculus. We show that only a subset of the π-calculus is needed, namely *internal* π [30]. We then prove that this encoding is correct up to strong bisimilarity.

CCSK processes use both names and keys. Both will be encoded as π-calculus names. We will call *channel names* the encoding of the former and *key names* the encoding of the latter. We assume the two sets of π names to be disjoint. In CCSK, keys are created in forward computation and consumed to manage backward computation. Hence, in the π-calculus, images of CCSK forward computations will create new key names (via the ν operator) and images of CCSK backward computations will use these key names to go back to past states.

The operational semantics of CCSK guarantees that if a key occurs in a reachable process then it has one or two occurrences; see [12, Prop. 3.5]. Keys are either free or bound, depending on which prefixes they are attached to (Definition 2.5). Free and bound keys will be modelled by free and bound key names, respectively, in the π-calculus.

To track which key names are used to model which keys, the encoding is parametric on a bijection ϕ, which records the correspondence between free keys and free key names. In CCSK each key is always bound to a single name (either as an input action a, or an output action $\overline{a}$, or both); hence ϕ can be written as a set of items of the form $(a, k) \leftrightarrow x$, where a, k are a CCSK name and a key, respectively, and x is a π-calculus key name. We assume the usual operations on bijections: *extension* $\phi[(a, k) \leftrightarrow x]$, meaning that ϕ is extended with $(a, k) \leftrightarrow x$ (assuming that (a, k) is not in the domain of ϕ); and *restriction* $\phi \setminus (a, k)$, meaning that (a, k) is removed from the domain of ϕ. The correspondence only tracks free keys, since bound keys correspond to bound names and hence are α-convertible.

▶ **Definition 5.1** (CCSK to π encoding). *The encoding function* $[\![\cdot]\!] : \mathsf{Proc} \to \mathsf{Proc}_\pi$ *is defined by* $[\![P]\!] = \nu K.[\![P, \mathbf{0}, \phi]\!]$ *where* K *is the set of key names for bound keys in* P*, and* $[\![P, \mathbf{0}, \phi]\!]$ *is defined in Figure 6, where* ϕ *is omitted since it is fixed.*

Note that if we encode a standard process then the set of bound keys is empty. The second parameter of the encoding in Figure 6 is a π-calculus process (ranged over by R), and is computed by the encoding itself. It is used to build the previous state of the π-calculus process which is the encoding of a CCSK process P. We will often refer to this parameter as a *backtrack process*.

We now explain the encoding rules in Figure 6. At the top-level, the only place in which parallel composition is allowed, the backtrack process R is $\mathbf{0}$. The parallel and restriction operators are encoded homomorphically. A process $\Sigma_i \mu_i.P_i$ represents an n-ary choice among n prefixed processes. Since CCSK is a reversible process calculus, we can get back to the original process from the executed branch. We enable backward execution via the recursion variable X, which gives access to the state before the choice is taken. Variable X is fresh and is passed to all the branches of the encoding of the choice. In this way, every branch has enough information to get back to the previous state X. Hence, the encoding of the n-ary choice is rendered as an $(n+1)$-ary choice in which the first branch is the backtrack process R. This idea is illustrated in Example 5.2 below.

$$
\begin{aligned}
[\![P \mid Q, \mathbf{0}]\!] &= [\![P, \mathbf{0}]\!] \mid [\![Q, \mathbf{0}]\!] \\
[\![(\nu a)P, R]\!] &= \nu a.[\![P, R]\!] \\
[\![\Sigma_i \mu_i.P_i, R]\!] &= \mathtt{rec}\ X.(R + \Sigma_i [\![\mu_i.P_i, X]\!]) && X \text{ fresh} \\
[\![\overline{a}[k].P + \Sigma_i \mu_i.P_i, R]\!] &= [\![P, \overline{x_k}.[\![\overline{a}.\mathsf{toStd}(P) + \Sigma_i \mu_i.P_i, R]\!]]\!] && \text{with } (a,k) \leftrightarrow x_k \in \phi \\
[\![a[k].P + \Sigma_i \mu_i.P_i, R]\!] &= [\![P, x_k.[\![a.\mathsf{toStd}(P) + \Sigma_i \mu_i.P_i, R]\!]]\!] && \text{with } (a,k) \leftrightarrow x_k \in \phi \\
[\![\tau[k].P + \Sigma_i \mu_i.P_i, R]\!] &= [\![P, \tau.[\![\tau.\mathsf{toStd}(P) + \Sigma_i \mu_i.P_i, R]\!]]\!]
\end{aligned}
$$

$$
[\![\overline{a}.P, X]\!] = \overline{a}(y).[\![P, \overline{y}.X]\!] \qquad [\![a.P, X]\!] = a(y).[\![P, y.X]\!] \qquad [\![\tau.P, X]\!] = \tau.[\![P, \tau.X]\!]
$$

Figure 6 Encoding of CCSK into the internal π-calculus.

A process $\overline{a}[k].P + \Sigma_i \mu_i.P_i$ represents a choice process, where the first branch $\overline{a}[k].P$ is being executed (and $\Sigma_i \mu_i.P_i$ is unused). Hence, the encoding proceeds to encode the prefix continuation P. Meanwhile, it builds the parameter R in such a way that once the action $\overline{a}[k]$ is reversed via the prefix $\overline{x_k}$, the process evolves to the encoding of the entire choice process $a.\mathsf{toStd}(P) + \Sigma_i \mu_i.P_i$. Since the process P may have executed actions with keys, we use the erasing function $\mathsf{toStd}(\cdot)$ to remove them, obtaining a standard version of P.

To simplify the definition of the encoding, and make it more modular, we define it on single prefixes (last line of the encoding) and then use this definition to specify the encoding of choice. The encoding of a prefix $\overline{a}.P$ is rendered as an output prefix which sends on the channel name a a new key name y, and continues as the encoding of its continuation. The freshly created key name is used to synchronise the undo of the output with the corresponding input. This is why the backtrack process X is prefixed with an action on key name y. The encodings of a keyed input prefix and a τ prefix are similar. The only difference is that the encoding of the input receives the fresh key name from the output one, while we do not need a key name for a τ prefix.

The image of the above encoding is a subset of internal π of order two $\pi\mathbf{I}^2$, since communicated names are used for synchronisation only. According to [30], CCS and $\pi\mathbf{I}^2$ have distinct expressive power. This result is not however explicitly lifted in [30] to an impossibility result of encoding internal π into CCS. See Section 7 for further discussion.

▶ **Example 5.2.** Consider the CCSK process $a.\overline{b}.\mathbf{0}$. Since $a.\overline{b}.\mathbf{0}$ is standard, ϕ is $\emptyset$. The encoding $[\![a.\overline{b}.\mathbf{0}]\!]$ is $\nu\emptyset.[\![a.\overline{b}.\mathbf{0}, \mathbf{0}, \phi]\!]$. Simplifying and omitting ϕ, we calculate $[\![a.\overline{b}.\mathbf{0}, \mathbf{0}]\!]$ as

$$
\begin{aligned}
&\mathtt{rec}\ X_a.(\mathbf{0} + a(x_a).[\![\overline{b}.\mathbf{0}, x_a.X_a]\!]) = \mathtt{rec}\ X_a.a(x_a).\mathtt{rec}\ Y_{\overline{b}}.(x_a.X_a + \overline{b}(y_b).[\![\mathbf{0}, \overline{y_b}.Y_{\overline{b}}]\!]) \\
&\quad = \mathtt{rec}\ X_a.a(x_a).\mathtt{rec}\ Y_{\overline{b}}.(x_a.X_a + \overline{b}(y_b).(\overline{y_b}.Y_{\overline{b}} + \mathbf{0})) \\
&\quad = \mathtt{rec}\ X_a.a(x_a).\mathtt{rec}\ Y_{\overline{b}}.(x_a.X_a + \overline{b}(y_b).\overline{y_b}.Y_{\overline{b}})
\end{aligned}
$$

As highlighted by the choice of the names of the recursion variables (which is immaterial, since they are bound), undoing a by synchronising on channel x_a leads back to X_a, and analogously undoing $\overline{b}$ by synchronising on channel y_b leads back to $Y_{\overline{b}}$. ⌟

▶ **Example 5.3.** Consider process $P = a[k].(b[h].\mathbf{0} + c.\mathbf{0})$. Letting $\phi = \{(a,k) \leftrightarrow x_k, (b,h) \leftrightarrow x_h\}$, and noting that K is empty since both keys are free, $[\![P, \mathbf{0}, \phi]\!]$ is calculated as

$$
\begin{aligned}
&[\![(b[h].\mathbf{0} + c.\mathbf{0}), x_k.[\![a.\mathsf{toStd}(b[h].\mathbf{0} + c.\mathbf{0}), \mathbf{0}]\!]]\!] = [\![(b[h].\mathbf{0} + c.\mathbf{0}), x_k.[\![a.(b.\mathbf{0} + c.\mathbf{0}), \mathbf{0}]\!]]\!] \\
&\quad = [\![\mathbf{0}, x_h.[\![b.\mathbf{0} + c.\mathbf{0}, x_k.[\![a.(b.\mathbf{0} + c.\mathbf{0}), \mathbf{0}]\!]]\!]]\!] = \mathbf{0} + x_h.[\![b.\mathbf{0} + c.\mathbf{0}, x_k.[\![a.(b.\mathbf{0} + c.\mathbf{0}), \mathbf{0}]\!]]\!] \\
&\quad = x_h.[\![b.\mathbf{0} + c.\mathbf{0}, x_k.[\![a.(b.\mathbf{0} + c.\mathbf{0}), \mathbf{0}]\!]]\!]
\end{aligned}
$$

where (for the sake of readability, we use colours to highlight the substituted subterms):

$$[\![a.(b.\mathbf{0}+c.\mathbf{0}),\mathbf{0}]\!] = \texttt{rec}\ X_k.[\![a.(b.\mathbf{0}+c.\mathbf{0}),X_k]\!]$$
$$[\![b.\mathbf{0}+c.\mathbf{0},x_k.[\![a.(b.\mathbf{0}+c.\mathbf{0}),\mathbf{0}]\!]]\!] = \texttt{rec}\ Y_h.(x_k.[\![a.(b.\mathbf{0}+c.\mathbf{0}),\mathbf{0}]\!]+[\![b.\mathbf{0},Y_h]\!]+[\![c.\mathbf{0},Y_h]\!]).$$

Assuming $R = x_h.\texttt{rec}\ Y_h.(x_k.\texttt{rec}\ X_k.[\![a.(b.\mathbf{0}+c.\mathbf{0}),X_k]\!]+[\![b.\mathbf{0},Y_h]\!]+[\![c.\mathbf{0},Y_h]\!])$, we have $[\![P,\mathbf{0},\phi]\!] = \nu\emptyset.R$. We shall omit the outermost restriction in the rest of the paper.

If we look at the process P, the only move it can make is $P \overset{b[h]}{\rightsquigarrow} a[k].(b.\mathbf{0}+c.\mathbf{0})$, to a process which can still either undo $a[k]$ or move forward by doing b or c. This behaviour is perfectly matched by R since the only action that R can perform is x_h (mimicking the undoing of $b[h]$):

$$R \xrightarrow[\pi]{x_h} \quad \texttt{rec}\ Y_h.(x_k.\texttt{rec}\ X_k.[\![a.(b.\mathbf{0}+c.\mathbf{0}),X_k]\!]+[\![b.\mathbf{0},Y_h]\!]+[\![c.\mathbf{0},Y_h]\!]) \qquad \lrcorner$$

▶ **Example 5.4.** Let us consider the CCSK processes $P_1 = a.\overline{b}.\mathbf{0}$ and $P_2 = b.\mathbf{0}$. According to CCSK semantics we can derive the following computation:

$$P_1 \mid P_2 \overset{a[k]}{\rightharpoonup} a[k].\overline{b}.\mathbf{0} \mid b.\mathbf{0} \overset{\tau[w]}{\rightharpoonup} a[k].\overline{b}[w].\mathbf{0} \mid b[w].\mathbf{0} \overset{\tau[w]}{\rightsquigarrow} a[k].\overline{b}.\mathbf{0} \mid b.\mathbf{0}$$

The encoding of $P_1 \mid P_2$ is $[\![P_1 \mid P_2,\mathbf{0}]\!]$, which we calculate as

$$[\![P_1,\mathbf{0}]\!] \mid [\![P_2,\mathbf{0}]\!] = \texttt{rec}\ X_a.a(x_a).\texttt{rec}\ Y_{\overline{b}}.(x_a.X_a+\overline{b}(y_b).\overline{y_b}.Y_{\overline{b}}) \mid \texttt{rec}\ Y_b.b(y_b).y_b.Y_b$$

The encoded process reproduces the transitions of $P_1 \mid P_2$ as follows. Let us note that for the sake of readability we do not substitute the recursive process variables.

$$[\![P_1 \mid P_2,\mathbf{0}]\!] \xrightarrow[\pi]{ax_a}$$
$$\texttt{rec}\ Y_{\overline{b}}.(x_a.X_a+\overline{b}(y_b).\overline{y_b}.Y_{\overline{b}}) \mid \texttt{rec}\ Y_b.b(y_b).y_b.Y_b \xrightarrow[\pi]{\tau} \tag{1}$$
$$\nu y_b.(\overline{y_b}.Y_{\overline{b}} \mid y_b.Y_b) \xrightarrow[\pi]{\tau} \tag{2}$$
$$\nu y_b.(\texttt{rec}\ Y_{\overline{b}}.(x_a.X_a+\overline{b}(y_b).\overline{y_b}.Y_{\overline{b}}) \mid \texttt{rec}\ Y_b.b(y_b).y_b.Y_b) \equiv_\pi \tag{3}$$
$$\texttt{rec}\ Y_{\overline{b}}.(x_a.X_a+\overline{b}(y_b).\overline{y_b}.Y_{\overline{b}}) \mid \texttt{rec}\ Y_b.b(y_b).y_b.Y_b \tag{4}$$

The two processes in (1) can synchronise on b and reach the process (2), from which the two processes can *undo* the synchronisation on b by synchronising on the restricted key name y_b. This synchronisation will lead to the process (3), and by garbage collecting the restricted key name y_b via $\equiv_\pi$ we get to process (4), which is equal to process (1). $\lrcorner$

Correctness

To prove the correctness of our encoding w.r.t. strong bisimulation, we have to recast the classical definition of strong bisimilarity [31] to work on two different semantics and calculi.

▶ **Definition 5.5** (CCSK-π strong bisimilarity). *A relation $\mathcal{R}$ between CCSK processes and π processes parametrised with a bijection ϕ from CCSK names and keys to π key names is a* CCSK-π bisimulation *iff $(P,R)_\phi \in \mathcal{R}$ implies the following:*

- *if $P \overset{\overline{a}[k]}{\rightharpoonup} P'$ then $R \xrightarrow[\pi]{\overline{a}(x)} R'$ with $(P',R')_{\phi[(a,k)\leftrightarrow x]} \in \mathcal{R}$;*
- *if $P \overset{a[k]}{\rightharpoonup} P'$ then $R \xrightarrow[\pi]{ax} R'$ with $(P',R')_{\phi[(a,k)\leftrightarrow x]} \in \mathcal{R}$;*
- *if $P \overset{\tau[k]}{\rightharpoonup} P'$ then $R \xrightarrow[\pi]{\tau} R'$ with $(P',R')_{\phi} \in \mathcal{R}$;*

- *if* $P \overset{\overline{a}[k]}{\rightsquigarrow} P'$ *then* $R \overset{\overline{x}}{\underset{\pi}{\longrightarrow}} R'$ *with* $(P', R')_{\phi\setminus(a,k)} \in \mathcal{R}$ *where* $\phi(a,k) = x$;
- *if* $P \overset{a[k]}{\rightsquigarrow} P'$ *then* $R \overset{x}{\underset{\pi}{\longrightarrow}} R'$ *with* $(P', R')_{\phi\setminus(a,k)} \in \mathcal{R}$ *where* $\phi(a,k) = x$;
- *if* $P \overset{\tau[k]}{\rightsquigarrow} P'$ *then* $R \overset{\tau}{\underset{\pi}{\longrightarrow}} R'$ *with* $(P', R')_{\phi} \in \mathcal{R}$.

and vice versa. We denote with $\sim$*, dubbed* CCSK-π strong bisimilarity*, the largest CCSK-*π *strong bisimulation.*

As mentioned before, ϕ only tracks "free names", which in the case of CCSK means keys with just one occurrence inside the term, attached to a non-τ prefix. This is needed since such occurrences correspond to synchronisations with the context, which contains the other occurrence of the name. Hence, ϕ is needed to remember how we translated keys whose corresponding name is known by the context, to ensure that the CCSK process and its π-calculus encoding remain aligned on which name represents which key. In the case of bound keys, the corresponding key name is bound in the π-calculus and can be α-converted. There is no need to track the correspondence with such a name, since the name will never occur in the context.

We now define a function φ that, given a CCSK process P, builds the concrete relation used in its encoding. Function φ is actually a bijection, thanks to [12, Prop. 3.5].

▶ **Definition 5.6.** *Let* φ *be a bijection calculated inductively on the CCSK syntax as follows:*

$$\varphi(P) = \emptyset \text{ if } \mathsf{std}(P) \qquad \varphi(\tau[k].P) = \varphi(P)$$
$$\varphi(a[k].P) = \varphi(\overline{a}[k].P) = \{(a,k) \leftrightarrow x_k\} \cup \varphi(P) \qquad \varphi(P+Q) = \varphi(P) \cup \varphi(Q)$$
$$\varphi(P \mid Q) = \varphi(P) \cup \varphi(Q) \setminus (\varphi(P) \cap \varphi(Q)) \qquad \varphi(\nu a.P) = \varphi(P)$$

▶ **Theorem 5.7.** *Let* P *be a CCSK process with parallel composition only at the top level. Then* $(P, [\![P, \mathbf{0}, \varphi(P)]\!]) \in \sim$.

Strong bisimilarity, which we use as the correctness criterion in Theorem 5.7, is stronger than both interaction sensitivity, used in the definition of basic encoding (cf. Definition 4.4), and success sensitivity (cf. Definition 4.14). The reason why the result above is not in contrast with the separation result (Theorem 4.18) in Section 4 is that the encoding is not compositional. Indeed, the encoding of P in a process such as $a[k_1].P$ depends on its context $a[k_1].\bullet$. More precisely, by looking at the encoding in Figure 6, one can see that P is encoded in the case of a choice as a recursive process containing itself a choice, whose first alternative is the backtrack process R, needed to implement the backward move undoing $a[k_1]$. Reversing to $a.P$ requires knowing a (and how it has been encoded), and cannot be deduced by looking at P alone. This is indeed why R is needed as a parameter of the encoding. When R is not $\mathbf{0}$, the encoding is not interaction sensitive either, since the encoding of $\mathbf{0}$ is R. This does not spoil our bisimilarity result, since at the top level R is always $\mathbf{0}$.

The encoding presented in Figure 6 only considers top-level parallel composition (cf. Definition 2.2). If instead, in $a[k_1].P$, process P is composed of multiple parallel processes, they need to coordinate to decide whether the undo of a is enabled. Such an interaction is at least n-ary, where n is the number of parallel components, but this is done in a single CCSK step, thanks to predicate $\mathsf{std}(Q)$ in rule (Act1$^\bullet$) of Figure 2. Hence, no encoding correct w.r.t. strong bisimilarity preserving the degree of parallelism can exist.

In the literature (e.g., in [20]), the concept of preserving the degree of parallelism is frequently formalised by stating that the encoding is homomorphic w.r.t. parallel composition, namely that $[\![P_1 \mid P_2]\!] = [\![P_1]\!] \mid [\![P_2]\!]$. Indeed, the encoding above actually satisfies this property.

However, such a property can be useful when reasoning about lower-level parallel composition, especially when combined with a form of compositionality for other operators, prefix in particular. One could require that for each CCSK prefix ρ there exists a π-calculus context C_ρ such that the encoding of $\rho.P$ is $C_\rho[[\![P]\!]]$. However, this would lead to a compositional encoding, but as we have shown in Section 4 this is not possible (paired with interaction sensitivity; see Theorem 4.18).

Hence we require a weaker form of compositionality, allowing the context to influence the encoding, but preserving the number of parallel components, and requiring parallel components to be encoded independently.

▶ **Definition 5.8.** *An encoding is* parallel preserving *if for every context $C[\bullet]$ and every $n \geq 2$ there is a context $G[\bullet]$, such that for all $P_1, \ldots, P_n$ with no top-level parallel composition there are $R_1, \ldots, R_n$ such that $[\![C[P_1 \mid \cdots \mid P_n]]\!] = G[R_1 \mid \cdots \mid R_n]$. Also, R_i only depends on C, n, i and P_i $(i = 1, \ldots, n)$.*

The last condition in the definition means that the encodings R_i of different parallel components are independent.

▶ **Proposition 5.9.** *The encoding in Figure 6 is parallel preserving.*

The encoding presented in Figure 6 is parallel preserving and preserves strong bisimilarity; however, it is defined on the fragment of CCSK with only top-level parallel composition.

▶ **Theorem 5.10.** *No parallel-preserving encoding of CCSK into the π-calculus preserves strong bisimilarity.*

6 Encoding lower-level parallel composition

The encoding in Section 5 is only defined for CCSK processes with top-level parallel composition. We have shown that no parallel-preserving encoding of CCSK (with parallel composition at lower levels as well as at top level) into the π-calculus can be correct with respect to strong bisimilarity. However, in this section, we extend our encoding to cover the whole of CCSK, and show that the resulting encoding is parallel preserving. The price to pay is that we can only state its correctness under a weaker notion, namely *weak mutual simulation* [19].

The main challenge in extending the encoding is in the implementation of backward moves, since a backward move is enabled only if its continuation is standard (cf. the hypothesis of rule (Act1$^\bullet$) in Figure 2). In the encoding in Figure 6, a standard process triggered the backward move via a process R. Now, however, all the processes in an arbitrary parallel composition need to agree on allowing a backward move: they all need to be standard. To this end, we define the $\mathtt{tr}(N, R)$ function, which given a set N of names and a process R generates a sequential π process that implements a "tree" of all possible sequences of inputs on names in N leading to R. This function builds a process which waits for "rollback signals", as outputs on names in N, from all the parallel subprocesses. When they are all received, meaning that all subprocesses are standard, function $\mathtt{tr}(N, R)$ triggers rollback by executing R. Hence, it needs to account for all the possible interleavings of "rollback signals".

▶ **Example 6.1.** Consider $N = \{a, b, c\}$, and some R. We want to construct a π process that requires synchronisations on names a, b, c, in any order before reaching R. This process is $a.(b.c.R + c.b.R) + b.(a.c.R + c.a.R) + c.(a.b.R + b.a.R)$; it has a tree-like structure. ⌟

▶ **Definition 6.2** (Extended encoding). *The function* $\mathtt{tr} : 2^{\mathcal{N}_\pi} \times \mathsf{Proc}_\pi \to \mathsf{Proc}_\pi$ *is given by:*

$$\mathtt{tr}(N, R) = \begin{cases} R & \text{if } |N| = 0; \\ \Sigma_{a \in N} a.(\mathtt{tr}(N \setminus \{a\}, R)) & \text{otherwise.} \end{cases}$$

We extend the encoding in Figure 6 with the following clause:

$$[\![\prod_I P_i, R]\!] = \nu \tilde{x}.\big(\prod_I [\![P_i, \overline{x_i}]\!] \mid \mathtt{tr}(\tilde{x}, R)\big)$$

where $\tilde{x} = x_1, \cdots, x_n$ *and* $|I| = |\tilde{x}|$*, and each* P_i *does not contain any top-level parallel composition operator.*

Process $\mathtt{tr}(\tilde{x}, P)$ waits for rollback signals $\overline{x_i}$ from every parallel component and, when all of them have been received, it triggers the rollback process R. If R is $\mathbf{0}$ the system terminates, while in CCSK this means that the process is standard and so backward moves are disabled. To avoid this issue, top-level parallel composition is encoded with the rule in Figure 6.

▶ Remark 6.3. The encoding above allows for rollback messages $\overline{x_i}$ to be received in any order. Picking a specific order would not change the correctness result, but will make the encoding of structural congruent processes different, making the proof more complex. This would be however better in a practical setting, since it reduces the size of the resulting process (from a factorial of the number of parallel components to linear).

▶ **Example 6.4.** Let us consider the encoding of the CCSK process $a.(b.\mathbf{0} \mid c.\mathbf{0})$.

$$\begin{aligned}
&[\![a.(b.\mathbf{0} \mid c.\mathbf{0}), \mathbf{0}]\!] = \mathtt{rec}\ X_a.(\mathbf{0} + [\![a.(b.\mathbf{0} \mid c.\mathbf{0}), X_a]\!]) \\
&= \mathtt{rec}\ X_a.\big(\mathbf{0} + a(y_a).([\![b.\mathbf{0} \mid c.\mathbf{0}, y_a.X_a]\!])\big) \\
&= \mathtt{rec}\ X_a.\big(\mathbf{0} + a(y_a).(\nu x_1, x_2.([\![b.\mathbf{0}, \overline{x_1}]\!] \mid [\![c.\mathbf{0}, \overline{x_2}]\!] \mid \mathtt{tr}(\{x_1, x_2\}, y_a.X_a)\big) \\
&= \mathtt{rec}\ X_a.\big(\mathbf{0} + a(y_a).(\nu x_1, x_2.([\![b.\mathbf{0}, \overline{x_1}]\!] \mid [\![c.\mathbf{0}, \overline{x_2}]\!] \mid x_1.x_2.y_a.X_a + x_2.x_1.y_a.X_a)\big) \\
&= \mathtt{rec}\ X_a.\big(a(y_a).(\nu x_1, x_2.(\mathtt{rec}\ X_b.(\overline{x_1} + [\![b.\mathbf{0}, X_b]\!]) \mid \\
&\qquad \mathtt{rec}\ X_c.(\overline{x_2} + [\![c.\mathbf{0}, X_c]\!]) \mid x_1.x_2.y_a.X_a + x_2.x_1.y_a.X_a)\big) \\
&= \mathtt{rec}\ X_a.\big(a(y_a).(\nu x_1, x_2.(\mathtt{rec}\ X_b.(\overline{x_1} + b(y_1).y_1.X_b) \mid \\
&\qquad \mathtt{rec}\ X_c.(\overline{x_2} + c(y_2).y_2.X_c) \mid x_1.x_2.y_a.X_a + x_2.x_1.y_a.X_a)\big)
\end{aligned}$$

⌟

▶ **Example 6.5.** Consider the encoding of $P = a[h].(b[k].\mathbf{0} \mid c[l].\mathbf{0})$ with a bijection $\phi = \{(a, h) \leftrightarrow x_h, (b, k) \leftrightarrow x_k, (c, l) \leftrightarrow x_l\}$. We have

$$\begin{aligned}
[\![P, \phi]\!] &= [\![(b[k].\mathbf{0} \mid c[l].\mathbf{0}), x_h.[\![a.\mathsf{toStd}(b[k].\mathbf{0} \mid c[l].\mathbf{0}), \mathbf{0}]\!]\!] \\
&= [\![(b[k].\mathbf{0} \mid c[l].\mathbf{0}), x_h.[\![a.(b.\mathbf{0} \mid c.\mathbf{0}), \mathbf{0}]\!]\!] \\
&= \nu x_1, x_2.\big([\![b[k].\mathbf{0}, \overline{x_1}]\!] \mid [\![c[l].\mathbf{0}, \overline{x_2}]\!] \mid \mathtt{tr}(\{x_1, x_2\}, x_h.[\![a.(b.\mathbf{0} \mid c.\mathbf{0})]\!])\big)
\end{aligned}$$

We obtain $\mathtt{tr}(\{x_1, x_2\}, x_h.[\![a.(b.\mathbf{0} \mid c.\mathbf{0})]\!]) = x_1.x_2.x_h.[\![a.(b.\mathbf{0} \mid c.\mathbf{0})]\!] + x_2.x_1.x_h.[\![a.(b.\mathbf{0} \mid c.\mathbf{0})]\!]$, and abbreviating the tree process as $\mathtt{tr}(\cdots)$ below, we continue working out $[\![P, \phi]\!]$:

$$\begin{aligned}
&= \nu x_1, x_2.\big([\![\mathbf{0}, x_k.[\![b.\mathbf{0}, \overline{x_1}]\!]]\!] \mid [\![\mathbf{0}, x_l.[\![c.\mathbf{0}, \overline{x_2}]\!]]\!] \mid \mathtt{tr}(\cdots)\big) \\
&= \nu x_1, x_2.\big(x_k.[\![b.\mathbf{0}, \overline{x_1}]\!] \mid x_l.[\![c.\mathbf{0}, \overline{x_2}]\!] \mid \mathtt{tr}(\cdots)\big) \\
&= \nu x_1, x_2.\big(x_k.\mathtt{rec}\ X_b.(\overline{x_1} + b(y_b).[\![\mathbf{0}, y_b.X_b]\!]) \mid x_l.\mathtt{rec}\ X_c.(\overline{x_2} + c(y_c).[\![\mathbf{0}, y_c.X_c]\!]) \mid \mathtt{tr}(\cdots)\big) \\
&= \nu x_1, x_2.\big(x_k.\mathtt{rec}\ X_b.(\overline{x_1} + b(y_b).y_b.X_b) \mid x_l.\mathtt{rec}\ X_c.(\overline{x_2} + c(y_c).y_c.X_c) \mid \mathtt{tr}(\cdots)\big)
\end{aligned}$$

The process P can undo either $b[k]$ or $c[l]$. This is mimicked by its encoding since $[\![P, \phi]\!]$ can undo b on channel x_k or can undo c on channel x_l. Consider $P \overset{b[k]}{\rightsquigarrow} \overset{c[l]}{\rightsquigarrow} a[h].(b.\mathbf{0} \mid c.\mathbf{0}) = P_a$. The corresponding transitions of its encoding are as follows:

$$[\![P, \phi]\!] \xrightarrow[\pi]{x_k} \xrightarrow[\pi]{x_l} \nu x_1, x_2.\big(\texttt{rec}\ X_b.(\overline{x_1}{+}b(y_b).y_b.X_b) \mid \texttt{rec}\ X_c.(\overline{x_2}{+}c(y_c).y_c.X_c) \mid \texttt{tr}(\cdots)\big) = R_a$$

P_a can immediately undo $a[h]$ to $a.(b.\mathbf{0} \mid c.\mathbf{0})$, while the two `rec` subprocesses of R_a need to first synchronise with $\texttt{tr}(\cdots)$ on x_1, x_2 (see below), before performing x_h, which corresponds to $a[h]$, and returning to $[\![a.(b.\mathbf{0} \mid c.\mathbf{0})]\!]$. So an atomic transition in CCSK is represented by three transitions in a π encoding.

$$\begin{aligned} R_a &\xrightarrow[\pi]{\tau} \nu x_1, x_2.(\mathbf{0} \mid \texttt{rec}\ X_c.(\overline{x_2} + c(y_c).y_c.X_c) \mid x_2.x_h.[\![a.(b.\mathbf{0} \mid c.\mathbf{0})]\!]) \\ &\xrightarrow[\pi]{\tau} (\mathbf{0} \mid \mathbf{0} \mid x_h.[\![a.(b.\mathbf{0} \mid c.\mathbf{0})]\!]) \equiv_\pi x_h.[\![a.(b.\mathbf{0} \mid c.\mathbf{0})]\!] \end{aligned}$$ ⌟

Correctness

In this subsection we establish the correctness of the weak encoding $[\![-]\!]$ from CCSK into the π-calculus. The result is stated as a (weak) mutual simulation property [19].

▶ **Definition 6.6** (CCSK-π mutual simulation)**.** *A pair of relations $(\mathcal{R}_1, \mathcal{R}_2)$ between CCSK processes and π processes indexed by a bijection ϕ from CCSK names and keys to π key names is a* CCSK-π mutual simulation *iff $(P, R)_\phi \in \mathcal{R}_1$ implies*

1. *if $P \overset{\overline{a}[k]}{\rightharpoonup} P'$ then $R \xrightarrow[\pi]{\overline{a}(x)} R'$ with $(P', R')_{\phi[(a,k)\mapsto x]} \in \mathcal{R}_1$;*
2. *if $P \overset{a[k]}{\rightharpoonup} P'$ then $R \xrightarrow[\pi]{ax} R'$ with $(P', R')_{\phi[(a,k)\mapsto x]} \in \mathcal{R}_1$;*
3. *if $P \overset{\tau[k]}{\rightharpoonup} P'$ then $R \xrightarrow[\pi]{\tau} R'$ with $(P', R')_\phi \in \mathcal{R}_1$;*
4. *if $P \overset{\overline{a}[k]}{\rightsquigarrow} P'$ then $R \overset{\overline{x}}{\underset{\pi}{\Longrightarrow}} R'$ with $(P', R')_{\phi_{\setminus(a,k)}} \in \mathcal{R}_1$ where $\phi(a, k) = x$;*
5. *if $P \overset{a[k]}{\rightsquigarrow} P'$ then $R \overset{x}{\underset{\pi}{\Longrightarrow}} R'$ with $(P', R')_{\phi_{\setminus(a,k)}} \in \mathcal{R}_1$ where $\phi(a, k) = x$;*
6. *if $P \overset{\tau[k]}{\rightsquigarrow} P'$ then $R \overset{\tau}{\underset{\pi}{\Longrightarrow}} R'$ with $(P', R')_\phi \in \mathcal{R}_1$;*

and $(R, P)_\phi \in \mathcal{R}_2$ implies

7. *if $R \xrightarrow[\pi]{\overline{a}(x)} R'$ then $P \overset{\overline{a}[k]}{\rightharpoonup} P'$ with $(R', P')_{\phi[(a,k)\mapsto x]} \in \mathcal{R}_2$;*
8. *if $R \xrightarrow[\pi]{ax} R'$ then $P \overset{a[k]}{\rightharpoonup} P'$ with $(R', P')_{\phi[(a,k)\mapsto x]} \in \mathcal{R}_2$;*
9. *if $R \overset{\overline{x}}{\underset{\pi}{\Longrightarrow}} R'$ then $P \overset{\overline{a}[k]}{\rightsquigarrow} P'$ with $(R', P')_{\phi_{\setminus(a,k)}} \in \mathcal{R}_2$ where $\phi(a, k) = x$;*
10. *if $R \overset{x}{\underset{\pi}{\Longrightarrow}} R'$ then $P \overset{a[k]}{\rightsquigarrow} P'$ with $(R', P')_{\phi_{\setminus(a,k)}} \in \mathcal{R}_2$ where $\phi(a, k) = x$;*
11. *if $R \overset{\tau}{\underset{\pi}{\Longrightarrow}} R'$ then one of the following happens:*
    - a. *$P \overset{\tau[k]}{\rightharpoonup} P'$ with $(R', P')_\phi \in \mathcal{R}_2$;*
    - b. *$P \overset{\tau[k]}{\rightsquigarrow} P'$ with $(R', P')_\phi \in \mathcal{R}_2$;*
    - c. *$(R', P)_\phi \in \mathcal{R}_2$.*

We say that a CCSK process P and a π-calculus process R are mutually similar if there exists a CCSK-π mutual simulation $(\mathcal{R}_1, \mathcal{R}_2)$ with $(P, R)_\phi \in \mathcal{R}_1$ and $(R, P)_\phi \in \mathcal{R}_2$.

Note that $(\mathcal{R}_1, \mathcal{R}_2)$ is a pair of simulations: $\mathcal{R}_1$ shows how a CCSK process can be simulated by a π process, and $\mathcal{R}_2$ shows how a π process can be simulated by a CCSK process. $\mathcal{R}_1$ relates a CCSK process to its canonical encoding, whereas $\mathcal{R}_2$ also accounts for intermediate τ-reachable target states arising from synchronisation with $\texttt{tr}(\cdots)$ (rollback propagation).

▶ **Theorem 6.7** (Correctness for arbitrary parallel composition). *Let P be a CCSK process. Then P and $[\![P, \mathbf{0}, \varphi(P)]\!]$ are CCSK-π mutually similar.*

Mutual simulation implies both interaction sensitivity and success sensitivity, but the encoding is not compositional; hence this result is not in contrast with the impossibility result in Section 4.

▶ **Proposition 6.8.** *The encoding in Figure 6 extended in Definition 6.2 is parallel preserving.*

7 Related work and conclusion

Related work. The expressive power of process calculi has been extensively studied by means of encodings and separation results, see, e.g., [20, 6, 29, 22, 24]. Among the calculi closer to ours, we recall the following results. Boreale [2] gives a compositional encoding of the asynchronous π-calculus (π_a) (without the match operator) into the π-calculus with internal mobility (πI) [30], which is the target of our encodings of CCSK. However, there exists no valid encoding of π_a (with the match operator) into CCS [6, Theorem 5.1]. (A weaker version of this result was first shown in [7].) The π-calculus with implicit matching can be encoded in CCS_γ [32], which raises the question of whether CCSK can also be encoded in CCS_γ.

Encodings or their impossibility have been far less studied for reversible calculi. In [9] an LTS isomorphism between CCSK and RCCS (and vice versa) is presented (but it relies on encodings which are not compositional), while in [16] a truly concurrent semantics of RCCS is given via an encoding in Petri nets. We are only aware of one encoding of a reversible calculus into a non-reversible one [11]: an encoding of a reversible higher-order asynchronous π-calculus into its irreversible version (extended with abstractions, applications, biadic channels and join patterns) is presented, and proved correct w.r.t. a form of weak barbed bisimilarity. This is in line with our results, since the target calculus in [11] is more expressive than internal π due to the presence of abstractions [30] and join patterns; hence our impossibility results do not apply. Also, like ours, their encoding is not compositional, since it takes as additional parameter a name to be used for rollback.

Conclusion. We have studied the encodability of reversible process calculi into forward-only concurrent models, using CCSK as a representative. Our results show that reversibility strictly increases expressive power: even with parallel composition restricted to the top level, no basic, success-sensitive encoding into CCS or the π-calculus exists.

We pinpoint the boundary of encodability. When parallel composition is limited to the top level, CCSK can be encoded into the internal π-calculus up to strong bisimilarity. With arbitrary parallelism, however, no parallel-preserving encoding achieves strong behavioural correspondence; instead, we provide an encoding correct under weak mutual simulation. Taken together, these results clarify the rôle of reversibility in concurrent computation, showing that it cannot be compiled away without either restricting the source language or weakening the behavioural equivalence. Concerning behavioural equivalence, we conjecture that our last result can be made stronger by using instead of weak mutual simulation a relation inspired by correspondence simulation [23]. This is by design an *asymmetric* relation, where the second term in order to answer a challenge may go through intermediate steps which have no equivalent in the first term. However, the definition in [23] is in an unlabelled setting; hence we would first need to extend it to a labelled one.

Several other directions for future work emerge from this study. First, our encodability and separation results rely on fundamental features of reversibility rather than on specific aspects of CCSK. This suggests that similar limitations and encodability boundaries may

hold for other reversible calculi, such as RCCS [4], reversible variants of the π-calculus [3] or higher-order process calculi [11]. However, this requires detailed analysis, since such calculi separate history information from the actual process; hence, it is not even clear what it means to replace $\mathbf{0}$ with a process with past behaviour, as we have done in the proof of Proposition 4.17. Establishing general criteria for the encodability of reversible models remains an open problem. Second, the encoding for arbitrary parallel composition requires a coordination protocol to ensure that all descendants of a process are reversed before reversing the process itself. Alternative protocols, possibly targeting richer calculi (e.g., with join patterns or higher-order communication), may be more efficient or allow stronger behavioural correspondences. Third, while our work is purely semantic, reversibility is often motivated by practical applications such as debugging, fault recovery, and reversible programming. Investigating how the theoretical limits identified here affect the design of reversible languages and compilation techniques is a relevant direction for future research.

References

1 Laura Bocchi, Ivan Lanese, Claudio Antares Mezzina, and Shoji Yuen. revTPL: The reversible temporal process language. *Log. Methods Comput. Sci.*, 20(1), 2024. doi:10.46298/LMCS-20(1:11)2024.

2 Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theor. Comput. Sci.*, 195(2):205–226, 1998. doi:10.1016/S0304-3975(97)00220-X.

3 Ioana Cristescu, Jean Krivine, and Daniele Varacca. A compositional semantics for the reversible π-calculus. In *Proceedings of LICS 2013*, pages 388–397. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.45.

4 Vincent Danos and Jean Krivine. Reversible communicating systems. In Philippa Gardner and Nobuko Yoshida, editors, *Proceedings of CONCUR 2004*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004. doi:10.1007/978-3-540-28644-8_19.

5 Elena Giachino, Ivan Lanese, and Claudio Antares Mezzina. Causal-consistent reversible debugging. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2014. doi:10.1007/978-3-642-54804-8_26.

6 Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010. doi:10.1016/J.IC.2010.05.002.

7 Bjørn Haagensen, Sergio Maffeis, and Iain Phillips. Matching systems for concurrent calculi. In Roberto M. Amadio and Thomas T. Hildebrandt, editors, *Proceedings of the 14th International Workshop on Expressiveness in Concurrency, EXPRESS 2007, Lisbon, Portugal, September 3, 2007*, volume 194:2 of *Electronic Notes in Theoretical Computer Science*, pages 85–99. Elsevier, 2007. doi:10.1016/J.ENTCS.2007.11.004.

8 Stefan Kuhn and Irek Ulidowski. Modelling of DNA mismatch repair with a reversible process calculus. *Theor. Comput. Sci.*, 925:68–86, 2022. doi:10.1016/J.TCS.2022.06.009.

9 Ivan Lanese, Doriana Medic, and Claudio Antares Mezzina. Static versus dynamic reversibility in CCS. *Acta Informatica*, 58(1-2):1–34, 2021. doi:10.1007/S00236-019-00346-6.

10 Ivan Lanese, Claudio Antares Mezzina, Iain Phillips, Irek Ulidowski, and Shoji Yuen. On the encodability of reversible process calculi. *CoRR*, 2606.25916, 2026. doi:10.48550/arXiv.2606.25916.

11 Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversibility in the higher-order π-calculus. *Theoretical Computer Science*, 625:25–84, 2016. doi:10.1016/j.tcs.2016.02.019.

12 Ivan Lanese and Iain Phillips. Forward-reverse observational equivalences in CCSK. In Shigeru Yamashita and Tetsuo Yokoyama, editors, *Reversible Computation - 13th International Conference, RC 2021*, volume 12805 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2021. doi:10.1007/978-3-030-79837-6_8.

13 Ivan Lanese, Ulrik Pagh Schultz, and Irek Ulidowski. Reversible computing in debugging of Erlang programs. *IT Prof.*, 24(1):74–80, 2022. doi:10.1109/MITP.2021.3117920.

14 James McNellis, Jordi Mola, and Ken Sykes. Time travel debugging: root causing bugs in commercial scale software. CppCon talk, https://www.youtube.com/watch?v=l1YJTg_A914, 2017.

15 Hernán C. Melgratti, Claudio Antares Mezzina, and G. Michele Pinna. A Petri net view of covalent bonds. *Theor. Comput. Sci.*, 908:89–119, 2022. doi:10.1016/J.TCS.2022.01.013.

16 Hernán C. Melgratti, Claudio Antares Mezzina, and G. Michele Pinna. A truly concurrent semantics for reversible CCS. *Log. Methods Comput. Sci.*, 20(4), 2024. doi:10.46298/LMCS-20(4:20)2024.

17 Claudio Antares Mezzina. On reversibility and broadcast. In Jarkko Kari and Irek Ulidowski, editors, *Proceedings of RC 2018*, volume 11106 of *LNCS*, pages 67–83. Springer, 2018. doi:10.1007/978-3-319-99498-7_5.

18 Claudio Antares Mezzina, Francesco Tiezzi, and Nobuko Yoshida. Checkpoint-based rollback recovery in session programming. *Log. Methods Comput. Sci.*, 21(1):2, 2025. doi:10.46298/LMCS-21(1:2)2025.

19 Robin Milner. An algebraic definition of simulation between programs. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*, pages 481–489. William Kaufmann, 1971. URL: http://ijcai.org/Proceedings/71/Papers/044.pdf.

20 Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Math. Struct. Comput. Sci.*, 13(5):685–719, 2003. doi:10.1017/S0960129503004043.

21 Kirstin Peters. Comparing process calculi using encodings. In Jorge A. Pérez and Jurriaan Rot, editors, *Proceedings Combined 26th International Workshop on Expressiveness in Concurrency and 16th Workshop on Structural Operational Semantics, EXPRESS/SOS 2019, Amsterdam, The Netherlands, 26th August 2019*, volume 300 of *EPTCS*, pages 19–38, 2019. doi:10.4204/EPTCS.300.2.

22 Kirstin Peters and Uwe Nestmann. Is it a "good" encoding of mixed choice? In Lars Birkedal, editor, *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7213 of *Lecture Notes in Computer Science*, pages 210–224. Springer, 2012. doi:10.1007/978-3-642-28729-9_14.

23 Kirstin Peters and Rob J. van Glabbeek. Analysing and comparing encodability criteria. In Silvia Crafa and Daniel Gebler, editors, *Proceedings of the Combined 22th International Workshop on Expressiveness in Concurrency and 12th Workshop on Structural Operational Semantics, EXPRESS/SOS 2015, Madrid, Spain, 31st August 2015*, volume 190 of *EPTCS*, pages 46–60, 2015. doi:10.4204/EPTCS.190.4.

24 Kirstin Peters and Nobuko Yoshida. Separation and encodability in mixed choice multiparty sessions. In Pawel Sobocinski, Ugo Dal Lago, and Javier Esparza, editors, *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2024, Tallinn, Estonia, July 8-11, 2024*, pages 62:1–62:15. ACM, 2024. doi:10.1145/3661814.3662085.

25 Iain Phillips. CCS with priority guards. *J. Log. Algebraic Methods Program.*, 75(1):139–165, 2008. doi:10.1016/J.JLAP.2007.06.005.

26 Iain C.C. Phillips and Irek Ulidowski. Reversing algebraic process calculi. In Luca Aceto and Anna Ingólfsdóttir, editors, *Proceedings of FoSSaCS 2006*, volume 3921 of *LNCS*, pages 246–260. Springer, 2006. doi:10.1007/11690634_17.

27 Iain C.C. Phillips and Irek Ulidowski. Reversing algebraic process calculi. *Journal of Logic and Algebraic Programming*, 73(1-2):70–96, 2007. doi:10.1016/j.jlap.2006.11.002.

28 Iain C.C. Phillips, Irek Ulidowski, and Shoji Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In Robert Glück and Tetsuo Yokoyama, editors, *Proceedings of RC 2012*, volume 7581 of *LNCS*, pages 218–232. Springer, 2012. doi:10.1007/978-3-642-36315-3_18.

29 Rosario Pugliese and Francesco Tiezzi. Replacement freeness: A criterion for separating process calculi. *J. Log. Algebraic Methods Program.*, 116:100579, 2020. doi:10.1016/J.JLAMP.2020.100579.

30 Davide Sangiorgi. pi-calculus, internal mobility, and agent-passing calculi. *Theor. Comput. Sci.*, 167(1&2):235–274, 1996. doi:10.1016/0304-3975(96)00075-8.

31 Davide Sangiorgi and David Walker. *The π-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

32 Rob van Glabbeek. Comparing the expressiveness of the π-calculus and CCS. *ACM Trans. Comput. Log.*, 25(1):1:1–1:58, 2024. doi:10.1145/3611013.

33 Martin Vassor and Jean-Bernard Stefani. Checkpoint/rollback vs causally-consistent reversibility. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation - 10th International Conference, RC 2018*, volume 11106 of *Lecture Notes in Computer Science*, pages 286–303. Springer, 2018. doi:10.1007/978-3-319-99498-7_20.