Concurrent Flexible Reversibility

Ivan Lanese, Michaël Lienhardt, Claudio Mezzina, Alan Schmitt,
Jean-Bernard Stefani

Concurrent Flexible Reversibility*

Ivan Lanese[1], Michael Lienhardt[2], Claudio Antares Mezzina[3], Alan Schmitt[4], and Jean-Bernard Stefani[4]

[1] Focus Team, University of Bologna/Inria, Italy lanese@cs.unibo.it
[2] PPS Laboratory, Paris Diderot University, France lienhard@cs.unibo.it
[3] SOA Unit, FBK, Trento, Italy mezzina@fbk.eu
[4] Inria, France alan.schmitt@inria.fr, jean-bernard.stefani@inria.fr

Abstract. Concurrent reversibility has been studied in different areas, such as biological or dependable distributed systems. However, only "rigid" reversibility has been considered, allowing to go back to a past state and restart the exact same computation, possibly leading to divergence. In this paper, we present CCSK, a concurrent calculus featuring *flexible reversibility*, allowing the specification of alternatives to a computation to be used upon rollback. Alternatives in CCSK are attached to messages. We show the robustness of this mechanism by encoding more complex idioms for specifying flexible reversibility, and we illustrate the benefits of our approach by encoding a calculus of communicating transactions.

1 Introduction

Reversible programs can be executed both in the standard, forward direction as well as in the backward direction, to go back to past states. Reversible programming is attracting much interest for its potential in several areas. For instance, chemical and biological reactions are typically bidirectional, and the direction of execution is fixed by environmental conditions such as temperature. Similarly, quantum computations are reversible as long as they are not observed. Reversibility is also used for backtracking in the exploration of a program state-space toward a solution, either as part of the design of the programming language as in Prolog, or to implement transactions. We are particularly interested in the use of reversibility for modeling and programming concurrent reliable systems. In this setting, the main idea is that in case of an error the program backtracks to a past state where the decisions leading to the error have not been taken yet, so that a new forward execution may avoid repeating the (same) error.

Reversibility has a non trivial interplay with concurrency. Understanding this interplay is fundamental in many of the areas above, e.g., for biological or reliable distributed systems, which are naturally concurrent. In the spirit of concurrency, independent threads of execution should be rolled-back independently, but causal dependencies between related threads should be taken into account.

* This work has been partially supported by the French National Research Agency (ANR), projects REVER ANR 11 INSE 007 and PiCoq ANR 10 BLAN 0305.

This form of reversibility, termed *causal consistent*, was first introduced by RCCS [12], a reversible variant of CCS. RCCS paved the way to the definition of reversible variants of more expressive concurrent calculi [9, 20, 22, 24]. This line of research considered rigid, uncontrolled, step-by-step reversibility. *Step-by-step* means that each single step can be undone, as opposed, e.g., to checkpointing where many steps are undone at once. *Uncontrolled* means that there is no hint as to when to go forward and when to go backward, and up to where. *Rigid* means that the execution of a forward step followed by the corresponding backward step leads back to the starting state, where an identical computation can restart.

While these works have been useful to understand the basics of concurrent reversibility in different settings, some means to *control* reversibility are required in practice. In the literature four different forms of control have been proposed: relating the direction of execution to some energy parameter [3], introducing irreversible actions [13], using an explicit rollback primitive [19], and using a superposition operator to control forward and backward execution [26].

With the exception of [26], these works have causally consistent reversibility but exhibit rigid reversibility. However, rigid reversibility may not always be the best choice. In the setting of reliable systems, for instance, rigid reversibility means that to recover from an error a past state is reached. From this past state the computation that lead to the error is still possible. If the error was due to a transient fault, retrying the same computation may be enough to succeed. If the failure was permanent, the program may redo the same error again and again.

Our goal is to overcome this limitation by providing the programmer with suitable linguistic constructs to specify what to do after a causally consistent backward computation. Such constructs can be used to ensure that new forward computations explore new possibilities. To this end, we build on our previous work on roll-π [19], a calculus where concurrent reversibility is controlled by the roll γ operator. Executing it reverses the action referred by γ together with all the dependent actions. Here, we propose a new calculus called croll-π, for compensating roll-π, as a framework for *flexible reversibility*. We attempt to keep croll-π as close as possible to roll-π while enabling many new possible applications. We thus simply replace roll-π communication messages $a\langle P\rangle$ by *messages with alternative* $a\langle P\rangle \div c\langle Q\rangle$. In forward computation, a message $a\langle P\rangle \div c\langle Q\rangle$ behaves exactly as $a\langle P\rangle$. However, if the interaction consuming it is reversed, the original message is not recreated—as would be the case with rigid reversibility—but the alternative $c\langle Q\rangle$ is released instead. Our rollback and alternative message primitives provide a simple form of reversibility control, which always respects the causal consistency of reverse computation. It contrasts with the fine-grained control provided by the superposition constructs in [26], where the execution of a CCS process can be constrained by a controller, possibly reversing identified past actions in a way that is non-causally consistent.

Our contributions are as follows. We show that the simple addition of alternatives to roll-π greatly extends its expressive power. We show that messages with alternative allow for programming different patterns for flexible reversibility. We show that croll-π can be used to model the communicating transactions

of [14]. Notably, the tracking of causality of croll-π is more precise than the one in [14], thus allowing to improve on the original proposal by avoiding some spurious undo of actions. Additionally, we study some aspects of the behavioral theory of croll-π, including a context lemma for barbed congruence. This allows us to reason about croll-π programs, in particular to prove the correctness of the encodings of primitives for flexible reversibility and of the transactional calculus of [14]. Finally, we present an interpreter, written in Maude [11], for a small language based on croll-π.

Outline. Section 2 gives an informal introduction to croll-π. Section 3 defines the croll-π calculus, its reduction semantics, and it introduces the basics of its behavioral theory. Section 4 presents various croll-π idioms for flexible reversibility. Section 5 outlines the croll-π interpreter in Maude and the concurrent solution for the Eight Queens problem. Section 6 presents an encoding and an analysis of the TransCCS constructs from [14]. Section 7 concludes the paper with related work and a mention of future studies. The paper includes short proof sketches for the main results. We refer to the online technical report [18] for full proofs and an additional example, an encoding of the transactional constructs from [2].

2 Informal Presentation

Rigid reversibility in roll-π. The croll-π calculus is a conservative extension of the roll-π calculus introduced in [19].[5] We briefly review the roll-π constructs before presenting the extension added by croll-π. Processes in roll-π are essentially processes of the asynchronous higher-order π-calculus [27], extended with a rollback primitive. Processes in roll-π cannot directly execute, only *configurations* can. A configuration is essentially a parallel composition of *tagged processes* along with *memories* tracking past interactions and *connectors* tracing causality information. In a tagged process of the form $k : P$, the tag k uniquely identifies the process P in a given configuration. We often use the term *key* instead of tag.

The uniqueness of tags in configurations is achieved thanks to the following reduction rule that defines how parallel processes are split.

$$k : P \mid Q \longrightarrow \nu k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

In the above reduction, $\mid$ is the parallel composition operator and ν is the restriction operator, both standard from the π-calculus. As usual, the scope of restriction extends as far to the right as possible. Connector $k \prec (k_1, k_2)$ is used to remember that the process tagged by k has been split into two sub-processes identified by the new keys k_1 and k_2. Thus complex processes can be split into *threads*, where a thread is either a *message*, of the form $a\langle P\rangle$ (where a is a channel name), a receiver process (also called a *trigger*), of the form $a(X) \triangleright_\gamma P$, or a *rollback* instruction of the form roll k, where k is a key.

[5] The version of roll-π presented here is slightly refined w.r.t. the one in [19].

A *forward* communication step occurs when a message on a channel can be received by a trigger on the same channel. It takes the following form (roll-π is an asynchronous higher-order calculus).

$$(k_1 : a\langle P\rangle) \mid (k_2 : a(X) \triangleright_\gamma Q) \longrightarrow \nu k.\, k : Q\{^{P,k}/_{X,\gamma}\} \mid [\mu; k]$$

In this forward step, keys k_1 and k_2 identify threads consisting respectively of a message $a\langle P\rangle$ on channel a and a trigger $a(X) \triangleright_\gamma Q$ expecting a message on channel a. The result of the message input yields, as in higher-order π, the body of the trigger Q with the formal parameter X instantiated by the received value, i.e., process P. Message input also has three side effects: (i) the tagging of the newly created process $Q\{^{P,k}/_{X,\gamma}\}$ by a fresh key k; (ii) the creation of a memory $[\mu; k]$, which records the original two threads,[6] $\mu = (k_1 : a\langle P\rangle) \mid (k_2 : a(X) \triangleright_\gamma Q)$, together with key k; and (iii) the instantiation of variable γ with the newly created key k (the trigger construct is a binder both for its process parameter and its key parameter).

In roll-π, a forward computation, i.e., a series of forward reduction steps as above, can be perfectly undone by backward reductions triggered by the occurrence of an instruction of the form roll k, where k refers to a previously instantiated memory. In roll-π, we have for instance the following forward and backward steps, where $M = (k_1 : a\langle Q\rangle) \mid (k_2 : a(X) \triangleright_\gamma X \mid \mathsf{roll}\ \gamma)$:

$$\begin{aligned} &M \longrightarrow \nu k.\, (k : Q \mid \mathsf{roll}\ k) \mid [M; k] \longrightarrow \\ &\nu k\, k_3\, k_4.\, k \prec (k_3, k_4) \mid k_3 : Q \mid k_4 : \mathsf{roll}\ k \mid [M; k] \longrightarrow M \end{aligned}$$

The communication between threads k_1 and k_2 in the first step and the split of process k into k_3 and k_4 are perfectly undone by the third (backward) step.

More generally, the set of memories and connectors of a configuration M provides us with an ordering $<:$ between the keys of M that reflects their causal dependency: $k' <: k$ means that key k' has key k as *causal descendant*. Thus, the effects of a rollback can be characterized as follows. When a rollback takes place in a configuration M, triggered by an instruction $k_r : \mathsf{roll}\ k$, it suppresses all threads and processes whose tag is a causal descendant of k, as well as all connectors $k' \prec (k_1, k_2)$ and memories $m = [k_1 : \tau_1 \mid k_2 : \tau_2; k']$ whose key k' is a causal descendant of k. When suppressing such a memory m, the rollback operation may release a thread $k_i : \tau_i$ if k_i is not a causal descendant of k (at least one of the threads of m must have k as causal antecedent if k' has k as causal antecedent). This is due to the fact that a thread that is not a causal descendant of k may be involved in a communication (and then captured into a memory) by a descendant of k. This thread can be seen as a resource that is taken from the environment through interaction, and it should be restored in case of rollback. Finally, rolling-back also releases the content μ of the memory $[\mu; k]$ targeted by the roll, reversing the corresponding communication step.

[6] Work can be done to store memories in a more efficient way. We will not consider this issue in the current paper; an approach can be found in [22].

Flexible reversibility in croll-π. In roll-π, a rollback perfectly undoes a computation originated by a specific message receipt. However, nothing prevents the same computation from taking place again and again (although not necessarily in the same context, as independent computations may have proceeded on their own in parallel). To allow for flexible reversibility, we extend roll-π with a single new construct, called a *message with alternative*. In croll-π, a message may now take the form $a\langle P\rangle \div C$, where alternative C may either be a message $c\langle Q\rangle \div \mathbf{0}$ with null alternative or the null process $\mathbf{0}$. When the message receipt of $k : a\langle P\rangle \div C$ is rolled-back, configuration $k : C$ is released instead of the original $k : a\langle P\rangle$, as would be the case in roll-π. (Only the alternative associated to the message in the memory $[\mu; k]$ targeted by the roll is released: other processes may be restored, but not modified.) For example, if $M = (k_1 : a\langle Q\rangle \div \mathbf{0}) \mid (k_2 : a(X) \triangleright_\gamma X \mid \mathsf{roll}\ \gamma)$ then we have the following computation, where the communication leading to the rollback becomes disabled.

$$
\begin{aligned}
&M \longrightarrow \nu k.\, (k : Q \mid \mathsf{roll}\ k) \mid [M; k] \longrightarrow \\
&\nu k\, k_3\, k_4.\, k \prec (k_3, k_4) \mid k_3 : Q \mid k_4 : \mathsf{roll}\ k \mid [M; k] \longrightarrow \\
&k_1 : \mathbf{0} \mid (k_2 : a(X) \triangleright_\gamma X \mid \mathsf{roll}\ \gamma)
\end{aligned}
$$

We will show that croll-π is powerful enough to devise various kinds of alternatives (see Section 4), whose implementation is not possible in roll-π (cf. Theorem 2). Also, thanks to the higher-order aspect of the calculus, the behavior of roll-π can still be programmed: rigid reversibility can be seen as a particular case of flexible reversibility. Thus, the introduction of messages with alternatives has limited impact on the definition of the syntax and of the operational semantics, but it has a strong impact on what can actually be modeled in the calculus and on its theory.

3 The croll-π Calculus: Syntax and Semantics

3.1 Syntax

Names, keys, and variables. We assume the existence of the following denumerable infinite mutually-disjoint sets: the set $\mathcal{N}$ of *names*, the set $\mathcal{K}$ of *keys*, the set $\mathcal{V}_\mathcal{K}$ of *key variables*, and the set $\mathcal{V}_\mathcal{P}$ of *process variables*. $\mathbb{N}$ denotes the set of natural numbers. We let (together with their decorated variants): a, b, c range over $\mathcal{N}$; h, k, l range over $\mathcal{K}$; u, v, w range over $\mathcal{N} \cup \mathcal{K}$; γ range over $\mathcal{V}_\mathcal{K}$; X, Y, Z range over $\mathcal{V}_\mathcal{P}$. We denote by $\tilde{u}$ a finite set $u_1 \ldots u_n$.

Syntax. The syntax of the croll-π calculus is given in Figure 1. *Processes*, given by the P, Q productions, are the standard processes of the asynchronous higher-order π-calculus [27], except for the presence of the roll primitive, the extra bound tag variable in triggers, and messages with alternative that replace roll-π messages $a\langle P\rangle$. The alternative operator $\div$ binds more strongly than any other operator. *Configurations* in croll-π are given by the M, N productions. A configuration is built up from *tagged processes* $k : P$, *memories* $[\mu; k]$, and *connectors*

$$
\begin{array}{rl}
P, Q ::= & \mathbf{0} \mid X \mid \nu a.\, P \mid (P \mid Q) \mid a(X) \triangleright_\gamma P \mid a\langle P \rangle \div C \mid \mathsf{roll}\ k \mid \mathsf{roll}\ \gamma \\
M, N ::= & \mathbf{0} \mid \nu u.\, M \mid (M \mid N) \mid k : P \mid [\mu; k] \mid k \prec (k_1, k_2) \qquad C ::= a\langle P \rangle \div \mathbf{0} \mid \mathbf{0} \\
\mu ::= & (k_1 : a\langle P \rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q) \\
& a, b, c \in \mathcal{N} \quad X, Y, Z \in \mathcal{V}_\mathcal{P} \quad \gamma \in \mathcal{V}_\mathcal{K} \quad u, v, w \in \mathcal{N} \cup \mathcal{K} \quad h, k, l \in \mathcal{K}
\end{array}
$$

Fig. 1. Syntax of croll-π

$k \prec (k_1, k_2)$. In a memory $[\mu; k]$, we call μ the *configuration part* of the memory and k its *key*. $\mathcal{P}$ denotes the set of croll-π processes and $\mathcal{C}$ the set of croll-π configurations. We let (together with their decorated variants) P, Q, R range over $\mathcal{P}$ and L, M, N range over $\mathcal{C}$. We call *thread* a process that is either a message with alternative $a\langle P \rangle \div C$, a trigger $a(X) \triangleright_\gamma P$, or a rollback instruction $\mathsf{roll}\ k$. We let τ and its decorated variants range over threads. We write $\prod_{i \in I} M_i$ for the parallel composition of configurations M_i for each $i \in I$ (by convention $\prod_{i \in I} M_i = \mathbf{0}$ if $I = \emptyset$), and we abbreviate $a\langle \mathbf{0} \rangle$ to $\overline{a}$.

Free identifiers and free variables. Notions of free identifiers and free variables in croll-π are standard. Constructs with binders are of the following forms: $\nu a.\, P$ binds the name a with scope P; $\nu u.\, M$ binds the identifier u with scope M; and $a(X) \triangleright_\gamma P$ binds the process variable X and the key variable γ with scope P. We denote by $\mathtt{fn}(P)$ and $\mathtt{fn}(M)$ the set of free names and keys of process P and configuration M, respectively. Note in particular that $\mathtt{fn}(k : P) = \{k\} \cup \mathtt{fn}(P)$, $\mathtt{fn}(\mathsf{roll}\ k) = \{k\}$. We say that a process P or a configuration M is *closed* if it has no free (process or key) variable. We denote by $\mathcal{P}_{cl}$ and $\mathcal{C}_{cl}$ the sets of closed processes and configurations, respectively. We abbreviate $a(X) \triangleright_\gamma P$, where X is not free in P, to $a \triangleright_\gamma P$; and $a(X) \triangleright_\gamma P$, where γ is not free in P, to $a(X) \triangleright P$.

Remark 1. We have no construct for replicated processes or internal choice in croll-π: as in the higher-order π-calculus, these can easily be encoded.

Remark 2. In the remainder of the paper, we adopt *Barendregt's Variable Convention*: if terms $t_1, \ldots, t_n$ occur in a certain context (e.g., definition, proof), then in these terms all bound identifiers and variables are chosen to be different from the free ones.

3.2 Reduction Semantics

The reduction semantics of croll-π is defined via a reduction relation $\longrightarrow$, which is a binary relation over closed configurations ($\longrightarrow \subset \mathcal{C}_{cl} \times \mathcal{C}_{cl}$), and a structural congruence relation $\equiv$, which is a binary relation over configurations ($\equiv \subset \mathcal{C} \times \mathcal{C}$). We define *configuration contexts* as "configurations with a hole $\bullet$", given by the grammar: $\mathbb{C} ::= \bullet \mid (M \mid \mathbb{C}) \mid \nu u.\, \mathbb{C}$. *General contexts* $\mathbb{G}$ are just configurations with a hole $\bullet$ in a place where an arbitrary process P can occur. A *congruence* on processes or configurations is an equivalence relation $\mathcal{R}$ that

$$\text{(E.ParC)}\ M \mid N \equiv N \mid M \qquad \text{(E.ParA)}\ M_1 \mid (M_2 \mid M_3) \equiv (M_1 \mid M_2) \mid M_3$$

$$\text{(E.NilM)}\ M \mid \mathbf{0} \equiv M \qquad \text{(E.NewN)}\ \nu u.\, \mathbf{0} \equiv \mathbf{0}$$

$$\text{(E.NewC)}\ \nu u.\, \nu v.\, M \equiv \nu v.\, \nu u.\, M \qquad \text{(E.NewP)}\ (\nu u.\, M) \mid N \equiv \nu u.\, (M \mid N)$$

$$\text{(E.}\alpha\text{)}\ M =_\alpha N \implies M \equiv N \qquad \text{(E.TagC)}\ k \prec (k_1, k_2) \equiv k \prec (k_2, k_1)$$

$$\text{(E.TagA)}\ \nu h.\, k \prec (h, k_3) \mid h \prec (k_1, k_2) \equiv \nu h.\, k \prec (k_1, h) \mid h \prec (k_2, k_3)$$

Fig. 2. Structural congruence for croll-π.

$$\text{(S.Com)}\ \frac{\mu = (k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q_2)}{(k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q_2) \longrightarrow \nu k.\, (k : Q_2\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$\text{(S.TagN)}\ k : \nu a.\, P \longrightarrow \nu a.\, k : P$$

$$\text{(S.TagP)}\ k : P \mid Q \longrightarrow \nu k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

$$\text{(S.Roll)}\ \frac{k <: N \qquad \texttt{complete}(N \mid [\mu; k] \mid (k_r : \mathsf{roll}\ k)) \qquad \mu' = \texttt{xtr}(\mu)}{N \mid [\mu; k] \mid (k_r : \mathsf{roll}\ k) \longrightarrow \mu' \mid N\!\downarrow_k}$$

$$\text{(S.Ctx)}\ \frac{M \longrightarrow N}{\mathbb{C}[M] \longrightarrow \mathbb{C}[N]} \qquad \text{(S.Eqv)}\ \frac{M \equiv M' \qquad M' \longrightarrow N' \qquad N' \equiv N}{M \longrightarrow N}$$

Fig. 3. Reduction rules for croll-π

is closed for general or configuration contexts: $P\,\mathcal{R}\,Q \implies \mathbb{G}[P]\,\mathcal{R}\,\mathbb{G}[Q]$ and $M\,\mathcal{R}\,N \implies \mathbb{C}[M]\,\mathcal{R}\,\mathbb{C}[N]$.

Structural congruence $\equiv$ is defined as the smallest congruence on configurations that satisfies the axioms in Figure 2, where $t =_\alpha t'$ denotes equality of t and t' modulo α-conversion. Axioms E.ParC to E.α are standard from the π-calculus. Axioms E.TagC and E.TagA model commutativity and associativity of connectors, in order not to have a rigid tree structure. Thanks to axiom E.NewC, $\nu\tilde{u}.\, A$ stands for $\nu u_1 \ldots u_n.\, A$ if $\tilde{u} = u_1\ \ldots\ u_n$.

Configurations can be written in normal form using structural congruence.

Lemma 1 (Normal form). *Given a configuration M, we have:*

$$M \equiv \nu\tilde{n}.\, \prod_i (k_i : P_i) \mid \prod_j [\mu_i; k_j] \mid \prod_l k_l \prec (k'_l, k''_l)$$

The reduction relation $\longrightarrow$ is defined as the smallest binary relation on closed configurations satisfying the rules of Figure 3. This extends the naïve semantics of

roll-π introduced in [19],[7] and outlined here in Section 2, to manage alternatives. We denote by $\Longrightarrow$ the reflexive and transitive closures of $\longrightarrow$.

Reductions are either forward, given by rules S.Com, S.TagN, and S.TagP, or backward, defined by rule S.Roll. They are closed under configuration contexts (rule S.Ctx) and under structural congruence (rule S.Eqv). The rule for communication S.Com is the standard communication rule of the higher-order π-calculus with the side effects discussed in Section 2. Rule S.TagN allows restrictions in processes to be lifted at the configuration level. Rule S.TagP allows to split parallel processes. Rule S.Roll enacts rollback, canceling all the effects of the interaction identified by the unique key k, and releasing the initial configuration that gave rise to the interaction, where the alternative replaces the original message. This is the only difference between croll-π and roll-π: in the latter, the memory μ was directly released. However, this small modification yields significant changes to the expressive power of the calculus, as we will see later.

The rollback impacts only the causal descendants of k, defined as follows.

Definition 1 (Causal dependence). *Let M be a configuration and let $\mathcal{T}_M$ be the set of keys occurring in M. Causal dependence $<:_M$ is the reflexive and transitive closure of $<_M$, which is defined as the smallest binary relation on $\mathcal{T}_M$ satisfying the following clauses:*

- *$k <_M k'$ if $k \prec (k_1, k_2)$ occurs in M with $k' = k_1$ or $k' = k_2$;*
- *$k <_M k'$ if a thread $k : P$ occurs (inside μ) in a memory $[\mu; k']$ of M.*

If the configuration M is clear from the context, we write $k <: k'$ for $k <:_M k'$.

A backward reduction triggered by roll k involves *all* and *only* the descendants of key k. We ensure they are all selected by requiring that the configuration is *complete*, and that no other term is selected by requiring *k-dependence*.

Definition 2 (Complete configuration). *A configuration M is* complete*, denoted as* `complete`(M)*, if, for each memory $[\mu; k]$ and each connector $k' \prec (k, k_1)$ or $k' \prec (k_1, k)$ that occurs in M there exists in M either a connector $k \prec (h_1, h_2)$ or a tagged process $k : P$ (possibly inside a memory).*

A configuration M is k-dependent if all its components depend on k.

Definition 3 (k-dependence). *Let M be a configuration such that:*
$M \equiv \nu\tilde{u}.\ \prod_{i\in I}(k_i : P_i) \mid \prod_{j\in J}[\mu_j; k_j] \mid \prod_{l\in L} k_l \prec (k'_l, k''_l)$ with $k \notin \tilde{u}$.
Configuration M is k-dependent, written $k <: M$ by overloading the notation for causal dependence among keys, if for every i in $I \cup J \cup L$, we have $k <:_M k_i$.

Rollback should release all the resources consumed by the computation to be rolled-back which were provided by other threads. They are computed as follows.

[7] We extend the naïve semantics instead of the high-level or the low-level semantics (also defined in [19]) for the sake of simplicity. However, reduction semantics corresponding to the high-level and low-level semantics of roll-π can similarly be specified.

Definition 4 (Projection). *Let M be a configuration such that:* $M \equiv \nu\tilde{u}.\, \prod_{i\in I}(k_i : P_i) \mid \prod_{j\in J}[k'_j : R_j \mid k''_j : T_j; k_j] \mid \prod_{l\in L} k_l \prec (k'_l, k''_l)$ *with* $k \notin \tilde{u}$. *Then:*

$$M \not\downarrow_k = \nu\tilde{u}.\, \Big(\prod_{j'\in J'} k'_{j'} : R_{j'}\Big) \mid \Big(\prod_{j''\in J''} k''_{j''} : T_{j''}\Big)$$

where $J' = \{j \in J \mid k \not<: k'_j\}$ *and* $J'' = \{j \in J \mid k \not<: k''_j\}$.

Intuitively, $M \not\downarrow_k$ consists of the threads inside memories in M which are not dependent on k.

Finally, and this is the main novelty of croll-π, function `xtr` defined below replaces messages from the memory targeted by the roll by their alternatives.

Definition 5 (Extraction function).

$$\texttt{xtr}(M \mid N) = \texttt{xtr}(M) \mid \texttt{xtr}(N) \qquad \texttt{xtr}(k : a\langle P\rangle \div C) = k : C$$
$$\texttt{xtr}(k : a(X) \triangleright_\gamma Q) = k : a(X) \triangleright_\gamma Q$$

No other case needs to be taken into account as `xtr` is only called on the contents of memories.

Remark 3. Not all syntactically licit configurations make sense. In particular, we expect configurations to respect the causal information required for executing croll-π programs. We therefore work only with *coherent* configurations. A configuration is coherent if it is obtained by reduction starting from a configuration of the form $\nu k.\, k : P$ where P is closed and contains no roll h primitive (all the roll primitives should be of the form roll γ).

3.3 Barbed Congruence

We define notions of strong and weak barbed congruence to reason about croll-π processes and configurations. Name a is *observable* in configuration M, denoted as $M\downarrow_a$, if $M \equiv \nu\tilde{u}.\, (k : a\langle P\rangle \div C) \mid N$, with $a \notin \tilde{u}$. We write $M\mathcal{R}\downarrow_a$, where $\mathcal{R}$ is a binary relation on configurations, if there exists N such that $M\mathcal{R}N$ and $N\downarrow_a$. The following definitions are classical.

Definition 6 (Barbed congruences for configurations). *A relation* $\mathcal{R} \subseteq \mathcal{C}_{cl} \times \mathcal{C}_{cl}$ *on closed configurations is a* strong *(respectively* weak*)* barbed simulation *if whenever* $M\,\mathcal{R}\,N$,

- $M\downarrow_a$ *implies* $N\downarrow_a$ *(respectively* $N \Longrightarrow\downarrow_a$*);*
- $M \longrightarrow M'$ *implies* $N \longrightarrow N'$ *(respectively* $N \Longrightarrow N'$*) with* $M'\mathcal{R}N'$.

A relation $\mathcal{R} \subseteq \mathcal{C}_{cl} \times \mathcal{C}_{cl}$ *is a* strong (weak) barbed bisimulation *if* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are strong (weak) barbed simulations. We call* strong (weak) barbed bisimilarity *and denote by* $\sim$ $(\approx)$ *the largest strong (weak) barbed bisimulation. The largest congruence for configuration contexts included in* $\sim$ $(\approx)$ *is called* strong (weak) barbed congruence*, denoted by* $\sim_c$ $(\approx_c)$.

The notion of strong and weak barbed congruence extends to closed and open processes, by considering general contexts that form closed configurations.

Definition 7 (Barbed congruences for processes). *A relation $\mathcal{R} \subseteq \mathcal{P}_{cl} \times \mathcal{P}_{cl}$ on closed processes is a* strong *(resp.* weak*) barbed congruence if whenever $P\mathcal{R}Q$, for all general contexts $\mathbb{G}$ such that $\mathbb{G}[P]$ and $\mathbb{G}[Q]$ are closed configurations, we have $\mathbb{G}[P] \sim_c \mathbb{G}[Q]$ (resp. $\mathbb{G}[P] \approx_c \mathbb{G}[Q]$).*

Two open processes P and Q are said to be strong (resp. weak) barbed congruent, denoted by $P \sim_c^o Q$ (resp. $P \approx_c^o Q$) if for all substitutions σ such that $P\sigma$ and $Q\sigma$ are closed, we have $P\sigma \sim_c Q\sigma$ (resp. $P\sigma \approx_c Q\sigma$).

Working with arbitrary contexts can quickly become unwieldy. We offer the following Context Lemma to simplify the proofs of congruence.

Theorem 1 (Context lemma). *Two processes P and Q are weak barbed congruent, $P \approx_c^o Q$, if and only if for all substitutions σ such that $P\sigma$ and $Q\sigma$ are closed, all closed configurations M, and all keys k, we have: $M \mid (k : P\sigma) \approx M \mid (k : Q\sigma)$.*

The proof of this Context Lemma is much more involved than the corresponding one in the π-calculus, notably because of the bookkeeping required in dealing with process and thread tags. It is obtained by composing the lemmas below.

The first lemma shows that the only relevant configuration contexts are parallel contexts.

Lemma 2 (Context lemma for closed configurations). *For any closed configurations M, N, $M \sim_c N$ if and only if, for all closed configurations L, $M \mid L \sim N \mid L$. Likewise, $M \approx_c N$ if and only if, for all L, $M \mid L \approx N \mid L$.*

Proof. The left to right implication is immediate, by definition of $\sim_c$. For the other direction, the proof consists in showing that $\mathcal{R} = \{\langle \mathbb{C}[M], \mathbb{C}[N] \rangle \mid \forall L, M \mid L \sim N \mid L\}$ is included in $\sim$. The weak case is identical to the strong one. □

We can then prove the thesis on closed processes.

Lemma 3 (Context lemma for closed processes). *Let P and Q be closed processes. We have $P \approx_c Q$ if and only if, for all closed configuration contexts $\mathbb{C}$ and $k \notin \mathtt{fn}(P, Q)$, we have $\mathbb{C}[k : P] \approx \mathbb{C}[k : Q]$.*

Proof. The left to right implication is clear. One can prove the right to left direction by induction on the form of general contexts for processes, using the factoring lemma below for message contexts. □

Lemma 4 (Factoring). *For all closed processes P, all closed configurations M such that $M\{^P/_X\}$ is closed, and all $c, t, k, k' \notin \mathtt{fn}(M, P)$, we have*

$$M\{^P/_X\} \approx_c \nu c, t, k_0, k_0'.\, M\{^{\overline{c}}/_X\} \mid k_0 : t\langle Y_P \rangle \mid k_0' : Y_P$$

where $Y_P = t(Y) \triangleright (c \triangleright P) \mid t\langle Y \rangle \mid Y$.

We then deal with open processes.

Lemma 5 (Context lemma for open processes). *Let P and Q be (possibly open) processes. We have $P \approx_c^o Q$ if and only if for all closed configuration contexts $\mathbb{C}$, all substitutions σ such that $P\sigma$ and $Q\sigma$ are closed, and all $k \notin \mathtt{fn}(P, Q)$, we have $\mathbb{C}[k : P\sigma] \approx \mathbb{C}[k : Q\sigma]$.*

Proof. For the only if part, one proceeds by induction on the number of bindings in σ. The case for zero bindings follows from Lemma 3. For the inductive case, we write $\mathbb{P}[\bullet]$ for a process where an occurrence of $\mathbf{0}$ has been replaced by $\bullet$, and we show that contexts of the form $\mathbb{P} = a\langle R\rangle \mid a(X) \triangleright \mathbb{P}'[\bullet]$ where a is fresh and $\mathbb{P} = a\langle R\rangle \mid a(X) \triangleright_\gamma \mathbb{P}'[\bullet]$ where a is fresh and X never occurs in the continuation actually enforce the desired binding.

For the if part, the proof is by induction on the number of triggers. If the number of triggers is 0 then the thesis follows from Lemma 3. The inductive case consists in showing that equivalence under substitutions ensures equivalence under a trigger context. □

Proof (of Theorem 1). A direct consequence of Lemma 5 and Lemma 2. □

4 croll-π Expressiveness

4.1 Alternative Idioms

The message with alternative $a\langle P\rangle \div C$ triggers alternative C upon rollback. We choose to restrict C to be either a message with $\mathbf{0}$ alternative or $\mathbf{0}$ itself in order to have a minimal extension of roll-π. However, this simple form of alternative is enough to encode far more complex alternative policies and constructs, as shown below. We define the semantics of the alternative idioms below by only changing function `xtr` in Definition 5. We then encode them in croll-π and prove the encoding correct w.r.t. weak barbed congruence. More precisely, for every extension below the notion of barbs is unchanged. The notion of barbed bisimulation thus relates processes with slightly different semantics (only `xtr` differs) but sharing the same notion of barbs. Since we consider extensions of croll-π, in weak barbed congruence we consider just closure under croll-π contexts. By showing that the extensions have the same expressive power of croll-π, we ensure that allowing them in contexts would not change the result. Every encoding maps unmentioned constructs homomorphically to themselves. After having defined each alternative idiom, we freely use it as an abbreviation.

Arbitrary alternatives. Messages with arbitrary alternative can be defined by allowing C to be any process Q. No changes are required to the definition of function `xtr`. We can encode arbitrary alternatives as follows, where c is not free in P, Q.

$$(\!|a\langle P\rangle \div Q|\!)_{aa} = \nu c.\, a\langle (\!|P|\!)_{aa}\rangle \div c\langle (\!|Q|\!)_{aa}\rangle \div \mathbf{0} \mid c(X) \triangleright X$$

Proposition 1. *$P \approx_c (\!|P|\!)_{aa}$ for any closed process with arbitrary alternatives.*

$$\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \cup \mathcal{R}_4 \cup \mathcal{R}_5 \cup Id$$
$$\mathcal{R}_1 = \{\langle k : a\langle P\rangle \div Q \mid L\ ,\ k : (\nu c.\, a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid c(X) \triangleright X) \mid L\rangle\}$$
$$\mathcal{R}_2 = \{\langle k : a\langle P\rangle \div Q \mid L\ ,\ \nu c\, k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid k_2 : c(X) \triangleright X \mid L\rangle\}$$
$$\mathcal{R}_3 = \{\langle \nu h.\, [k : a\langle P\rangle \div Q \mid k' : a(X) \triangleright_\gamma R; h] \mid L''\ ,$$
$$\nu c\, k_1\, k_2\, h.\, k \prec (k_1, k_2) \mid [k_1 : a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid k' : a(X) \triangleright_\gamma R; h] \mid k_2 : c(X) \triangleright X \mid L''\rangle\}$$
$$\mathcal{R}_4 = \{\langle k : Q \mid L'''\ ,\ \nu c\, k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : c\langle Q\rangle \div \mathbf{0} \mid k_2 : c(X) \triangleright X \mid L'''\rangle\}$$
$$\mathcal{R}_5 = \{\langle k : Q \mid L'''\ ,\ \nu c\, k_1\, k_2\, h.\, k \prec (k_1, k_2) \mid [k_1 : c\langle Q\rangle \div \mathbf{0} \mid k_2 : c(X) \triangleright X; h] \mid h : Q \mid L'''\rangle\}$$

Fig. 4. Bisimulation relation for arbitrary alternatives.

Proof. We consider just one instance of arbitrary alternative, the thesis will follow by transitivity.

Thanks to Lemma 5 and Lemma 2, we only need to prove that for all closed configurations L and $k \notin \mathtt{fn}(P)$, we have $k : a\langle P\rangle \div Q \mid L \approx k : (\nu c.\, a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid c(X) \triangleright X) \mid L$. We consider the relation $\mathcal{R}$ in Figure 4 and prove that it is a weak barbed bisimulation. In every relation, L is closed and $k \notin \mathtt{fn}(P)$.

In $\mathcal{R}_1$, the right configuration can reduce via rule S.TagN followed by S.TagP. These lead to $\mathcal{R}_2$. Performing these reductions is needed to match the barb and the relevant reductions of the left configuration, thus we consider directly $\mathcal{R}_2$. In $\mathcal{R}_2$ the barbs coincide. Rollbacks lead to the identity. The only possible communication is on a, and requires $L \equiv L' \mid k' : a(X) \triangleright_\gamma R$. It leads to $\mathcal{R}_3$, where $L'' = L' \mid R\{^{P,h}/_{X,\gamma}\}$. In $\mathcal{R}_3$ the barbs coincide too. All the reductions can be matched by staying in $\mathcal{R}_3$ or going to the identity, but for executing a roll with key h. This leads to $\mathcal{R}_4$. From $\mathcal{R}_4$ we can always execute the internal communication at c leading to $\mathcal{R}_5$. The thesis follows from the result below, whose proof requires again to find a suitable bisimulation relation.

Lemma 6. *For each configuration M k-dependent and complete such that k', t, $k_1, k_2 \notin \mathtt{fn}(M)$ we have $M \approx_c \nu k'\, t\, k_1\, k_2.\, k \prec (k_1, k_2) \mid [k_1 : t\langle Q\rangle \div C \mid k_2 : t(X) \triangleright R; k'] \mid M\{^{k'}/_{k}\}$.*

□

Proofs concerning other idioms follow similar lines, and can be found in the online technical report [18].

A particular case of arbitrary alternative $a\langle P\rangle \div Q$ is when Q is a message whose alternative is not $\mathbf{0}$. By applying this pattern recursively we can write $a_1\langle P_1\rangle \div \ldots \div a_n\langle P_n\rangle \div Q$. In particular, by choosing $a_1 = \cdots = a_n$ and $P_1 = \cdots = P_n$ we can try n times the alternative P before giving up by executing Q.

Endless retry. We can also retry the same alternative infinitely many times, thus obtaining the behavior of roll-π messages. These messages can be integrated into croll-π semantics by defining function `xtr` as the identity on them.

$$(\!|a\langle P\rangle|\!)_{er} = \nu t.\, Y \mid a\langle (\!|P|\!)_{er}\rangle \div t\langle Y\rangle \qquad Y = t(Z) \triangleright Z \mid a\langle (\!|P|\!)_{er}\rangle \div t\langle Z\rangle$$

Proposition 2. *$P \approx_c (\!|P|\!)_{er}$ for any closed process with roll-π messages.*

As corollary of Proposition 2 we thus have the following.

Corollary 1. *croll-π is a conservative extension of roll-π.*

Triggers with alternative. Until now we attached alternatives to messages. Symmetrically, one may attach alternatives to triggers. Thus, upon rollback, the message is released and the trigger is replaced by a new process.

The syntax for triggers with alternative is $(a(X) \triangleright_\gamma Q) \div b\langle Q' \rangle \div \mathbf{0}$. As for messages, we use a single message as alternative, but one can use general processes as described earlier. Triggers with alternative are defined by the extract clause below.

$$\mathtt{xtr}(k : (a(X) \triangleright_\gamma Q) \div b\langle Q' \rangle \div \mathbf{0}) = k : b\langle Q' \rangle \div \mathbf{0}$$

Interestingly, messages with alternative and triggers with alternative may coexist. The encoding of triggers with alternative is as follows.

$$(\!|(a(X) \triangleright_\gamma Q) \div b\langle Q' \rangle \div \mathbf{0}|\!)_{at} = \nu c\, d.\, \overline{c} \div \overline{d} \div \mathbf{0} \mid (c \triangleright_\gamma a(X) \triangleright (\!|Q|\!)_{at}) \mid (d \triangleright b\langle (\!|Q'|\!)_{at} \rangle \div \mathbf{0})$$

Proposition 3. *$P \approx_c (\!|P|\!)_{at}$ for any closed process with triggers with alternative.*

4.2 Comparing croll-π and roll-π

While Corollary 1 shows that croll-π is at least as expressive as roll-π, a natural question is whether croll-π is actually strictly more expressive than roll-π or not. The theorem below gives a positive answer to this question.

Theorem 2. *There is no encoding $(\!|\bullet|\!)$ from croll-π to roll-π such that for each croll-π configuration M:*

1. *if M has a computation including at least a backward step, then $(\!|M|\!)$ has a computation including at least a backward step;*
2. *if M has only finite computations, then $(\!|M|\!)$ has only finite computations.*

Proof. Consider configuration $M = \nu k.\, k : \overline{a} \div \overline{b} \div \mathbf{0} \mid a \triangleright_\gamma \mathsf{roll}\ \gamma$. This configuration has a unique possible computation, composed by one forward step followed by one backward step. Assume towards a contradiction that an encoding exists and consider $(\!|M|\!)$. $(\!|M|\!)$ should have at least a computation including a backward step. From roll-π loop lemma [19, Theorem 1], if we have a backward step, we are able to go forward again, and then there is a looping computation. This is in contrast with the second condition of the encoding. The thesis follows. □

The main point behind this result is that the Loop Lemma, a cornerstone of roll-π theory [19] capturing the essence of rigid rollback (and similar results in [9, 20, 22, 24]), does not hold in croll-π. Naturally, the result above does not imply that croll-π cannot be encoded in HOπ or in π-calculus. However, these calculi are too low level for us, as hinted at by the fact that the encoding of a simple reversible higher order calculus into HOπ is quite complex, as shown in [20].

$$
\begin{aligned}
Q_i \;\triangleq\; & (act_i(z_i) \triangleright p_i\langle i,1\rangle \div \ldots \div p_i\langle i,8\rangle \div f_i\langle \mathbf{0}\rangle \div \mathbf{0} \mid \\
& (p_i(\mathbf{x_i}) \triangleright_{\gamma_i} act_{i+1}\langle \mathbf{0}\rangle \mid f_{i+1}(z) \triangleright \mathsf{roll}\ \gamma_i \mid ok_i(w_1) \triangleright \ldots ok_i(w_{i-1}) \triangleright !c_i\langle \mathbf{x_i}\rangle \div \mathbf{0} \mid \\
& \textstyle\prod_{j=1}^{i-1} c_j(\mathbf{y_j}) \triangleright \mathtt{if}\ err(\mathbf{x_i},\mathbf{y_j})\ \mathtt{then}\ \mathsf{roll}\ \gamma_i\ \mathtt{else}\ ok_i\langle \mathbf{0}\rangle \div \mathbf{0})) \\
err((x_1,x_2),(y_1,y_2)) \;\triangleq\; & (x_1 = y_1 \vee x_2 = y_2 \vee |x_1 - y_1| = |x_2 - y_2|)
\end{aligned}
$$

Fig. 5. The i-th queen

5 Programming in croll-π

A main goal of croll-π is to make reversibility techniques exploitable for application development. Even if croll-π is not yet a full-fledged language, we have developed a proof-of-concept interpreter for it. To the best of our knowledge, this is the first interpreter for a causal-consistent reversible language. We then put the implementation at work on a few simple, yet interesting, programming problems. We detail below the algorithm we devised to solve the Eight Queens problem [4, p. 165]. The interpreter and the code for solving the Eight Queens problem are available at `http://proton.inrialpes.fr/~mlienhar/croll-pi/implem`, together with examples of encodings of primitives for error handling, and an implementation of the car repair scenario of the EU project Sensoria.

The interpreter for croll-π is written in Maude [11], a language based on both equational and rewriting logic that allows the programmer to define terms and reduction rules, e.g., to execute reduction semantics of process calculi. Most of croll-π's rules are straightforwardly interpreted, with the exception of rule S.Roll. This rule is quite complex as it involves checks on an unbounded number of interacting components. Such an issue is already present in roll-π [19], where it is addressed by providing an easier to implement, yet equivalent, low-level semantics. This semantics replaces rule S.Roll with a protocol that sends notifications to all the involved components to roll-back, then waits for them to do so. Extending the low-level semantics from roll-π to croll-π simply requires the application of function `xtr` to the memory targeted by the rollback. We do not detail the low-level semantics of croll-π here, and refer the reader to [19] for a detailed description in the setting of roll-π. Our Maude interpreter is based on this low-level semantics, extended with values (integers and pairs) and with the `if-then-else` construct. It is fairly concise (less than 350 lines of code).

The Eight Queens problem is a well-known constraint-programming probem which can be formulated as follows: how to place 8 queens on an 8×8 chess board so that no queen can directly capture another? We defined an algorithm in croll-π where queens are concurrent entities, numbered from 1 to 8, all executing the code schema shown in Figure 5. We use $\mathbf{x}$ to indicate a pair of integer variables (x_1, x_2), and replicated messages $!c_i\langle \mathbf{x}\rangle \div \mathbf{0}$ to denote the encoding of a parallel composition of an infinite number of messages $c_i\langle \mathbf{x}\rangle \div \mathbf{0}$ (cf. Remark 1).

The queens are activated in numeric order. The i-th queen is activated by messages on channels act_i from its predecessor. When a queen is activated it

looks for its position by trying sequentially all the positions in the i-th row of the chess board. To try a position, it sends it over channel p_i and checks whether the position conflicts with the choices of the other queens. This is done by computing (in parallel) $err(\mathbf{x_j}, \mathbf{x_i})$ for each $j < i$. If a check fails, $\mathsf{roll}\ \gamma_i$ rolls-back the choice of the position. The alternatives mechanism allows to try the next position. If no suitable position is available, the choice of position of the previous queen is rolled-back (possibly recursively) by the communication over f_i. If instead the check succeeds, it generates a message on channel ok_i. When there are exactly $i-1$ messages on the channel ok_i, the queen commits its position on c_i.

6 Asynchronous Interacting Transactions

This section shows how croll-π can model in a precise way interacting transactions with compensations as formalized in TransCCS [14]. Actually, the natural croll-π encoding improves on the semantics in [14], since croll-π causality tracking is more precise than the one in TransCCS, which is based on dynamic embedding of processes into transactions. Thus croll-π avoids some spurious undo of actions, as described below. Before entering the details of TransCCS, let us describe the general idea of transaction encoding.

We consider a very general notion of atomic (but not necessarily isolated) transaction, i.e., a process that executes completely or not at all. Informally, a transaction $[P,Q]_\gamma$ with name γ executing process P with compensation Q can be modeled by a process of the form:

$$[P,Q]_\gamma = \nu a\, c.\, \overline{a} \div \overline{c} \div \mathbf{0} \mid (a \triangleright_\gamma P) \mid (c \triangleright Q)$$

Intuitively, when $[P,Q]_\gamma$ is executed, it first starts process P under the rollback scope γ. Abortion of the transaction can be triggered in P by executing a $\mathsf{roll}\ \gamma$. Whenever P is rolled-back, the rollback does not restart P (since the message on a is substituted by the alternative on c), but instead starts the compensation process Q. In this approach commit is implicit: when there is no reachable $\mathsf{roll}\ \gamma$, the transaction is committed. From the explanation above, it should be clear that in the execution of $[P,Q]_\gamma$, either P executes completely, i.e., until it reaches a commit, or not at all, in the sense that it is perfectly rolled-back. If P is ever rolled-back, its failed execution can be compensated by that of process Q. Interestingly, and in contrast with irreversible actions used in [13], our rollback scopes can be nested without compromising this all-or-nothing semantics.

Let us now consider an asynchronous fragment of TransCCS [14], removing choice and recursion. Dealing with the whole calculus would not add new difficulties related to rollback, but only related to the encoding of such operators in higher-order π. The syntax of the fragment of TransCCS we consider is:

$$P ::= \ \mathbf{0} \ \mid\ \nu a.\, P \ \mid\ (P \mid Q) \ \mid\ \overline{a} \ \mid\ a.P \ \mid\ \mathsf{co}\ k \ \mid\ [\![P \triangleright_k Q]\!]$$

Essentially, it extends CCS with a transactional construct $[\![P \triangleright_k Q]\!]$, executing a transaction with body P, name k and compensation Q, and a commit operator $\mathsf{co}\ k$.

$$\text{(R-Comm)}\quad \overline{a} \mid a.P \longrightarrow P \qquad \text{(R-Emb)}\quad \frac{k \notin \mathtt{fn}(R)}{[\![P \rhd_k Q]\!] \mid R \longrightarrow [\![P \mid R \rhd_k Q \mid R]\!]}$$

$$\text{(R-Co)}\quad [\![P \mid \mathsf{co}\ k \rhd_k Q]\!] \longrightarrow P \qquad \text{(R-Ab)}\quad [\![P \rhd_k Q]\!] \longrightarrow Q$$

and is closed under active contexts $\nu a.\, \bullet$, $\bullet \mid Q$ and $[\![\bullet \rhd_k Q]\!]$, and structural congruence.

Fig. 6. Reduction rules for TransCCS

The rules defining the semantics of TransCCS are given in Figure 6. Structural congruence contains the usual rules for parallel composition and restriction. Keep in mind that transaction scope is a binder for its name k, thus k does not occur outside the transaction, and there is no name capture in rules R-Co and R-Emb.

A croll-π transaction $[P, Q]_\gamma$ as above has explicit abort, specified by $\mathsf{roll}\ \gamma$, where γ is used as the transaction name, and implicit commit. TransCCS takes different design choices, using non-deterministic abort and programmable commit. Thus we have to instantiate the encoding above.

Definition 8 (TransCCS encoding). *Let P be a TransCCS process. Its encoding $(\!|\bullet|\!)_t$ in croll-π is defined as:*

$$(\!|\nu a.\, P|\!)_t = \nu a.\, (\!|P|\!)_t \qquad (\!|P \mid Q|\!)_t = (\!|P|\!)_t \mid (\!|Q|\!)_t \qquad (\!|\overline{a}|\!)_t = \overline{a}$$
$$(\!|a.P|\!)_t = a \rhd (\!|P|\!)_t \qquad (\!|\mathsf{co}\ l|\!)_t = l(X) \rhd \mathbf{0} \qquad (\!|\mathbf{0}|\!)_t = \mathbf{0}$$

$$(\!|[\![P \rhd_l Q]\!]|\!)_t = [\nu l.\, (\!|P|\!)_t \mid l\langle \mathsf{roll}\ \gamma\rangle \mid l(X) \rhd X, (\!|Q|\!)_t]_\gamma$$

Since in croll-π only configurations can execute, the behavior of P should be compared with $\nu k.\, k : (\!|P|\!)_t$.

In the encoding, abort is always possible since at any time the only occurrence of the roll in the transaction can be activated by a communication on l. On the other hand, executing the encoding of a TransCCS commit disables the roll related to the transaction. This allows to garbage collect the compensation, and thus corresponds to an actual commit. Note, however, that in croll-π the abort operation is not atomic as in TransCCS since the roll related to a transaction first has to be enabled through a communication on l, disabling in this way any possibility to commit, and then it can be executed. Clearly, until the roll is executed, the body of the transaction can continue its execution. To make abort atomic one would need the ability to disable an active roll, as could be done using a (mixed) choice such as $(\mathsf{roll}\ k) + (l \rhd \mathbf{0})$. In this setting an output on l would commit the transaction. Adding choice would not make the reduction semantics more difficult, but its impact on behavioral equivalence has not been studied yet.

The relation between the behavior of a TransCCS process P and of its translation $(\!|P|\!)_t$ is not immediate, not only because of the comment above on atomicity, but also because of the approximate tracking of causality provided by TransCCS.

TransCCS tracks interacting processes using rule (R-Emb): only processes inside the same transaction may interact, and when a process enters the transaction it is saved in the compensation, so that it can be restored in case of abort. However, no check is performed to ensure that the process actually interacts with the transaction code. For instance, a process $\overline{a} \mid a.P$ may enter a transaction $[\![Q \triangleright_k R]\!]$ and then perform the communication at a. Such a communication would be undone in case of abort. This is a spurious undo, since the communication at a is not related to the transaction code. Actually, the same communication could have been performed outside the transaction, and in this case it would not have been undone.

In croll-π encoding, a process is "inside" the transaction with key k if and only if its tag is causally dependent on k. Thus a process enters a transaction only by interacting with a process inside it. For this reason, there is no reduction in croll-π corresponding to rule (R-Emb), and since no process inside the transaction is involved in the reduction at a above, the reduction would not be undone in case of abort, since it actually happens "outside" the transaction. Thus our encoding avoids spurious undo, and computations in croll-π correspond to computations in TransCCS with minimal applications of rule (R-Emb). These computations are however very difficult to characterize because of syntactic constraints. In fact, for two processes inside two parallel transactions k_1 and k_2 to interact, either k_1 should move inside k_2 or vice versa, but in both the cases not only the interacting processes move, as minimality would require, but also all the other processes inside the same transactions have to move. Intuitively, TransCCS approximates the causality relation, which is a dag, using the tree defined by containment. The spurious reductions undone in TransCCS can always be redone so to reach a state corresponding to the croll-π one. In this sense croll-π minimizes the set of interactions undone.

We define a notion of weak barbed bisimilarity ${}_t\approx_{c\pi}$ relating a TransCCS process P and a croll-π configuration M. First, we define barbs in TransCCS by the predicate $P{\downarrow_a}$, which is true in the cases below, false otherwise.

$$\begin{array}{ll} \overline{a}{\downarrow_a} & \nu b.\, P{\downarrow_a} \text{ if } P{\downarrow_a} \wedge a \neq b \\ P \mid P'{\downarrow_a} \text{ if } P{\downarrow_a} \vee P'{\downarrow_a} & [\![P \triangleright_k Q]\!]{\downarrow_a} \text{ if } P{\downarrow_a} \wedge a \neq k \end{array}$$

Here, differently from [14], we observe barbs inside the transaction body, to have a natural correspondence with croll-π barbs.

Definition 9. *A relation $\mathcal{R}$ relating TransCCS processes P and croll-π configurations M is a weak barbed bisimulation if and only if for each $(P, M) \in \mathcal{R}$:*

1. *if $P{\downarrow_a}$ then $M \Longrightarrow{\downarrow_a}$;*
2. *if $M{\downarrow_a}$ then $P \Longrightarrow{\downarrow_a}$;*
3. *if $P \longrightarrow P_1$ is derived using rule* (R-Ab) *then $M \Longrightarrow M'$, $P_1 \Longrightarrow P_2$ and $P_2 \mathcal{R} M'$;*
4. *if $P \longrightarrow P_1$ is derived without using rule* (R-Ab) *then $M \Longrightarrow M'$ and $P_1 \mathcal{R} M'$;*
5. *if $M \longrightarrow M'$ then either: (i) $P \mathcal{R} M'$ or (ii) $P \longrightarrow P_1$ and $P_1 \mathcal{R} M'$ or (iii) $M' \longrightarrow M''$, $P \longrightarrow P_1$ and $P_1 \mathcal{R} M''$.*

Weak barbed bisimilarity ${}_t\approx_{c\pi}$ *is the largest weak barbed bisimulation.*

The main peculiarities of the definition above are in condition 3, which captures the need of redoing some reductions that are unduly rolled-back in TransCCS, and in case (iii) of condition 5, which forces atomic abort.

Theorem 3. *For each TransCCS process* P, $P\ {}_t\approx_{c\pi}\ \nu k.\, k : (\!|P|\!)_t$.

Proof. The proof has to take into account the fact that different croll-π configurations may correspond to the same TransCCS process. In particular, a TransCCS transaction $[\![P \triangleright_k Q]\!]$ is matched in different ways if Q is the original compensation or if part of it is the result of an application of rule (R-Emb).

Thus, in the proof, we give a syntactic characterization of the set of croll-π configurations $(\!|P|\!)^p$ matching a TransCCS process P. Then we show that $\nu k.\, k : (\!|P|\!)_t \in (\!|P|\!)^p$, and that there is a match between reductions of P and the weak reductions of each configuration in $(\!|P|\!)^p$. The proof, in the two directions, is by induction on the rule applied to derive a single step. □

7 Related Work and Conclusion

We have presented a concurrent process calculus with explicit rollback and minimal facilities for alternatives built on a reversible substrate analogous to a Lévy labeling [5] for concurrent computations. We have shown by way of examples how to build more complex alternative idioms and how to use rollback and alternatives in conjunction to encode transactional constructs. In particular, we have developed an analysis of communicating transactions proposed in TransCCS [14]. We also developed a proof-of-concept interpreter of our language and used it to give a concurrent solution of the Eight Queens problem.

Undo or rollback capabilities in sequential languages have a long history (see [21] for an early survey). In a concurrent setting, interest has developed more recently. Works such as [10] introduce logging and process group primitives as a basis for defining fault-tolerant abstractions, including transactions. Ziarek et al. [28] introduce a checkpoint abstraction for concurrent ML programs. Field et al. [16] extend the actor model with checkpointing constructs. Most of the approaches relying instead on a fully reversible concurrent language have already been discussed in the introduction. Here we just recall that models of reversible computation have also been studied in the context of computational biology, e.g., [9]. Also, the effect of reversibility on Hennessy-Milner logic has been studied in [25]. Several recent works have proposed a formal analysis of transactions, including [14] studied in this paper, as well as several other works such as [23, 6, 8] (see [1] for numerous references to the line of work concentrating on software transactional memories). Note that although reversible calculi can be used to implement transactions, they offer more flexibility. For instance, transactional events [15] only allow an all-or-nothing execution of transactions. Moreover, no visible side-effect is allowed during the transaction, as there is no way to specify how to compensate the side-effects of a failed transaction. A reversible calculus with alternatives allows the encoding of such compensations.

With the exception of the seminal work by Danos and Krivine [13] on RCCS, we are not aware of other work exploiting precise causal information as provided by our reversible machinery to analyze recovery-oriented constructs. Yet this precision seems important: as we have seen in Section 6, it allows us to weed out spurious undo of actions that appear in an approach that relies on a cruder transaction "embedding" mechanism. Although we have not developed a formal analysis yet, it seems this precision would be equally important, e.g., to avoid uncontrolled cascading rollbacks (domino effect) in [28] or to ensure that, in contrast to [16], rollback is always possible in failure-free computations. Although [10] introduces primitives able to track down causality information among groups of processes, called conclaves, it does not provide automatic support for undoing the effects of aborted conclaves, while our calculus directly provides a primitive to undo all the effects of a communication.

While encouraging, our results in Section 6 are only preliminary. Our concurrent rollback and minimal facilities for alternatives provide a good basis for understanding the "all-or-nothing" property of transactions. To this end it would be interesting to understand whether we are able to support both strong and weak atomicity of [23]. How to support isolation properties found, e.g., in software transactional memory models, in a way that combines well with these facilities remains to be seen. Further, we would like to study the exact relationships that exist between these facilities and the different notions of compensation that have appeared in formal models of computation for service-oriented computing, such as [6, 8]. It is also interesting to compare with zero-safe Petri nets [7], since tokens in zero places dynamically define transaction scopes as done by communications in croll-π.

From a practical point of view, we want both to refine the interpreter, and to test it against a wider range of more complex case studies. Concerning the interpreter, a main point is to allow for garbage collection of memories which cannot be restored any more, so to improve space efficiency.

References

[1] M. Abadi and T. Harris. Perspectives on transactional memory. In *CONCUR'09*, volume 5710 of *LNCS*. Springer, 2009.
[2] L. Acciai, M. Boreale, and S. Dal-Zilio. A concurrent calculus with atomic transactions. In *ESOP'07*, volume 4421 of *LNCS*. Springer, 2007.
[3] G. Bacci, V. Danos, and O. Kammar. On the statistical thermodynamics of reversible communicating processes. In *CALCO 2011*, volume 6859 of *LNCS*, 2011.
[4] W. W. Rouse Ball. *Mathematical Recreations and Essays (12th ed.)*. Macmillan, New York, 1947.
[5] G. Berry and J.-J. Lévy. Minimal and optimal computations of recursive programs. *J. ACM*, 26(1), 1979.
[6] R. Bruni, H. C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL'05*. ACM, 2005.
[7] R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Information and Computation*, 156(1-2), 2000.

[8] M. J. Butler, C.A.R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years CSP*, number 3525 in LNCS. Springer, 2004.
[9] L. Cardelli and C. Laneve. Reversible structures. In *CMSB 2011*. ACM, 2011.
[10] T. Chothia and D. Duggan. Abstractions for fault-tolerant global computing. *Theor. Comput. Sci.*, 322(3), 2004.
[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comp. Sci.*, 285(2), 2002.
[12] V. Danos and J. Krivine. Reversible communicating systems. In *CONCUR'04*, volume 3170 of *LNCS*. Springer, 2004.
[13] V. Danos and J. Krivine. Transactions in RCCS. In *CONCUR'05*, volume 3653 of *LNCS*. Springer, 2005.
[14] E. de Vries, V. Koutavas, and M. Hennessy. Communicating transactions. In *CONCUR 2010*, volume 6269 of *LNCS*. Springer, 2010.
[15] K. Donnelly and M. Fluet. Transactional events. *Journal of Functional Programming*, 18(5–6), 2008.
[16] J. Field and C.A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *POPL'05*. ACM, 2005.
[17] T. Harris, S. Marlow, S. L. Peyton Jones, and M. Herlihy. Composable memory transactions. *Commun. ACM*, 51(8), 2008.
[18] I. Lanese, M. Lienhardt, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Concurrent flexible reversibility (TR). `http://www.cs.unibo.it/~lanese/publications/fulltext/TR-crollpi.pdf.gz`, 2012.
[19] I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling reversibility in higher-order pi. In *CONCUR 2011*, volume 6901 of *LNCS*. Springer, 2011.
[20] I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversing higher-order pi. In *CONCUR 2010*, volume 6269 of *LNCS*. Springer, 2010.
[21] G.B. Leeman. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8(1), 1986.
[22] M. Lienhardt, I. Lanese, C. A. Mezzina, and J.-B. Stefani. A reversible abstract machine and its space overhead. In *FMOODS/FORTE 2012*, volume 7273 of *LNCS*, 2012.
[23] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL'08*. ACM, 2008.
[24] I. Phillips and I. Ulidowski. Reversing algebraic process calculi. *J. Log. Algebr. Program.*, 73(1-2), 2007.
[25] I. Phillips and I. Ulidowski. A logic with reverse modalities for history-preserving bisimulations. In *EXPRESS 2011*, volume 64 of *EPTCS*, 2011.
[26] I. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In *Reversible Computation 2012*, volume 7581 of *LNCS*, 2012.
[27] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
[28] L. Ziarek and S. Jagannathan. Lightweight checkpointing for concurrent ML. *J. Funct. Program.*, 20(2), 2010.