Bounded Reversibility in HOpi

Bounded Reversibility in HO$\pi^\star$

Ivan Lanese[1], Claudio A. Mezzina[2], and Martin Vassor[3]

[1] Olas Team, University of Bologna & Inria - Université Côte d'Azur, Bologna, Italy
[2] Department of Pure and Applied Sciences, University of Urbino, Urbino, Italy
[3] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Abstract. The higher-order causally-consistent reversible π-calculus, known as **roll**$-\pi$, enables the rollback of arbitrary past actions while preserving causal consistency – ensuring that effects are undone before their causes. This prevents the occurrence of actions without justifications, even after a rollback. However, in practical scenarios, not all events can be reversed; for example, once a document is printed, it cannot be *unprinted*.

To better model real-world constraints, we introduce **broll**$-\pi$ (*bounded* **roll**$-\pi$), an extension of **roll**$-\pi$ that limits reversibility. Our approach imposes two key restrictions: *spatial bounds*, which prevent certain processes from being affected by rollbacks, and *temporal bounds*, which restrict how far back in time a rollback can go. Bounded reversibility allows for more realistic and controlled rollbacks in computational systems. In this paper, we provide an informal introduction to **broll**$-\pi$ and discuss its implications for modeling reversible processes in practical applications.

1 Introduction

Motivation. Reversibility attracted interest for many relevant application areas, including very fast simulations [3], promising low-energy digital circuits [7], and powerful debugging techniques [14,6]. An application area that attracted the interest of Jean-Bernard was the possibility of using reversibility for reliability [11,9,8,17]. The basic idea here is that in case of errors one could rollback to states before the error occurred, and retry or find an alternative solution. This triggered the introduction of rollback operators in process calculi [9], allowing one to undo an action possibly far in the past, including all and only its consequences. Reversible process calculi often aim at providing *sound* and *complete* rollbacks. Soundness states that rolling back reaches configurations that could be reached using forward-only reductions, while completeness relates to the fact that all events can be reverted (in particular, concurrent rollbacks do not prevent

* This work has been partially supported by French ANR project Smart-Cloud ANR-23-CE25-0012, by INdAM – GNCS 2025 project MARQ, code CUP_E53C24001950001, and by the project FREEDA (CUP: I53D23003550006), funded by the frameworks PRIN (MUR, Italy) and Next Generation EU. We thanks reviewers for their comments and suggestions.

each other). While very practical to reason on systems, those requirements do not fit practical applications. For instance:

- in a typical client-server architecture, clients and the server interact, hence their actions are causally related. Reverting a client would also cause to rollback the server, which in turn would rollback other clients.
- in a distributed algorithm, e.g. to implement consensus in a fail-stop model [2, Section 2.7 and Chapter 5], reversibility is very appealing, since it provides fault-tolerance quite transparently. However, in languages such as **roll**$-\pi$, processes can always revert to the beginning of the computation, hence there is no eventual termination[4].

The two examples above illustrate the difficulty of using reversible calculi such as **roll**$-\pi$ in practice: to reason on real world applications, we need notions of abstraction and composition; but reversibility, as it is implemented in **roll**$-\pi$, breaks boundaries of subsystems.

This paper introduces **broll**$-\pi$, for *bounded* **roll**$-\pi$, an extension of **roll**$-\pi$ with two novelties:

Spatial boundaries: The possibility to restrict rollback to part of the system only. While in **roll**$-\pi$ communication always establishes a bidirectional causal dependency between sender and receiver, hence rollback of one of them triggers the rollback of the other, this is not always the case in **broll**$-\pi$. Indeed, rollback propagates only if both the sender and the receiver agree to do so. This addresses the issue with client-server architectures described above: communications between clients and the server may not establish causality links; hence rollbacks would be contained to individual clients or to the server.

Temporal boundaries: The calculus includes a *commit* primitive. Upon commit, the committed past event cannot be undone any more, thus addressing the second problem we mentioned above. Memories related to committed events become useless, hence this approach allows one to limit the amount of history information that needs to be kept, thus limiting a main issue of reversible computing.

These advantages come at the price of breaking the nice theoretical framework built in [4,13], which assumes every action to be reversible forever. We believe that such a theory needs to be extended to cope with systems which are only partially reversible. This paper only adds a small block in this direction.

Outline. We begin by recalling **roll**$-\pi$ in Section 2. This is instrumental to better understand the rest of the paper, which builds on it. We then give an

[4] Even providing an implementation, e.g., a library, that is eventually free of rollback requests, and therefore that is eventually terminating, is not enough. Indeed, there is no way to ensure that the context has no rollback requests targeting states before the call of the library, thus invalidating eventual termination.

$$
\begin{array}{lll}
\mathcal{P},\mathcal{Q} & ::= 0 & \textit{Empty process} \\
& \mid X & \textit{Variable} \\
& \mid \nu c.\,(\mathcal{P}) & \textit{Channel restriction} \\
& \mid \mathcal{P} \parallel \mathcal{P} & \textit{Parallel composition} \\
& \mid c\,\langle \mathcal{P} \rangle & \textit{Sending message} \\
& \mid c(X) \triangleright_{\gamma} (\mathcal{P}) & \textit{Receiving message} \\
& \mid \texttt{roll}\ k \quad \mid \quad \texttt{roll}\ \gamma & \textit{Rollback primitive} \\
\mathcal{M},\mathcal{N} & ::= 0 & \textit{Empty configuration} \\
& \mid \nu u.\,(\mathcal{M}) & \textit{New identifier (name or key)} \\
& \mid \mathcal{M} \parallel \mathcal{N} & \textit{Parallel configuration} \\
& \mid \kappa : \mathcal{P} & \textit{Tagged process} \\
& \mid [\mu; k]^{\bullet} \quad \mid \quad [\mu; k] & \textit{(Marked) memory} \\
\kappa & ::= k \quad \mid \quad \langle h, \tilde{h} \rangle \cdot k & \textit{Simple/Complex tag} \\
\mu & ::= \kappa_1 : c\,\langle \mathcal{P} \rangle \parallel \kappa_2 : c(X) \triangleright_{\gamma} (Q) & \textit{Memory content}
\end{array}
$$

Grammar 1.1. Grammar of **roll**$-\pi$.

informal presentation of **broll**$-\pi$ in Section 3, where we also explain the main design choices to implement the two new mechanisms. Then, in Section 4, we introduce **broll**$-\pi$, its syntax and its semantics. We conclude the section with a few examples that illustrate the new features of **broll**$-\pi$. The new extensions we introduce allow us to broaden the scope of reversibility. In Section 5 we show how our framework can be used to implement a speculative consensus algorithm. In Section 6, we discuss the relations between rollback and commit. Section 7 discusses related work and Section 8 wraps up with some final considerations.

2 A recap of roll$-\pi$

Reversibility in **roll**$-\pi$ builds on the mechanisms introduced in $\rho\pi$ [10,11]. Both calculi extend **HO**π [15] with reversibility. In both **roll**$-\pi$ and $\rho\pi$ (as well as in **broll**$-\pi$), **HO**π processes do not execute directly, but are embedded in *configurations*: history on the past of the computation is recorded in *memories* and processes are associated with *tags* which uniquely identify processes. Memories also track causal dependencies, namely store which processes interact, and which are the tags of the processes generated by the interaction.

The syntax of **roll**$-\pi$ is given in Grammar 1.1. Channel names are ranged over by a, b, c, ... We note $[\mu; k]^{\circ}$ for $[\mu; k]$ or $[\mu; k]^{\bullet}$.

Processes of **roll**$-\pi$ are similar to processes of **HO**π: communications happen on channels (e.g. a) between a pending message $a\,\langle P \rangle$ and a receiver, also referred to as *trigger*, $a(X) \triangleright_{\gamma} (Q)$. The receiver binds the process variable X and the key variable γ in Q. Processes can be composed in parallel ($P \parallel Q$) and channel names can be restricted ($\nu a.\,(P)$ binds a in P). In addition to those primitives, **roll**$-\pi$ introduces a new rollback primitive $\texttt{roll}\ \gamma/\texttt{roll}\ k$, used to trigger rollbacks. Contrary to **HO**π, processes of **roll**$-\pi$ can not be executed directly: instead, they have to be embedded in *configurations*, which add *tags* (to keep track of causal dependencies) as well as *memories* (which record past

events). A tagged process is denoted by $k : P$, where k is the tag of process P. Tags can be restricted at the configuration level with $\nu k.\,(P)$. When a tagged process is a parallel composition (e.g. $k : P \mid\mid Q$), k can be split to tag separately P and Q. This leads to complex tags (e.g. $\langle k_1, \tilde{k}\rangle \cdot k : P \mid\mid \langle k_2, \tilde{k}\rangle \cdot k : P$, where $\tilde{k} = \{k_1, k_2\}$). Complex tags allow one to keep track of independent processes (e.g. P and Q) which depend on (i.e. which were created by) the same event. Finally, when a communication happens, a memory is created to store the state of processes involved in the communication, thus enabling rollbacks. A memory $[\mu; k]$ contains the processes in μ (which are always a pending message and a receiver) and a tag k to keep track of causal chains. For instance, the configuration $k_1 : c\langle \mathcal{P}\rangle \mid\mid k_2 : c(X) \triangleright_\gamma (Q)$ performs a communication and reduces to $k : Q\{^{P,k}/_{X,\gamma}\} \mid\mid [k_1 : c\langle \mathcal{P}\rangle \mid\mid k_2 : c(X) \triangleright_\gamma (Q); k]$. Notice how the created process and the memory share the same tag, thus storing the causal link.

While in $\rho\pi$ there is no policy specifying when to execute forwards and when to execute backwards, in **roll**$-\pi$ normal computation is forward, but processes may trigger a rollback using a dedicated primitive: `roll` k, where k refers to which memory to rollback. Two rollback semantics are provided in [9]: high-level (centralized) and low-level (distributed).

In the high-level semantics (which we focus on in this paper), the rollback happens atomically: in a single step, consequences of the event which created the memory to revert are removed and the configuration μ inside the target memory $[\mu; k]$ is re-instantiated.

Reversible communication. The reduction rule for **roll**$-\pi$ is as follows[5]:

$$(\textsc{Com})\ \frac{}{\kappa_1 : a\langle P\rangle \mid\mid \kappa_2 : a(X) \triangleright_\gamma (Q) \twoheadrightarrow \nu k.\,\big(k : Q\{^{P,k}/_{X,\gamma}\} \mid\mid [\kappa_1 : a\langle P\rangle \mid\mid \kappa_2 : a(X) \triangleright_\gamma (Q); k]\big)}$$

The main idea here is that the original configuration is stored inside the memory, together with a fresh tag k, linking the memory to the continuation of the trigger. The continuation of a memory, if composed by multiple parallel components, is eventually split using the following structural congruence rule:

$$(\text{E.TagP})\quad k : P \mid\mid Q = \nu h_1, h_2.\,\Big(\langle h_1, \tilde{h}\rangle \cdot k : P \mid\mid \langle h_2, \tilde{h}\rangle \cdot k : Q\Big)$$

where $\tilde{h} = \{h_1, h_2\}$ (this is the binary case, n-ary case follows the same idea).

Rolling back using **roll**$-\pi$ *high-level semantics.* In the high-level semantics of **roll**$-\pi$, a first rule (H.Start) marks the target memory to revert.

$$(\text{H.Start})\ [\mu; k] \mid\mid \kappa : \texttt{roll}\ k \twoheadrightarrow [\mu; k]^\bullet \mid\mid \kappa : \texttt{roll}\ k$$

Then, rollbacks are atomic (all memories are removed and replaced by the initial process at once). The rollback rule (H.Roll) is the following:

$$(\text{H.Roll})\ \frac{N \blacktriangleright k \qquad \text{complete}(N \mid\mid [\mu; k])}{N \mid\mid [\mu; k]^\bullet \twoheadrightarrow \mu \mid\mid N \not\downarrow_k}$$

[5] In [9], this rule is used both in the high-level semantics (named (H.Com)) and in the low-level one (named (L.Com)). We name it (Com) in this paper.

The two required predicates roughly mean that all processes and memories in N causally depend on k ($N \blacktriangleright k$) and that all successors of k are in N (complete($N \parallel [\mu; k]$)). $N \not\downarrow_k$ extracts from memories in N those messages or triggers which do not have k as a causal antecedent, but which participated in communications with causal descendants of k.

All the reductions mentioned above can happen inside evaluation contexts, in particular parallel composition and restriction contexts.

3 Informal presentation of broll$-\pi$

As mentioned in the Introduction, our goal is to set *spatial* and *temporal* boundaries on rollbacks. Those two kinds of boundaries are enforced by two independent mechanisms, and therefore we shall present them independently.

We first present *spatial boundaries*, allowing one to ensure that when two processes communicate, the rollback of one of them does not trigger the rollback of the other. We observe that, in **roll**$-\pi$, *memories* are responsible of rolling back both sides of a communication, since they record both the sender and the receiver. Thus restoring the memory restores both the sender and the receiver. Therefore, to decouple both sides of the communication, we modify the *content* of the memory, so that only one side is restored. In the case we present, messages do not have continuations, so they can not be rolled back. Therefore, we choose to only store the receiver process in the memory, dropping the message[6].

To allow the programmer to decide whether rollback should be propagated to the sender or not, we introduce two different usages of names: *bracketed* names (e.g. $\{a\}$) which specify that rollback should not be propagated, or normal (e.g., a). Notice that *usages* of names are normal or bracketed, not names themselves: a name can be used both normally and bracketed, albeit in different places.

Regarding *temporal boundaries*, we introduce a mechanism to make a past action irreversible, allowing one to eventually remove the corresponding memories. Similar to the `roll` k primitive, we introduce `commit` k, which commits memory tagged with k. Two points need attention:

1. if a memory is committed, all the previous memories (i.e. those that are causal ascendants of the target one) also need to be blocked (*commit consistency*);
2. commits need to be compatible with rollbacks, in particular when they are concurrent (*commit/rollback consistency*).

To deal with commit consistency, **broll**$-\pi$ adopts a spanning mechanism somewhat similar to that of the low level semantics of rollbacks (cf. [9]), but towards the past. When a memory is to be committed, we iteratively mark all causal ascendant memories as committed as well. Eventually, all marked memories are removed[7].

[6] In a variant with continuations, we would need to create two independent memories upon communication: one for the receiver, one for the sender.

[7] While garbage collecting memories is not the primary objective of this work, it is a nice side effect to have such a feature.

To deal with commit/rollback consistency, we prevent reverting memories that are marked for commit (by preventing the start of the rollback requests as well as by preventing to revert the memories). To ensure this, to execute a `roll` k process, we first check that no memory that depends on k is marked. This implies that the presence of a `roll` k is not enough to guarantee that memory k will actually be reverted. Indeed, a `commit` k' where k' is a descendant of k may prevent this. This design choice reminds *open-nested transactions* [1], but the behaviour is actually different. In open-nested transactions, children transactions may commit before the father and, if the father cannot commit, then children commits' are compensated. Our case is slightly different: if the children commits then the father cannot rollback any more.

4 The calculus

We now present the **broll**$-\pi$ calculus. In Section 4.1, we introduce its syntax. In Section 4.2, we introduce its semantics. The novelty of the semantics lies in an extra rule for communication on $\{c\}$ (rule (NCom)), which does not propagate the causal dependency to the sender; and a few new rules for committing memories. The latter is not trivial as it effectively rewrites history via memory removal. Therefore, most of the new rules are intended to properly cleanup leftovers. In Section 4.3, we illustrate the new commit semantics using an example, clarifying in particular the interaction between commits and rollbacks.

4.1 Syntax

The syntax of bounded **roll**$-\pi$, given in Grammar 1.2, extends the one of **roll**$-\pi$ [9] (see Grammar 1.1). We note $[\mu; k]^{\oplus}$ for $[\mu; k]$, $[\mu; k]^{\bullet}$, or $[\mu; k]^{\dagger}$. We note α (and decorated variants) for a or $\{a\}$; and define $\mathtt{name}(a) = \mathtt{name}(\{a\}) = a$.

$\mathcal{P}, \mathcal{Q}$	$::=$	$\ldots$	*As in* **roll**$-\pi$ *except Send/Receive*
		$\mid \alpha\langle\mathcal{P}\rangle$	*Sending message*
		$\mid \alpha(X) \triangleright_\gamma (\mathcal{P})$	*Receiving message*
		$\mid$ `commit` $k \mid$ `commit` γ	*Commit primitive*
$\mathcal{M}, \mathcal{N}$	$::=$	$\ldots$	*As in* **roll**$-\pi$
		$\mid [\mu; k]^{\dagger}$	*Memory marked for removal*
		$\mid$ `init` κ	*Tokens: initial tag*
κ	$::=$	$\ldots$	*As in* **roll**$-\pi$
μ	$::=$	$\ldots$	*As in* **roll**$-\pi$
		$\mid \kappa : \alpha(X) \triangleright_\gamma (Q)$	*Non-causally consistent memory content*
α	$::=$	$\{c\} \mid c$	*(Non) reversible channels*

Grammar 1.2. Grammar of bounded **roll**$-\pi$.

The two main differences between the syntax of **broll**$-\pi$ and of **roll**$-\pi$ are as follows. First, there is an additional *commit* primitive, which prevents a

memory (identified by its key) to be rolled back. Second, communications can happen in two ways: either causally consistently as in **roll**$-\pi$ (when both ends of the communication use plain names: c) or in a non causally consistent way (when either end of the communication uses bracketed names: $\{c\}$).This will affect how memories are built.

To accommodate for those two novelties, we have to adapt other elements of the grammar:

- In addition to $\kappa_1 : c\langle\mathcal{P}\rangle \parallel \kappa_2 : c(X) \triangleright_\gamma (Q)$, the content of memory can also be $\kappa : \{c\}(X) \triangleright_\gamma (Q)$ or $\kappa : c(X) \triangleright_\gamma (Q)$ when bracketed names are used (in the second case the bracketed name was on the sender side). In that case, the memory contains only the receiver process, since the sender and the receiver are not causally linked.
- Memories can be marked with †. This mark is used to identify committed memories that will eventually be erased.
- We introduce a token $\texttt{init}\ k$, which identifies *initial* processes, i.e. processes that are not associated with a memory. This is needed to identify whether a process depends on an existing memory or not.

Initial configuration. In **roll**$-\pi$, a configuration is initial if it has no memory, tags are unique and simple, and all variables are bound. In addition, initial configurations of **broll**$-\pi$ need to have a token $\texttt{init}\ k$ for each tagged process $k : P$.

4.2 Semantics

As mentioned before, **roll**$-\pi$ rollback has been given two semantics. We focus here on the high-level semantics, since it is simpler, while leaving the extension of the low-level one (more easily implementable) for future work. Interestingly, the mechanism used to remove memories upon commit has similarities with the low-level rollback mechanism. We believe this to be a relevant choice, since removing memories is only needed for garbage collection, hence can be done asynchronously.

Structural congruence. We use the same structural congruence than in [9].

Causally consistent communication. When communicating on unbounded channels, communication happens as in **roll**$-\pi$. We use the reduction rule (Com) presented above.

Non causally consistent communication. The first novelty of **broll**$-\pi$ is *non causally consistent* communication, which decouples the trigger and the message in rollback. In particular, rollbacking the message has no impact on the corresponding trigger. Rolling back the trigger instead does not restore the

message (which would be the only possible impact on the message, since messages are asynchronous hence they have no descendants). In order to ensure causal-consistent reversibility as in the previous paragraph, both sender and receiver need to agree by using the channel as normal, e.g. c. If instead any of them wants to communicate in a non-causal consistent way, it can use the channel bracketed, e.g. $\{c\}$. The rule regulating non causally consistent communication is then:

$$(\text{NCom})\ \frac{\mathtt{name}(\alpha_1) = \mathtt{name}(\alpha_2) \qquad \alpha_1 = \{a\} \text{ or } \alpha_2 = \{a\}}{\begin{array}{c}\kappa_1 : \ \alpha_1 \langle P \rangle \ \| \ \kappa_2 : \ \alpha_2(X) \triangleright_\gamma (Q) \\ \twoheadrightarrow \kappa_1 : \ 0 \ \| \ \nu k. \left(k : \ Q\{^{P,k}/_{X,\gamma}\} \ \| \ [\kappa_2 : \ \alpha_2(X) \triangleright_\gamma (Q); k]\right)\end{array}}$$

Let us note that doing non causal consistent reversibility is a unilateral decision, as it is sufficient that one of the two involved processes is not willing to revert the communication. There are two differences between rule (Com) and rule (NCom). Indeed, in rule (NCom):

1. the created memory only contains the receiver process, and has no link with the message; and
2. the message completely disappears, but for leaving a leftover $\kappa_1 : \ 0$.

Not storing the message has a limited impact, as it may be restored by rolling back the memory that initially created the message (in case the message was present in the initial configuration, one can add a communication to create a memory creating it).

Since the memory only contains the trigger, then a rollback on the message has no impact on it. However, we need the leftover $\kappa_1 : \ 0$ to allow for rollback on the sender side. A rollback on the trigger side will leave the sender side unaffected.

Committing computations. The second novelty of **broll**$-\pi$ are *commits*. A commit targets a past interaction, by means of the tag of the corresponding memory, and forbids to undo it. In other terms, rollbacks involving the undo of such an interaction, possibly as part of rollbacks of older interactions, are disallowed. Since the committed interaction cannot be rollbacked any more, the corresponding memory and all the memories causing it become useless and can be garbage collected. The garbage collection of memories is similar to the low level semantics of rollbacks (cf. [9]), but it works from the target memory towards the past instead of towards the present.

Commit acts by marking with a dagger the committed memory:

$$(\text{Commit.S})\ \frac{}{k_1 : \ \mathtt{commit}\ k \ \| \ [\mu; k] \twoheadrightarrow k_1 : \ \mathtt{commit}\ k \ \| \ [\mu; k]^\dagger}$$

In order to understand why this commits the memory, we need to discuss rollbacks.

Definition 1 (Commit-free configuration). *A configuration M is commit-free (noted* commit-free(M)*) if it does not contain any memory of the form* $[\mu; k]^\dagger$.

Thus, the rollback rule can easily be adapted:

$$\text{(B.Roll)}\ \frac{N \blacktriangleright k \qquad \text{complete}(N \parallel [\mu; k]) \qquad \text{commit-free}(N)}{N \parallel [\mu; k]^{\bullet} \twoheadrightarrow \mu \parallel N \not\downarrow_k}$$

Notice that in this approach a rollback either fully takes place, or it is forbidden, there is no partial rollback.

Garbage collecting useless memories. We now discuss how to garbage collect memories which are no more needed since a commit forbids to rollback them.

First, we need to understand which memories need to be garbage collected. To this end, the mark on a memory is propagated to its ancestors:

$$\text{(Commit.M)}\ \frac{\mu \equiv \kappa_1 : P \parallel M \qquad \kappa_1 = \langle \_, \tilde{\_} \rangle \cdot k' \text{ or } \kappa_1 = k'}{[\mu; k]^{\dagger} \parallel [\mu'; k']^{\circ} \twoheadrightarrow [\mu; k]^{\dagger} \parallel [\mu'; k']^{\dagger}}$$

Notice that a single application of the rule propagates the mark to a single ancestor, multiple applications are needed to cover all the ancestors.

The question then arises of when to stop propagating the mark, and when one can actually start erasing memories. Indeed, we can not distinguish the first memory of a configuration unless we look at the whole system, which is not practicable. To address this issue, we introduce $\texttt{init}\ \kappa$ tokens, which mark initial processes. We assume such a token exists for each initial process in an initial configuration[8]. When we mark a memory $[\mu; k]^{\dagger}$ for erasure and such tokens exist for all processes in μ, we know this memory has no ancestor and can be removed. Upon erasure, we also remove the tokens, and replace them with a new $\texttt{init}\ k$ token, as the immediate successor of the memory is now the oldest process (or memory) in the configuration.

$$\text{(Commit.E)}\ \frac{}{\left[\prod_{\kappa_i \in \tilde{\kappa}} (\kappa_i : P_i); k\right]^{\dagger} \parallel \prod_{\kappa_i \in \tilde{\kappa}} (\texttt{init}\ \kappa_i) \twoheadrightarrow \texttt{init}\ k}$$

To obtain the required $\texttt{init}\ \kappa_i$, we may need to split a token $\texttt{init}\ k$ into complex keys, if the corresponding processes are split as well.

$$\text{(Commit.B)}\ \frac{\langle h_i, \tilde{h} \rangle \cdot k \text{ occurs in } M}{\texttt{init}\ k \parallel M \twoheadrightarrow \prod_{h_i \in \tilde{h}} (\texttt{init}\ \langle h_i, \tilde{h} \rangle \cdot k) \parallel M}$$

Finally, introducing commits and memory erasure makes it possible to have dangling $\texttt{roll}\ k$ or $\texttt{commit}\ k$, i.e. where the target tag k does not refer to any

[8] Intuitively, we want to maintain the invariant that, for all processes $k : P$, there exists either a memory $[\mu; k]^{\oplus}$ or a token $\texttt{init}\ k$ (and similarly for complex tags).

memory (if the memory tagged with k is erased or rollbacked). We introduce a rule to clean-up such dangling processes. We identify this case when a key k used for a tag is only used by $\texttt{commit}\ k$ or $\texttt{roll}\ k$ processes, or $\texttt{init}\ \kappa$ tokens, i.e. when the binder of the key only binds such processes/tokens.

$$(\textsc{Commit.C1})\ \frac{\begin{array}{c} M_i = \kappa_i : \texttt{roll}\ k \text{ or } M_i = \kappa_i : \texttt{commit}\ k \\ I = 0,\ I = \texttt{init}\ k \text{ or } I = \prod_{h_i \in \tilde{h}} (\texttt{init}\ \langle h_i, \tilde{h} \rangle \cdot k) \end{array}}{\nu k.\left(I \parallel \prod_i (M_i)\right) \twoheadrightarrow \nu k.\left(I \parallel \prod_i (\kappa_i : 0)\right)}$$

Notice that we possibly end up with $\kappa_i : 0$ processes (with distinct tags), as in **roll**$-\pi$. In **roll**$-\pi$, those processes need to be kept in order to track causal dependencies (even when a process is 0, it can be reverted and we need the tag to identify which memory lead to this 0). However, in **broll**$-\pi$, thanks to $\texttt{init}\ k$ tokens, we can sometimes identify that those processes are now *initial* (either because the initial configuration contained them, or because the memory that created those processes were committed). In that case, we can collect them. We also collect the associated $\texttt{init}\ \kappa$ to avoid leftovers.

$$(\textsc{Commit.C2})\ \frac{}{\texttt{init}\ \kappa \parallel \kappa : 0 \twoheadrightarrow 0}$$

As in **roll**$-\pi$, all the reductions mentioned above can happen inside parallel composition and restriction contexts.

4.3 Example

We illustrate **broll**$-\pi$ with an example, which combines both non causally-consistent communications and commits. The initial configuration we consider is:

$$k_P : a(X) \triangleright_\gamma (\texttt{roll}\ \gamma \parallel b\langle 0 \rangle) \parallel k_Q : b(Y) \triangleright_\delta (\texttt{commit}\ \delta)$$
$$\parallel k_R : \{a\}\langle 0 \rangle \parallel \texttt{init}\ k_P \parallel \texttt{init}\ k_Q \parallel \texttt{init}\ k_R$$

For the sake of conciseness, we do not show all intermediate steps.

Non causally-consistent communications. This configuration first reduces with (NCom):

$$\overset{(\textsc{NCom})}{\twoheadrightarrow}\ \nu k'_P.\begin{pmatrix} k'_P : (\texttt{roll}\ k'_P \parallel b\langle 0 \rangle) \parallel k_Q : b(Y) \triangleright_\delta (\texttt{commit}\ \delta) \\ \parallel [k_P : a(X) \triangleright_\gamma (\texttt{roll}\ \gamma \parallel b\langle 0 \rangle); k'_P] \\ \parallel k_R : 0 \parallel \texttt{init}\ k_P \parallel \texttt{init}\ k_Q \parallel \texttt{init}\ k_R \end{pmatrix}$$

Notice that the created memory contains only the receiver part, not the pending message; and an empty process tagged with k_R replaces the message (also in

case a rollback occurs). In our case, this leftover is initial (since the message was initial as well), and can be collected with (Commit.C2):

$$\overset{(\textsc{Commit.C2})}{\twoheadrightarrow} \nu k'_P.\begin{pmatrix} k'_P : \ (\texttt{roll}\ k'_P \parallel b\langle 0\rangle) \parallel k_Q : \ b(Y) \triangleright_\delta (\texttt{commit}\ \delta) \\ \parallel \ [k_P : \ a(X) \triangleright_\gamma (\texttt{roll}\ \gamma \parallel b\langle 0\rangle); k'_P] \\ \parallel \texttt{init}\ k_P \parallel \texttt{init}\ k_Q \end{pmatrix}$$

If a rollback occurs, the message would not be reverted:

$$\overset{(\textsc{H.Start})^\star}{\overset{(\textsc{B.Roll})}{\rightsquigarrow}} k_P : \ a(X) \triangleright_\gamma (\texttt{roll}\ \gamma \parallel b\langle 0\rangle) \parallel k_Q : \ b(Y) \triangleright_\delta (\texttt{commit}\ \delta) \\ \parallel \texttt{init}\ k_P \parallel \texttt{init}\ k_Q$$

Committing memories. Instead of rolling back, let's assume forward computation continues with (Com), where $\tilde{k'_P} = \{k'_{P1}, k'_{P2}\}$. For the sake of conciseness, we replace $\nu k'_P, k'_{P1}, k'_{P2}, k'_Q.(\ldots)$ with $\nu_{\cdot\cdot\cdot\cdot}(\ldots)$. We call M the obtained configuration:

$$\twoheadrightarrow^\star \nu_{\cdot\cdot\cdot\cdot}.\begin{pmatrix} \langle k'_{P1}, \tilde{k'_P}\rangle \cdot k'_P : \ \texttt{roll}\ k'_P \parallel k'_Q : \ \texttt{commit}\ k'_Q \\ \parallel \ \left[\langle k'_{P2}, \tilde{k'_P}\rangle \cdot k'_P : \ b\langle 0\rangle \parallel k_Q : \ b(Y) \triangleright_\delta (\texttt{commit}\ \delta); k'_Q\right] \\ \parallel \ [k_P : \ a(X) \triangleright_\gamma (\texttt{roll}\ \gamma \parallel b\langle 0\rangle); k'_P] \\ \parallel \texttt{init}\ k_P \parallel \texttt{init}\ k_Q \end{pmatrix}$$

At this stage, two reductions can take place: we can initiate a rollback of k'_P; or we can initiate a commit of k'_Q. We focus on the commit, with a short remark on the interplay between rollbacks and commits.

Starting a commit phase happens by triggering (Commit.S). This rule requires a $\texttt{commit}\ k$ process. In our case, we target the tag k'_Q. This rule marks the memory with tag k'_Q with a †.

$$M \overset{(\textsc{Commit.S})}{\twoheadrightarrow} \nu_{\cdot\cdot\cdot\cdot}.\begin{pmatrix} \langle k'_{P1}, \tilde{k'_P}\rangle \cdot k'_P : \ \texttt{roll}\ k'_P \parallel k'_Q : \ \texttt{commit}\ k'_Q \\ \parallel \ \left[\langle k'_{P2}, \tilde{k'_P}\rangle \cdot k'_P : \ b\langle 0\rangle \parallel k_Q : \ b(Y) \triangleright_\delta (\texttt{commit}\ \delta); k'_Q\right]^\dagger \\ \parallel \ [k_P : \ a(X) \triangleright_\gamma (\texttt{roll}\ \gamma \parallel b\langle 0\rangle); k'_P] \\ \parallel \texttt{init}\ k_P \parallel \texttt{init}\ k_Q \end{pmatrix}$$

Remark 1 (Concurrent commits and rollback). The $\texttt{roll}\ k'_P$ process can initiate a rollback with (H.Start), thus marking the second memory. To perform the rollback, rule (B.Roll) needs to apply. However, the memory $\left[\ldots; k'_Q\right]^\dagger$ is causally-dependent on k'_P, thus the predicate commit-free(N) does not hold, which forbids (B.Roll) to apply. On the other hand, rule (Commit.M) ignores marks on memories marked for rollback. Thus, memories marked for rollback do not block commits.

Since we now have a †-marked memory, we can trigger the rule (Commit.M), which spans the mark to predecessor memories, e.g. the memory with tag k'_P.

$$\xrightarrow[(\text{Commit.M})]{} \nu \ldots. \begin{pmatrix} \langle k'_{P1}, \tilde{k'_P} \rangle \cdot k'_P : \texttt{roll}\ k'_P \parallel k'_Q : \texttt{commit}\ k'_Q \\ \parallel \left[\langle k'_{P2}, \tilde{k'_P} \rangle \cdot k'_P : b\langle 0 \rangle \parallel k_Q : b(Y) \triangleright_\delta (\texttt{commit}\ \delta); k'_Q \right]^\dagger \\ \parallel [k_P : a(X) \triangleright_\gamma (\texttt{roll}\ \gamma \parallel b\langle 0 \rangle); k'_P]^\dagger \\ \parallel \texttt{init}\ k_P \parallel \texttt{init}\ k_Q \end{pmatrix}$$

The memory with tag k'_P has no predecessor. We see that the process it contains (identified with k_P) is an initial process (there is an $\texttt{init}\ k_P$ token in the configuration). Therefore, we can erase that memory with (Commit.E). Notice that in the resulting configuration, the process k'_P, created by the communication recorded in the memory with the same tag, is now initial, and therefore, an $\texttt{init}\ k'_P$ token is created.

$$\xrightarrow[(\text{Commit.E})]{} \nu \ldots. \begin{pmatrix} \langle k'_{P1}, \tilde{k'_P} \rangle \cdot k'_P : \texttt{roll}\ k'_P \parallel k'_Q : \texttt{commit}\ k'_Q \\ \parallel \left[\langle k'_{P2}, \tilde{k'_P} \rangle \cdot k'_P : b\langle 0 \rangle \parallel k_Q : b(Y) \triangleright_\delta (\texttt{commit}\ \delta); k'_Q \right]^\dagger \\ \parallel \texttt{init}\ k_Q \parallel \texttt{init}\ k'_P \end{pmatrix}$$

The memory with tag k'_Q is the next one to be erased. However, rule (Commit.E) requires a token $\texttt{init}\ \langle k'_{P2}, \tilde{k'_P} \rangle \cdot k'_P$, which does not exist. However, $\langle k'_{P2}, \tilde{k'_P} \rangle \cdot k'_P$ appeared while splitting k'_P, which is initial. Therefore, we can split the token $\texttt{init}\ k'_P$ to obtain the key needed using (Commit.B).

$$\xrightarrow[(\text{Commit.B})]{} \nu \ldots. \begin{pmatrix} \langle k'_{P1}, \tilde{k'_P} \rangle \cdot k'_P : \texttt{roll}\ k'_P \parallel k'_Q : \texttt{commit}\ k'_Q \\ \parallel \left[\langle k'_{P2}, \tilde{k'_P} \rangle \cdot k'_P : b\langle 0 \rangle \parallel k_Q : b(Y) \triangleright_\delta (\texttt{commit}\ \delta); k'_Q \right]^\dagger \\ \parallel \texttt{init}\ k_Q \parallel \texttt{init}\ \langle k'_{P1}, \tilde{k'_P} \rangle \cdot k'_P \parallel \texttt{init}\ \langle k'_{P2}, \tilde{k'_P} \rangle \cdot k'_P \end{pmatrix}$$

From this point, we can trigger (Commit.E) again, removing the last memory. We are left with `commit` and `roll` processes only which refer to nonexistent memories. Those processes can be collected using the cleanup rules (Commit.C1) and (Commit.C2), resulting in an empty configuration.

$$\xrightarrow[(\text{Commit.E})]{} \nu \ldots. \begin{pmatrix} \langle k'_{P1}, \tilde{k'_P} \rangle \cdot k'_P : \texttt{roll}\ k'_P \parallel k'_Q : \texttt{commit}\ k'_Q \\ \parallel \texttt{init}\ \langle k'_{P1}, \tilde{k'_P} \rangle \cdot k'_P \parallel \texttt{init}\ k'_Q \end{pmatrix} \xrightarrow[(\text{Commit.C2})^3]{(\text{Commit.C1})^3} 0$$

5 Application: a speculative consensus algorithm

We present a practical example which illustrates the use of both non causally-consistent communications and commits: a consensus algorithm with speculative

execution. Like a regular consensus protocol [2], we are given n processes (named p_1 to p_n) which aim to agree on a single value. To this end, each process p_i *proposes* a value v_i, and eventually *decides* on the selected value v. To reach this agreement, the distributed system must run a *consensus protocol* such that (i) all p_i eventually decide (termination); (ii) all p_i decide the same value v (agreement); (iii) the value v is one of the proposed v_i (non-triviality); and (iv) every process decides once (integrity). In our case, *decision* occurs on committing the result of the algorithm. Notice that multiple messages may be sent on the decision channels due to rollbacks.

To implement such protocol, we augment **broll**$-\pi$ with high-level language features, such as control flow (if-then-else) and data types (integers and arrays). For the sake of simplicity we do not detail further those features and we assume they integrate seamlessly into **broll**$-\pi$. Our implementation focuses on the consensus protocol, implemented as a (set of) processes C_i, which p_i interact with. In a first step, we describe *how* processes p_i interact with C_i to initiate the consensus protocol and recover the decided value (i.e. we present the user interface of C_i). Then, we delve into C_i, first explaining a few preliminary items before describing the algorithm itself.

Consensus process interface. The consensus protocol C_i (which p_i interacts with) is shown in Figure 1, where the behaviour of a process C_i is presented. To propose a value v_i, p_i sends it on $\texttt{propose}_i$ and C_i then takes over, eventually sending the decided value on $\texttt{decide}_i$, which is to be received by p_i. Our algorithm includes speculative execution: even before all values from peers are received (or even proposed), a value is sent on $\texttt{decide}_i$, which allows p_i to continue. If this value is not consistent with the proposed values from the peers, reversible execution allows to roll back p_i and deciding another value. We assume p_i internally uses only the reversible fragment of **broll**$-\pi$. If p_is use non causally consistent communication during speculative execution, peers would not necessarily be reverted when the speculative execution is aborted[9]. If any p_i initiates a commit while performing speculative execution, this commit also affects the consensus part, which causally precedes the speculative execution. This permanently spoils the consensus algorithm.

Preliminary items. Before explaining the core content of our algorithm, we shall introduce a few preliminary elements.

First, our algorithm contains a channel l_i, which is initially and exclusively used to set up a rollback point (γ_i) prior to any event. In the following, if an inconsistency happens, the algorithm rolls back to γ_i, thus reverting any consequences of the inconsistency.

Second, our algorithm (for participant i) relies on an array which contains proposed values that are known to participant i. This is an array of n cells,

[9] Peers that take part in the consensus would be reverted, even though the rollback would not propagate through the communication.

$$
\begin{aligned}
&s_i\,\langle \mathtt{A_{init}}\rangle \;||\; l_i\,\langle 0\rangle \;||\; \mathbf{F}(\{s_i'\}(X) \triangleright (X)) \\
&||\; l_i(X) \triangleright_{\gamma_i} \left(\prod_{1\le j\le n} \left(c_{j\to i}(X_j) \triangleright \left(\{s_i\}(A) \triangleright \left(\begin{array}{l} \mathtt{if}\ \ i=j \\ \quad \mathtt{then} \prod_{\substack{1\le k\le n \\ k\neq i}} (c_{i\to k}\,\langle X_j\rangle) \\ \\ ||\; \{s_i'\} \left\langle \left(\begin{array}{l} s_i \left\langle A\{^{X_j}/_{A[j]}\ \} \right\rangle \\ ||\ \left(\begin{array}{l} \mathtt{if}\ \ \min(A) > X_j \\ \quad \mathtt{then\ roll}\ \gamma_i \end{array} \right) \end{array} \right) \right\rangle \\ ||\; \mathtt{if}\ \ \bot \notin A\ \mathtt{then\ commit}\ \gamma_i \end{array} \right) \right) \right) \right) \\
&\qquad ||\; s_i(A) \triangleright (s_i\,\langle A\rangle \;||\; \mathtt{if}\ \ \neg \mathrm{is\_empty}(A)\ \ \mathtt{then}\ \mathtt{decide}_i\,\langle \min(A)\rangle) \Big) \\
&||\; \mathtt{propose}_i(v_i) \triangleright (c_{i\to i}\,\langle v_i\rangle)
\end{aligned}
$$

Fig. 1. The consensus protocol C_i used by process p_i.

initially empty (we note $\bot$ the absence of value, and we call $\mathtt{A_{init}}$ the initial array). At any time, there is at most one copy of this array in a pending message $\{s_i\}\,\langle \mathtt{A}\rangle$. To access the array, one receives the pending message containing the array. The channels s_i and s_i' are used internally (they are restricted to the algorithm). They allow us to keep known proposed values even during rollback. The channel s_i is used to make the array available. The message is made available using s_i in a causally-consistent way, thus leaving the reader the choice of whether it is read in a causally-consistent way or not. Channel s_i' is used to update the array. Updating the array consists in emitting a message on s_i. To escape the scope of the rollback to γ_i, modifications of the array use $\{s_i'\}$ in a non-causally consistent way. There is a replicating receive-and-execute listener process on $\{s_i'\}$.

To replicate a process, we introduce $\mathbf{F}(P)$ which creates as many copies of P as needed. This replicator is implemented as[10]:

$$
\mathbf{F}(P) = \nu c.\, \left(\begin{array}{l} c\,\langle c(X) \triangleright_\gamma (P \;||\; X \;||\; c\,\langle X\rangle \;||\; \mathtt{commit}\ \gamma)\rangle \\ ||\; c(X) \triangleright_\gamma (P \;||\; X \;||\; c\,\langle X\rangle \;||\; \mathtt{commit}\ \gamma) \end{array} \right)
$$

Third, there are channels $c_{j\to i}$ for each pairs of participants. Those channels are used to broadcast proposed values. This includes a channel $c_{i\to i}$, used to forward internally the value proposed by p_i.

Finally, the value that is decided is the lowest proposed value. This is arbitrary, but it ensures the satisfaction of the agreement and non-triviality properties.

[10] The two occurrences of $\mathtt{commit}\ \gamma$ are not needed strictly speaking, but creating copies also creates memories. We add the $\mathtt{commit}\ \gamma$ to clean up those memories.

The algorithm. We can now look at the core content of the algorithm, which is the continuation of the trigger on l_i. This continuation is composed of two main parts.

First, let us discuss the actions to take when a new proposed value is received on one of the $c_{j\to i}$ channels (even when $j = i$, which is only marginally different). Upon reception of a proposed value X_j, we read the array (from $\{s_i\}$[11]) and up to three out of the four actions below take place in parallel:

- If the value is from p_i (i.e. if it is the local proposed value), it is sent to all other participants (which eventually trigger the same events on their own channels).
- If the array that has been read is already full, then X_j was already known and all proposed values are known and inconsistencies can no longer happen. In that case, we commit the protocol.
- In any case, we update the content of the array with the value received: we create a new pending message on s_i containing the array updated with X_j in the j-th cell. Notice that this message should persist in case of rollback. To enforce that, we first escape the scope of the rollback to γ_i using $\{s'_i\}$: we send the message $s_i \left\langle A\{{}^{X_j}/_{A[j]}\ \}\right\rangle$, that contains the updated array on $\{s'_i\}$, thus any future rollback will not revert this message.
- Finally, if X_j is strictly lower than all known values so far, then speculative execution of p_i is inconsistent, and is therefore rolled back: we roll back to γ_i. However, before rolling back, we have to make sure the newly created message on s_i (containing the updated array) has been received by $\{s'_i\}$: if not, an early rollback could rollback the pending message on $\{s'_i\}$, thus erasing the message on s_i contained in it. To prevent this unfortunate scheduling, the rollback decision is sent on $\{s'_i\}$ *together* with the array message itself, therefore the rollback can trigger only after the message on $\{s'_i\}$ is received.

Notice that the two last items are mutually exclusive: if the array is full, then X_j is already in the array and therefore, it can not be strictly smaller than the minimum value of the array.

The first part ensures that the array eventually contains all proposed values, but we still need to return the decided value to p_i, by sending it on $\texttt{decide}_i$. The second parallel element of the continuation does it. At any time, the array can be read, and (if it is not empty), its minimum value is sent on $\texttt{decide}_i$, thus made available to p_i. This decision can be speculative, if the array is not full (in which case a rollback may occur in the future). In this process, the array is not modified, which allows us to read it in a causally consistent way. This is needed due to rollbacks: when a rollback occurs, the pending $s_i \langle A\rangle$ is reverted, thanks to the causally-consistent receive, the message consumed by the trigger is restored, and that message is consistent as the array is not modified.

Our reversible approach interferes with the properties of consensus algorithms mentioned above. In particular, it requires a careful definition of what it means

[11] We read the array in a non-causally-consistent way, ensuring the old version of the array is not restored, even in case of rollbacks

deciding for a process in our context. In particular, sending on $\texttt{decide}_i$ can not constitute a decide event, since multiple messages can be sent and reverted on that channel, with values inconsistent w.r.t. that of other processes. Such a definition would thus violate agreement and integrity. Instead, the committing $\texttt{commit}\ \gamma_i$ constitutes a suitable choice for the *decide* event, for each process executes commit just once (integrity). Agreement and non-triviality should be straightforward, showing that each process ends up with the same array. Termination is less trivial, due to rollbacks and administrative reductions. However, the number of rollback for a process is bounded by n, since the a new element is inserted when rolling back. Administrative reductions (e.g. reducing the $\mathbf{F}(\dots)$) however could be unbounded, and ought to be ignored or limited.

6 Discussion

The two extensions introduced in **broll**$-\pi$ blur the line between normal (forward-only) calculi and strict causal consistent reversible calculi. In order to illustrate this relaxed boundary, in this section we sketch a few applications of our calculus and highlight some relevant points.

6.1 Reversibility domains

We call *reversibility domains* a system of multiple processes where subsets of the processes of the system can rollback together in a causally-consistent way, but where such rollbacks are limited to the subset considered. Intuitively, such system could, for instance be a computation distributed across multiple datacenters. Each datacenter could have a local fault-tolerance mechanism based on reversibility, and communications across datacenters would be considered as side-effects, i.e. non-reversible actions.

The **broll**$-\pi$ calculus is general enough to easily model such behaviour: processes belonging to the same domain would communicate using causally-consistent communications, while processes belonging to different domains would communicate using non causally-consistent communications.

In fact, **broll**$-\pi$ is strictly more general than a calculus implementing only domains, since channels can be used differently at different steps, while a strict application of reversibility domains would be more static. This allows us to model, for instance, merging or splitting domains.

6.2 Interactions between reversible and recursive processes.

Following the same idea of reversibility domains, we can also sketch an interesting application of our work: a single system (i.e. a single term of the calculus) can have a part that is programmed using a reversible paradigm, and another part that uses a more conventional forward-only recursive paradigm. Said otherwise, reversibility is seen as any other programming primitive/style/paradigm, not only for fault tolerance or debugging.

For instance, consider the following term:

$$k_p : \; i\,\langle 0\rangle \;||\; i(X) \triangleright_\gamma (\{c\}\,\langle P\rangle \;||\; \texttt{roll}\ \gamma) \;||\; k_q : \; \mathbf{F}(\{c\}(Y) \triangleright (Q))$$

The first part (with key k_p) is a process that first reduces on an internal (reversible) channel i into $\{c\}\,\langle P\rangle \;||\; \texttt{roll}\ \gamma$ (with γ substituted with the key generated during the reduction), i.e. into a process that sends P on $\{c\}$ (i.e. a non causally-consistent communication) and that reverts the first communication, reaching again the initial state. On the second hand, the other part of the term (with key k_q) only uses non causally-consistent communications, as if written in **HO**π (except for commits to cleanup memories). This process uses the $\mathbf{F}(...)$ replicator introduced in section 5, which generates $\{c\}(Y) \triangleright (Q)$ at will.

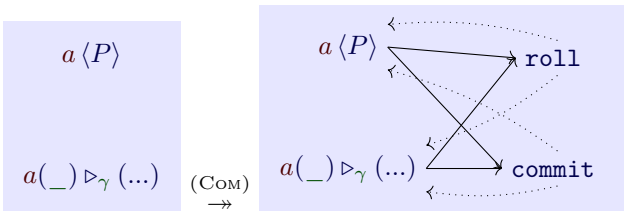
In summary, the first part is eventually able to send on $\{c\}$, while the second part is eventually able to receive on that channel, and this happens infinitely many times. The second implementation is a classical implementation of recursion in higher-order calculi, while the first one is a novel implementation enabled by the interaction between rollback and non causal-consistent communications.

6.3 On the duality of rollbacks and commits

It is to be noted that commits are, to some extent, dual of rollbacks (similarly to what happens in transactions). This is better seen by representing configurations in a more graphical way, where processes are represented as vertices of a graph, causal dependencies as (directed) edges, and tag references (e.g. in $\texttt{roll}\ k$) as backward edges. Communication rules consist in completing the graph. For instance, consider the configuration

$$k : \; a\,\langle P\rangle \;||\; l : \; a(\_) \triangleright_\gamma (\texttt{roll}\ \gamma \;||\; \texttt{commit}\ \gamma)$$

The configuration and its first reduction can be graphically represented as (where dotted arrows represent rollback/commit targets, and plain arrows causal dependencies):



If we abstract away processes (the content of the vertices), and rollback/commit edges, in order to keep the structure of the causal dependencies[12] then any configuration is represented as a directed acyclic graph. In such setting:

– communication reductions consist in adding nodes at the right of the graph;

[12] This is the graph of the $>$ relation in [9,10], which is a partial order.

– executing causally consistent rollbacks consists in removing nodes from the right, i.e. maxima of the underlying poset[13];
– executing commits consists in removing nodes from the left, i.e. minima of the underlying poset.

This seems to show an underlying duality between rollbacks and commits, which are essentially removing one end or the other of the causal dependency poset. This emphasizes that, somehow, only communications are actual computations, while rollbacks/commits are erasing one side or the other of the history.

To further emphasize this duality, instead of having two kinds of reduction (forwards and backwards), we could divide the semantics into three kinds of rules:

1. Computational reductions, i.e. reductions where actual computation occurs; i.e. rules (Com) and (NCom).
2. Backward reductions, i.e. reductions undoing some computations; i.e. all the rules related to rollbacks.
3. Forward or progress reductions, i.e. reductions that set some computations in stone; i.e. all the rules related to commits.

7 Related Work

Comparison with *irreversible actions* in RCCS [5]. When it comes to setting temporal boundaries, we think of *irreversible* actions introduced in RCCS by Danos and Krivine in [5]. Before comparing irreversible actions and commits, we shall first present reversibility in RCCS [4], which is quite different than in **roll**$-\pi/\rho\pi$ (see [12] for an interesting discussion on causal-consistent reversibility in various calculi, including the two aforementioned ones).

In RCCS, each process has a memory stack which grows when the process takes transitions. A synchronisation leads to two memories: one on each side of the communication. Rollbacks occur by popping memories from the stack, and soundness of rollbacks is ensured by the linear structure of the stack of memories. Irreversible actions are implemented by inserting special elements in the memory stack: a token $\langle\circ\rangle$. This token can not be popped, thus ensuring that actions prior to the irreversible ones can no longer be reversed.

There is a fundamental difference between irreversible actions and commits. With irreversible actions, being irreversible is a property of *actions*, while in **broll**$-\pi$, being committed is a property of *memories*. In addition, in **broll**$-\pi$, committing a memory is decided independently of the creation of the memory and, in particular, it is decided after the memory is created. However, in RCCS with irreversible actions, reversible and irreversible actions are two distinct synchronisation primitives, and choosing between the two is decided statically by the programmer.

[13] This is slightly more subtle though: all parallel processes resulting from a communication ought to be removed together, i.e. all sibling nodes should be removed in the same transition.

Implementing oracles as in $\Omega\rho\pi$ [16]. In [16], a variant of **roll**$-\pi$ called $\Omega\rho\pi$ is introduced. In $\Omega\rho\pi$, configurations of **roll**$-\pi$ are able to communicate with an *oracle*. Sending a message to the *oracle* changes its internal state; and receiving a message from the oracle reads its internal state. Contrary to other processes, the oracle state is not reverted during a rollback. This allows one to perform a computation, store the result, rollback; and then to *guess* the rolled-back result without repeating the computation (hence the name of the *oracle*).

We show that such an oracle can be implemented in **broll**$-\pi$: in fact, this paper is a generalisation of [16]. Following $\Omega\rho\pi$, we use a channel `inform` to update the state of the oracle. To read its state, we slightly diverge from $\Omega\rho\pi$: we first send a message to request a forecast (channel `req_fcast`), and then we receive the state on channel `forecast` as in $\Omega\rho\pi$. This slight modification is required for technical reasons explained below.

The state of the oracle is implemented as a pending message on a channel `state`. The oracle maintains such pending message, whose content is updated if needed. Then, the oracle consists of two (replicated) processes: one to deal with `inform`s, and the other to deal with `forecast`s. The pending message and the two replicated processes are respectively tagged with s, i and f in the oracle process below. Notice that the oracle is not intended to ever reverse. Thus, it communicates with the environment using only non causally-consistent communications, and memories are periodically committed to avoid leftovers.

$$s : \{\texttt{state}\}\,\langle S\rangle$$
$$\mid\mid\ i : \mathbf{F}(\{\texttt{inform}\}(S') \triangleright (\{\texttt{state}\}(\_) \triangleright_\gamma (\{\texttt{state}\}\,\langle S'\rangle)))$$
$$\mid\mid\ f : \mathbf{F}(\{\texttt{req\_fcast}\}(\_) \triangleright (\{\texttt{state}\}(X) \triangleright (\{\texttt{forecast}\}\,\langle X\rangle \mid\mid \{\texttt{state}\}\,\langle X\rangle)))$$

where $\mathbf{F}(P)$ is the replicator process introduced in Section 5.

Both the inform and forecast parts of the oracle are repeating using a process $\mathbf{F}(P)$, which generates as many such processes as needed. Upon receiving an oracle state update on channel `inform` the pending $\texttt{state}\,\langle S\rangle$ is consumed and a new one is produced. Upon receiving a forecast request on `req_fcast`, the pending $\texttt{state}\,\langle S\rangle$ is read, forwarded on `forecast` and a new $\texttt{state}\,\langle S\rangle$ is created. Without this reception on $\{\texttt{req\_fcast}\}$, pending messages on $\{\texttt{forecast}\}$ could spawn at will, and there would be no possibility to collect them before updating the oracle state, thus leaving pending messages on $\{\texttt{forecast}\}$ containing outdated values. Thanks to the introduction of $\{\texttt{req\_fcast}\}$, we can control the duplication of $\{\texttt{forecast}\}$, preventing this issue.

Overall, the process maintains an invariant: there is always at most one pending message on `state`, which ensures consistency w.r.t. the semantics of $\Omega\rho\pi$. Also, even when there is no such pending message (e.g. during intermediate steps), one is eventually created, thus ensuring the absence of deadlock.

Implementing compensations as in croll$-\pi$ [8]. In [8] an enhanced version of **roll**$-\pi$, named **croll**$-\pi$, featuring compensations is presented. The core idea of **croll**$-\pi$ is to enhance messages with alternatives, which serve as fallback

options. When a memory is hit by a rollback, instead of reactivating the original message, the system uses the attached alternative. This relatively simple change enables the modeling of compensations and supports communicating transactions, where interacting processes may need to jointly recover or backtrack.

While both [8] and the present paper tackles the issue of programming systems which are only partially reversible, the proposed mechanisms are quite different and may not be mutually encodable. We leave however for future work a more detailed comparison, and an integration of the two approaches if none of them would be able to encode the other.

8 Conclusion

In this paper, we introduced **broll**$-\pi$, an extension of **roll**$-\pi$ which allows (i) processes to communicate without recording the causal dependency of the receiver on the sender, and (ii) to commit past actions, making them permanent.

To implement non causally-consistent communications, we changed the content of the memory that is created when receiving a message. Instead of storing both the message and the receiver process, we store the receiver process only. Thus, upon reverting the receiver, the rollback information does not propagate to the sender side (and vice-versa).

To implement commits, **broll**$-\pi$ introduces a primitive `commit` k which removes the memory $[\mu; k]$ as well as all memories it causally depends on, effectively making it non-reversible. Concurrent commits and rollbacks are dealt with by preventing memories scheduled for erasure to be rolled back, thus preventing inconsistencies.

There are several promising directions for future work. First, we aim to develop a comprehensive behavioural theory for **broll**$-\pi$, providing deeper insights into its fundamental principles. Second, we plan to design a low-level semantics, inspired by distributed algorithms, to demonstrate how coordinated rollback/commit primitives can be implemented effectively. Furthermore, it will be crucial to prove the equivalence between that low-level semantics and the high-level one presented in this paper.

Additionally, we intend to explore whether **broll**$-\pi$ can support the encoding of long-running transactional schemas, which nowadays are used in microservices architectures.

References

1. Alejandro Buchmann. Open Nested Transaction Models. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 2597–2601. Springer, New York, NY, 2018.
2. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer, Berlin, Heidelberg, 2011.
3. Christopher D. Carothers, Kalyan S. Perumalla, and Richard Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, 1999.

4. Vincent Danos and Jean Krivine. Reversible Communicating Systems. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 292–307, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
5. Vincent Danos and Jean Krivine. Transactions in RCCS. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 – Concurrency Theory*, Lecture Notes in Computer Science, pages 398–412, Berlin, Heidelberg, 2005. Springer.
6. James Hoey, Ivan Lanese, Naoki Nishida, Irek Ulidowski, and Germán Vidal. A case study for reversible computing: Reversible debugging of concurrent programs. In Irek Ulidowski, Ivan Lanese, Ulrik Pagh Schultz, and Carla Ferreira, editors, *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, volume 12070 of *Lecture Notes in Computer Science*, pages 108–127. Springer, 2020.
7. Rolf Landauer. Irreversibility and heat generated in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.
8. Ivan Lanese, Michael Lienhardt, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Concurrent Flexible Reversibility. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 370–390, Berlin, Heidelberg, 2013. Springer.
9. Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Controlling Reversibility in Higher-Order Pi. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 – Concurrency Theory*, pages 297–311, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
10. Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversing Higher-Order Pi. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, Lecture Notes in Computer Science, pages 478–493, Berlin, Heidelberg, 2010. Springer.
11. Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversibility in the higher-order π-calculus. *Theoretical Computer Science*, 625:25–84, 2016.
12. Ivan Lanese, Claudio Antares Mezzina, and Francesco Tiezzi. Causal-Consistent Reversibility. *Bulletin of EATCS*, 3(114), October 2014. The concurrency column by Nobuko Yoshida.
13. Ivan Lanese, Iain C. C. Phillips, and Irek Ulidowski. An axiomatic theory for reversible computation. *ACM Transactions on Computational Logic*, 25(2):11:1–11:40, 2024.
14. J. McNellis, J. Mola, and K. Sykes. Time travel debugging: Root causing bugs in commercial scale software, 2017.
15. Davide Sangiorgi. Bisimulation for Higher-Order Process Calculi. *Information and Computation*, 131(2):141–178, December 1996.
16. Martin Vassor. Reversibility and Predictions. In Shigeru Yamashita and Tetsuo Yokoyama, editors, *Reversible Computation*, Lecture Notes in Computer Science, pages 163–181, Cham, 2021. Springer International Publishing.
17. 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, Leicester, UK, September 12-14, 2018, Proceedings*, volume 11106 of *Lecture Notes in Computer Science*, pages 286–303. Springer, 2018.