Causal-consistent rollback in a tuple-based language

Elena Giachino, Ivan Lanese, Claudio Antares Mezzina, Francesco Tiezzi

Causal-Consistent Rollback in a Tuple-Based Language

Elena Giachino[a], Ivan Lanese[a],
Claudio Antares Mezzina[b], Francesco Tiezzi[c]

[a] *Focus Team, University of Bologna/INRIA, Italy*
[b] *IMT, School for Advanced Studies Lucca, Italy*
[c] *School of Science and Technology, University of Camerino, Italy*

Abstract

Rollback is a fundamental technique for ensuring reliability of systems, allowing one, in case of troubles, to recover a past system state. However, the definition of rollback in a concurrent/distributed scenario is quite tricky. We propose an approach based on the notion of *causal-consistent reversibility*: any given past action can be undone, provided that all the actions caused by it are undone as well. Given that, we define a rollback as the minimal causal-consistent sequence of backward steps able to undo a given action. We define the semantics of such a rollback operator, and show that it satisfies the above specification. The approach that we present is quite general, but we instantiate it in the case of μKLAIM, a formal coordination language based on distributed tuple spaces. We remark that this is the first definition of causal-consistent rollback in a shared-memory setting. We illustrate the use of rollback in μKLAIM on a simple, but realistic, application scenario.

Keywords: reversible computation, process algebra, tuple space, KLAIM

1. Introduction

Rollback is a technique commonly used to ensure reliability of systems, e.g. for system recovery (see the survey in [1]), in operating systems [2], or in databases [3]. If an error occurs, rollback allows the system to restore a past state which was correct, and restart the computation from there. A main design decision when defining rollback is how to keep past states, and whether to keep them all or not. The issue is even more crucial in distributed systems, where it is difficult even to define what a past state is. Indeed,

different localities may have their own local states, and it is not obvious how the local states should be composed to get a global state, since due to concurrency and asynchrony it may not be clear whether two global states were simultaneous or not. Also, if local checkpoints are defined, the *domino effect* [4] problem may occur: a local rollback may require a remote rollback, which may require again a local rollback, but to a state further in the past, and so on. In practice, it may be the case that to undo a single action, the whole computation has to be undone.

We base our approach on the concept of *causal-consistent reversibility*, which is since [5] the standard notion of reversibility for concurrent systems:

> One may undo any action if no other action depending on it has been executed (and not undone).

Given that causal-consistent reversibility moves the emphasis from states to actions, the natural definition of rollback is:

> A rollback undoes a given past action γ.

Since, according to causal consistency, consequences need to be undone before their causes, undoing action γ requires to undo its consequences (if any) beforehand, thus a unique rollback may cause many actions to be undone. However, we want to minimize the number of undone actions, to avoid undoing actions not related to γ. These should not be affected by the rollback. Notice that relying on causal-consistent reversibility completely avoids the problem of understanding whether two states, or two actions, are simultaneous, replacing it with the problem of understanding whether two actions are causally related, which can be decided relying on local information. We also avoid the problem of domino effect by storing past states in an incremental way, thus being able to recover any distributed past state.

We now write a specification for our rollback, and we will prove that the rollback operator we define actually satisfies this specification. Indeed, another merit of our approach is that it is fully formal, thus enabling such a proof of correctness. Executing the rollback of action γ in a configuration M should lead to a configuration M' satisfying four requirements:

Redoability: action γ can be executed in M';

Causal consistency: M' contains no consequence of actions that have been undone;

Correctness: M can be reached (by means of forward execution of actions) from M';

Minimality: there is no M'' satisfying the three requirements above such that reaching M from M'' is shorter than reaching M from M'.

The first three properties specify that the rollback restores the system as if the action γ were never executed. Indeed, we will see that actions are uniquely identified, hence to redo action γ one has to first undo it. This implies that all its consequences – namely, all the actions depending on γ – have been undone and can be re-executed. The last requirement guarantees a further property: we undo the minimal portion of the execution needed to reach a redoable, correct and causal-consistent configuration of the system, but nothing more.

Our approach is quite general, and we discuss in Section 6 how it can be applied to different calculi, but we present it instantiated on μKLAIM [6]. This is a formal coordination language whose communication model is based on distributed shared repositories, called *tuple spaces*. Interaction in μKLAIM is supported by five primitives. Primitives **out** and **in** respectively insert tuples into and remove them from (possibly remote) tuple spaces. Primitives **eval**, to execute a process on a possibly remote location, and **newloc**, to create a new location, support distribution. Finally, μKLAIM features the primitive **read**, which reads a tuple without consuming it.

Main reasons for choosing μKLAIM are:

- it is able to describe distributed states in a natural way;
- it has a simple formal semantics based on reductions;
- besides including primitives for generating and consuming data, it also includes the primitive **read** for accessing data without consuming it.

This last point distinguishes μKLAIM from other languages where causal-consistent reversibility has been studied, such as CCS or the π-calculus, which are all based on message passing. This difference is relevant from the point of view of causality, which is a key element for causal-consistent reversibility. More precisely, μKLAIM **read** primitive allows concurrent processes to access a shared tuple while staying independent, thus undoing the actions of one of them has no impact on the others. On the contrary, languages based on message passing feature only primitives to generate and consume messages.

Indeed, μKLAIM **read** primitive is typical of shared-memory interaction. The choice of instantiating our approach on μKLAIM has therefore a twofold benefit. On the one hand, it allows us to investigate causal-consistent reversibility in a shared-memory setting: to the best of our knowledge, ours is the first study of this kind. On the other hand, the distributed nature of μKLAIM, as well as the form of interaction it supports, allows us to emphasize the effectiveness of the rollback technique we propose.

Our approach is as follows. We extend μKLAIM (described in Section 2) with a rollback operator and the machinery needed for reversibility. We present first a step-by-step backward semantics where actions are undone one-by-one (Section 3). Then we build on top of it the definition of the roll backward semantics, corresponding to the execution of the rollback operator, and show that it satisfies the specification above (Section 4). We put our reversible language, called RμKLAIM, at work on a practical example about speculative execution in a producer-consumers scenario (Section 5). Finally, we discuss how our results can be applied in different settings (Section 6) and we give an overview of related and future work (Section 7). Some proofs are deferred to the Appendix to enhance readability.

This paper is a revised and extended version of [7]. While the general aim remains the same, both the technical development and the presentation have been completely revised. Among the novelties, we highlight the fact that we now derive the semantics of rollback from an abstract specification, and we prove that the semantics actually satisfies it. Also, the rule for rollback, while equivalent to the one in [7], is formalized in a new and more modular way. We further clean up the formalization by introducing here the concept of resources produced, read, and consumed by a transition, which encompasses different notions which were unrelated in [7].

2. μKlaim syntax and semantics

KLAIM [8] is a formal coordination language designed to provide programmers with primitives for handling physical distribution, scoping and mobility of processes. KLAIM is based on the Linda [9] generative communication paradigm. Communication in KLAIM is achieved by sharing distributed tuple spaces, where processes insert, read and withdraw tuples of values. The data retrieving mechanism is based on associative pattern-matching to find the required data in the tuple spaces. In this paper, to simplify the presentation, we consider a core language of KLAIM, called μKLAIM. We refer to

(Nets)	N	::=	$\mathbf{0} \mid s :: C \mid N_1 \parallel N_2 \mid (\nu s)N$
(Components)	C	::=	$\langle et \rangle \mid P \mid C_1 \mid C_2$
(Processes)	P	::=	$\mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid A$
(Actions)	a	::=	$\mathbf{out}(t)@\ell \mid \mathbf{eval}(P)@\ell \mid \mathbf{in}(T)@\ell$
			$\mid \mathbf{read}(T)@\ell \mid \mathbf{newloc}(s)$
(Tuples)	t	::=	$e \mid \ell \mid t_1, t_2$
(Evaluated tuples)	et	::=	$v \mid s \mid et_1, et_2$
(Templates)	T	::=	$e \mid \ell \mid !\,x \mid !\,u \mid T_1, T_2$

Table 1: μKLAIM syntax

[10] for a detailed account of KLAIM and μKLAIM.

Syntax. The syntax of μKLAIM is in Table 1. We assume five disjoint sets: the set of *sites* (or localities), ranged over by s, s', ..., of *locality variables*, ranged over by u, u', ..., of *values*, ranged over by v, v', ..., of *value variables*, ranged over by x, x', ..., and of *process identifiers*, ranged over by A, B,.... *Locality references*, i.e., locality variables and localities, are ranged over by ℓ, ℓ',.... Localities are the addresses (i.e., network references) of nodes and are the syntactic ingredient used to model administrative domains. In μKLAIM, data are represented as *tuples* t, i.e., sequences of actual fields. Tuple fields may contain expressions or locality references. The syntax of *expressions*, ranged over by e, is deliberately not specified; we just assume that expressions contain values and value variables. Before being stored in tuple spaces, tuples need to be evaluated, that is their variables should be replaced by values and their expressions computed. We denote *evaluated tuples* by et.

Nets N are finite plain collections of nodes. A *node* is a pair $s :: C$, where site s is the address of the node and C is the hosted component. In the net $(\nu s)N$, the scope of the name s is restricted to N. The term $N_1 \parallel N_2$ denotes the parallel composition of nets N_1 and N_2, while $\mathbf{0}$ is the empty net. *Components* C can be either evaluated tuples $\langle et \rangle$ or processes P, while $C_1 \mid C_2$ is the parallel composition of components.

Processes P, the μKLAIM active computational units, are built up from the process **nil** – which does not perform any action – and process identifiers A, using action prefixing as $a.P$ and parallel composition as $P_1 \mid P_2$. We

assume each process identifier A has a single definition $A \triangleq P$, available at any locality of the net. We may drop trailing **nil**s. Processes may be executed concurrently either at the same locality or at different localities and can perform five different basic actions.

Actions **out**, **in** and **read** manage data repositories by adding/withdrawing/accessing data. More precisely, action $\mathbf{out}(t)@\ell$ adds the evaluated tuple resulting from the evaluation of t to the node at ℓ. Actions **in** and **read** are blocking and exploit templates as patterns to select evaluated tuples from shared repositories. *Templates* T are sequences of actual and formal fields, where the latter are written $!\,x$ and $!\,u$, and are used to bind value variables to values, and locality variables to localities, respectively. Action $\mathbf{in}(T)@\ell$ retrieves an evaluated tuple matching (the evaluation of) a template T from node ℓ, while action $\mathbf{read}(T)@\ell$ reads such an evaluated tuple without removing it from the node. In case many evaluated tuples match the template, one of them is chosen nondeterministically. Action $\mathbf{eval}(P)@\ell$ activates a new thread of execution in a node ℓ executing process P. Action $\mathbf{newloc}(s)$ permits to create a new net node with site s. All actions, except for **newloc**, indicate explicitly the locality reference ℓ where they will act.

Localities and variables can be *bound* inside processes and nets: $\mathbf{newloc}(s).P$ and $(\nu s)N$ bind locality s in, respectively, process P and net N. Prefixes $\mathbf{in}(T)@\ell.P$ and $\mathbf{read}(T)@\ell.P$ bind in P all the (locality and value) variables occurring in T as formal fields, i.e., prefixed by !. A locality/variable that is not bound is called *free*. The set $\mathrm{fn}(\cdot)$ of free names of a term (i.e., net, component, process, ...) is defined accordingly. As usual, we say that two terms are α-equivalent, written $=_\alpha$, if one can be obtained from the other by consistently renaming bound localities/variables. In the sequel, we assume Barendregt convention, i.e., we work only with terms whose bound variables and bound localities are all distinct and different from the free ones.

Operational semantics. The operational semantics of μKLAIM is given in terms of a structural congruence relation and a reduction relation expressing the evolution of a net. The structural congruence $\equiv$ identifies syntactically different representations of the same term. It is defined as the least congruence closed under the equational laws in Table 2. Most of the laws are standard, while laws *(Abs)* and *(Clone)* are peculiar to this setting. *(Monoid)* laws specify that the $\cdot \parallel \cdot$ composition is associative, commutative and that $\mathbf{0}$ is its neutral element. Law *(Alpha)* enables α-conversion. Law *(RCom)* allows one to permute restrictions, while law *(Ext)* extends their scope. Law

$$\textit{(Monoid)}\quad (N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3) \qquad N_1 \parallel N_2 \equiv N_2 \parallel N_1 \qquad N \parallel \mathbf{0} \equiv N$$

$$\textit{(Alpha)}\quad N \equiv N' \text{ if } N =_\alpha N'$$

$$\textit{(RCom)}\quad (\nu s_1)(\nu s_2)N \equiv (\nu s_2)(\nu s_1)N$$

$$\textit{(Ext)}\quad N_1 \parallel (\nu s)N_2 \equiv (\nu s)(N_1 \parallel N_2) \text{ if } s \notin \mathrm{fn}(N_1)$$

$$\textit{(PDef)}\quad s :: A \equiv s :: P \text{ if } A \triangleq P$$

$$\textit{(Abs)}\quad s :: (C \mid \mathbf{nil}) \equiv s :: C$$

$$\textit{(Clone)}\quad s :: C_1 | C_2 \equiv s :: C_1 \parallel s :: C_2$$

Table 2: μKLAIM structural congruence

(PDef) replaces a process identifier by its definition. Law *(Abs)* states that **nil** is the identity for $\cdot \mid \cdot$. Law *(Clone)* turns a parallel between co-located components into a parallel between nodes (thus, it is also used, together with *(Monoid)* laws, to achieve commutativity and associativity of $\cdot \mid \cdot$). Thanks to law *(Clone)*, a single node can also be represented as the composition of many nodes with the same locality.

To define the reduction relation, we need an auxiliary *pattern-matching* function $match(\cdot, \cdot)$, to verify the compliance of an evaluated tuple w.r.t. a template and to associate localities/values to variables bound in the template. The pattern-matching function is defined by the rules in Table 3 and undefined if none of the rules apply. Intuitively, an evaluated tuple matches a template if they have the same number of fields, and the corresponding fields do match: two values/localities match only if they are identical, value/locality variables which are bound match any value/locality, and the matching for free variables always fails. The function $match(T, et)$ returns a substitution for the bound variables in T when the template T and the evaluated tuple et do match, and is undefined otherwise. A *substitution* σ is a function with finite domain from variables to localities/values, and it is written as a collection of pairs of the form v/x or s/u. We use $\circ$ to denote substitution composition and ϵ to denote the empty substitution.

We use function $[\![\cdot]\!]$ for evaluating tuples and templates. Such evaluation consists in computing the value of closed expressions (i.e., expressions without variables) occurring in a tuple/template. The function is not explicitly

$$match(v, v) = \epsilon \qquad match(!x, v) = [v/x]$$

$$match(s, s) = \epsilon \qquad match(!u, s) = [s/u]$$

$$\frac{match(T_1, et_1) = \sigma_1 \qquad match(T_2, et_2) = \sigma_2}{match((T_1, T_2), (et_1, et_2)) = \sigma_1 \circ \sigma_2}$$

Table 3: μKLAIM matching rules

$$\frac{[\![t]\!] = et}{s :: \mathbf{out}(t)@s'.P \;\parallel\; s' :: \mathbf{nil} \;\rightarrow\; s :: P \parallel s' :: \langle et \rangle} \; \textit{(Out)}$$

$$\frac{match([\![T]\!], et) = \sigma}{s :: \mathbf{in}(T)@s'.P \;\parallel\; s' :: \langle et \rangle \;\rightarrow\; s :: P\sigma \parallel s' :: \mathbf{nil}} \; \textit{(In)}$$

$$\frac{match([\![T]\!], et) = \sigma}{s :: \mathbf{read}(T)@s'.P \;\parallel\; s' :: \langle et \rangle \;\rightarrow\; s :: P\sigma \parallel s' :: \langle et \rangle} \; \textit{(Read)}$$

$$s :: \mathbf{eval}(Q)@s'.P \;\parallel\; s' :: \mathbf{nil} \;\rightarrow\; s :: P \parallel s' :: Q \quad \textit{(Eval)}$$

$$s :: \mathbf{newloc}(s').P \;\rightarrow\; (\nu s')(s :: P \parallel s' :: \mathbf{nil}) \; \textit{(New)}$$

Table 4: μKLAIM operational semantics

defined since the exact syntax of expressions is deliberately not specified.

To define the semantics of μKLAIM and of its reversible extensions, we rely on the notion of evaluation-closed relation.

Definition 1 (Evaluation-closed relation)**.** *A relation $\mathcal{R}$ is* evaluation closed *if it is closed under parallel composition and restriction contexts, i.e., $N_1 \; \mathcal{R} \; N_1'$ implies $(N_1 \parallel N_2) \; \mathcal{R} \; (N_1' \parallel N_2)$ and $(\nu s)N_1 \; \mathcal{R} \; (\nu s)N_1'$, and under structural congruence, i.e., $N \equiv M \; \mathcal{R} \; M' \equiv N'$ implies $N \; \mathcal{R} \; N'$.*

Definition 2 (μKLAIM semantics)**.** *The μKLAIM reduction relation $\rightarrow$ is the smallest evaluation-closed relation satisfying the rules in Table 4.*

All rules for (possibly remote) actions **out**, **in**, **read** and **eval** require the existence of the target node s'. Requiring that s' contains only **nil** as in rules *(Out)* and *(Eval)*, or a single evaluated tuple as in rules *(In)* and *(Read)* is not restrictive, thanks to structural rule *(Clone)*. In rule *(Out)*, moreover, an **out** action can proceed only if the tuple in its argument is evaluable (otherwise, it is stuck). As a result of the execution, the evaluated tuple is released in the target node s'. Rules *(In)* and *(Read)* require the target node to contain an evaluated tuple matching their template argument T. Similarly to **out** actions, such template must be evaluable. The content of the matched evaluated tuple is then used to replace the occurrences of the variables bound by T in the continuation P of the process performing the action. Action **in** consumes the matched evaluated tuple, while action **read** does not. Rule *(Eval)* launches a new thread executing process Q on a target node s'. Rule *(New)* creates a new empty private node s'.

3. Step-by-step reversibility

In this section we introduce RμKLAIM, an extension of μKLAIM featuring an explicit rollback operator. In particular, we define its syntax, its *forward* semantics, formalizing the execution of actions, and its *step-by-step backward* semantics, formalizing the undoing of single actions in a causal-consistent way. The step-by-step backward semantics does not yet define the behavior of the rollback operator, but it is instrumental for its definition, detailed in Section 4. Also, it highlights which information is needed to enable backward steps, thus clarifying the definition of the forward semantics, which indeed adds to μKLAIM semantics the management of the information needed to enable reversibility. When we speak about the step-by-step semantics of RμKLAIM we refer to the coupling of its forward semantics and its step-by-step backward semantics. While the general approach follows the one for HOπ described in [11], the technical development changes considerably due to the differences between HOπ and μKLAIM.

We first present the syntax and step-by-step operational semantics of RμKLAIM, then we show that this semantics satisfies the typical properties expected for a causal-consistent reversible formalism with uncontrolled reversibility, such as the ones described in [5, 12, 13, 14, 11, 15]. Here, uncontrolled means that the semantics specifies how to go forward and how to go backward, but not how to choose whether to go forward or to go backward.

(Nets)	N	::=	$\mathbf{0} \mid s :: C \mid s :: \mathbf{empty} \mid N_1 \parallel N_2 \mid (\nu\, z)N$
(Components)	C	::=	$k : \langle et \rangle \mid k : P \mid C_1 \mid C_2 \mid \mu \mid k_1 \prec (k_2, k_3)$
(Processes)	P	::=	$\mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid A \mid \mathbf{roll}(\iota)$
(Actions)	a	::=	$\mathbf{out}_\gamma(t)@\ell \mid \mathbf{eval}_\gamma(P)@\ell \mid \mathbf{in}_\gamma(T)@\ell$
			$\mid \mathbf{read}_\gamma(T)@\ell \mid \mathbf{newloc}_\gamma(s)$
(Memories)	μ	::=	$[k : \mathbf{out}_\gamma(t)@s.P; k''; k']$
			$\mid [k : \mathbf{eval}_\gamma(Q)@s.P; k''; k']$
			$\mid [k : \mathbf{in}_\gamma(T)@s.P; h : \langle et \rangle; k']$
			$\mid [k : \mathbf{read}_\gamma(T)@s.P; h; k']$
			$\mid [k : \mathbf{newloc}_\gamma(s).P; k']$
(Tuples)	t	::=	$e \mid \ell \mid t_1, t_2$
(Evaluated tuples)	et	::=	$v \mid s \mid et_1, et_2$
(Templates)	T	::=	$e \mid \ell \mid !\,x \mid !\,u \mid T_1, T_2$

Table 5: RμKLAIM syntax

3.1. Syntax

The syntax of RμKLAIM (Table 5) extends the one of μKLAIM in two main directions: by introducing the **roll** operator, and by introducing the technical machinery needed to enable reversibility. The additions are highlighted by a grey background.

The first extension includes both the labeling of each action with a *reference* γ, thus making it possible to target them inside a rollback, and the addition of the new **roll** process, which requires to undo a target action. Actions are binders for references: if a_γ denotes an action with reference γ, then $a_\gamma.P$ binds γ in P. We say a net is closed if it contains no free references. For readability's sake, we omit references γ when they are not relevant. The syntax of tuples, evaluated tuples and templates is identical to that of μKLAIM.

The technical machinery to enable reversibility includes history information on past actions, in order to be able to undo them. To match the notion

of causal-consistent reversibility we also need to keep causality information. The main ingredients we introduce are *keys* to uniquely identify processes and evaluated tuples, *memories* to store the information that would be destroyed by a single (forward) step, and *connectors* tracing causal dependences. At runtime, processes and evaluated tuples need to be uniquely identified, in order to distinguish processes/evaluated tuples with the same form but different histories, thus allowing for different backward actions. Keys are ranged over by $k, k', \ldots, h, \ldots$. We use z to range over both keys and localities. As a matter of notation, we use $\tilde{z}$ to denote a sequence $z_1, \ldots, z_n$ of keys and localities, and we write $(\nu z_1, \ldots, z_n)N$ in place of $(\nu z_1) \ldots (\nu z_n)N$.

Nets are extended with the restriction on keys and with empty localities. Note that we distinguish empty localities, denoted by $s :: \mathbf{empty}$, containing no information and which can be garbage collected, from localities $s :: (k : \mathbf{nil})$ containing a **nil** process with its own key k, which may interact with a memory to perform a backward action and, thus, cannot be garbage collected. Processes and evaluated tuples are labeled by keys, uniquely identifying them. Components include, besides labeled processes and labeled evaluated tuples, also memories μ and connectors $k_1 \prec (k_2, k_3)$. At runtime, references γ are replaced by keys k. We let ι range over both keys and references γ. Note that the argument of **roll** is ι: when writing a process all the **roll**s are of the form $\mathbf{roll}(\gamma)$, but, when the action labeled by γ is executed, the γ is replaced by a key k, thus $\mathbf{roll}(\gamma)$ becomes $\mathbf{roll}(k)$.

We denote memories as $[k : a_\gamma.P; \xi]$, where a is one of the μKLAIM actions and ξ is the additional information. A memory keeps track of a past action, thus we have one kind of memory for each action. All of them store the prefix giving rise to the action and the fresh key k' generated for the continuation. They also store the original continuation P of the action. For uniformity we store the continuation in all the memories, but actually this is needed only for memories for **in** and **read**, since in this case it cannot be recovered from the running one, which is obtained by applying a substitution - i.e., a non-reversible transformation[1]. Also, the **out** memory stores the key k'' of the evaluated tuple created, while the **eval**'s one the key k'' of the spawned process, and the **in**'s one the evaluated tuple $h : \langle et \rangle$ consumed. The memory for **read** only needs the key h of the evaluated tuple read, since the evaluated

[1]One may look for more compact ways to store history information. We considered this issue for reversible μOz in [14], but this is out of the scope of the present paper.

(Monoid)	$(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3) \qquad N_1 \parallel N_2 \equiv N_2 \parallel N_1 \qquad N \parallel \mathbf{0} \equiv N$
(Alpha)	$N \equiv N' \quad \text{if } N =_\alpha N'$
(RCom)	$(\nu\, z_1)\,(\nu\, z_2)\, N \equiv (\nu\, z_2)\,(\nu\, z_1)\, N$
(Ext)	$N_1 \parallel (\nu\, z)\, N_2 \equiv (\nu\, z)\,(N_1 \parallel N_2) \quad \text{if } z \notin \mathrm{fn}(N_1)$
(PDef)	$s :: k : A \equiv s :: k : P \quad \text{if } A \triangleq P$
(Abs)	$s :: C \parallel s :: \mathbf{empty} \equiv s :: C$
(Clone)	$s :: C_1 \vert C_2 \equiv s :: C_1 \parallel s :: C_2$
(Garb)	$(\nu k)\, \mathbf{0} \equiv \mathbf{0}$
(Split)	$s :: k : P \mid Q \equiv (\nu k_1, k_2)\, s :: k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$

Table 6: rμKlaim structural congruence

tuple itself is still available in the term and uniquely identified by key h.

A connector $k_1 \prec (k_2, k_3)$ recalls that the process tagged by k_1 was a parallel composition of two sub-processes, and these sub-processes are respectively tagged by k_2 and k_3.

Uniqueness of keys is enforced by using restriction, and by only considering *consistent* nets. Notice that restriction is the only binder for keys. Free and bound keys and α-conversion are defined as usual, and from now on $\mathrm{fn}(N)$ also includes free keys.

Definition 3 (Initial and consistent nets)**.** *A* rμKlaim *net is* initial *if it contains no memories and no connectors, all its keys are distinct, and in all* **roll**(ι) *occurring in it* ι *is a reference. A* rμKlaim *net is* consistent *if it can be obtained by forward and/or backward reductions (cfr. Definition 4) starting from an initial net.*

3.2. Operational semantics

Structural congruence for rμKlaim is reported in Table 6. Rules *(Monoid)*, *(Alpha)* and *(Clone)* are as in μKlaim. Rules *(RCom)* and *(Ext)* are updated with respect to μKlaim to consider also restrictions on keys. Rule *(PDef)* now applies to process identifiers prefixed by keys. Rule *(Abs)* now does not delete **nil** terms, but only **empty** localities (in μKlaim a locality

Forward rules:

$$\frac{[\![t]\!] = et}{s :: k : \mathbf{out}_{\gamma}(t)@s'.P \;\|\; s' :: \mathbf{empty} \;\mapsto\; (\nu k', k'')\,(\, s :: k' : P\,[k/\gamma] \mid [k : \mathbf{out}_{\gamma}(t)@s'.P; k''; k'] \;\|\; s' :: k'' : \langle et \rangle\,)} \quad \textit{(Out)}$$

$$\frac{match([\![T]\!], et) = \sigma}{s :: k : \mathbf{in}_{\gamma}(T)@s'.P \;\|\; s' :: h : \langle et \rangle \;\mapsto\; (\nu k')\,(s :: k' : P\,[k/\gamma]\,\sigma \mid [k : \mathbf{in}_{\gamma}(T)@s'.P; h : \langle et \rangle; k']\,) \;\|\; s' :: \mathbf{empty}} \quad \textit{(In)}$$

$$\frac{match([\![T]\!], et) = \sigma}{s :: k : \mathbf{read}_{\gamma}(T)@s'.P \;\|\; s' :: h : \langle et \rangle \;\mapsto\; (\nu k')\,(s :: k' : P\,[k/\gamma]\,\sigma \mid [k : \mathbf{read}_{\gamma}(T)@s'.P; h; k']\,) \;\|\; s' :: h : \langle et \rangle} \quad \textit{(Read)}$$

$$s :: k : \mathbf{eval}_{\gamma}(Q)@s'.P \;\|\; s' :: \mathbf{empty} \;\mapsto\; (\nu k', k'')\,(\, s :: k' : P\,[k/\gamma] \mid [k : \mathbf{eval}_{\gamma}(Q)@s'.P; k''; k'] \;\|\; s' :: k'' : Q\,) \quad \textit{(Eval)}$$

$$s :: k : \mathbf{newloc}_{\gamma}(s').P \;\mapsto\; (\nu s')\,(\,(\nu k')\,(s :: k' : P\,[k/\gamma] \mid [k : \mathbf{newloc}_{\gamma}(s').P; k']\,) \;\|\; s' :: \mathbf{empty}\,) \quad \textit{(New)}$$

Table 7: RμKLAIM forward semantics

containing **nil** can be deleted by using rules *(Abs)* and *(Clone)*). Rule *(Garb)* garbage-collects unused keys. Rule *(Split)* splits parallel processes using a connector and generating fresh keys to preserve keys uniqueness.

Definition 4 (RμKLAIM step-by-step semantics)**.** *The step-by-step operational semantics of* RμKLAIM *consists of a* forward reduction relation $\mapsto$ *and a* step-by-step backward reduction relation $\rightsquigarrow$. *They are the smallest evaluation-closed relations (now closure under restriction contexts considers also restriction on keys) satisfying respectively the rules in Tables 7 and 8.*

Forward rules correspond to μKLAIM ones, adding the management of

Backward rules:

$$(\nu k'')\,(\, s :: k' : P' \mid [k : \mathbf{out}_\gamma(t)@s'.P; k''; k'] \;\parallel\; s' :: k'' : \langle et \rangle\,) \rightsquigarrow\; s :: k : \mathbf{out}_\gamma(t)@s'.P \;\parallel\; s' :: \mathbf{empty} \qquad \textit{(OutRev)}$$

$$s :: k' : P' \mid [k : \mathbf{in}_\gamma(T)@s'.P; h : \langle et \rangle; k'] \;\parallel\; s' :: \mathbf{empty} \rightsquigarrow\; s :: k : \mathbf{in}_\gamma(T)@s'.P \;\parallel\; s' :: h : \langle et \rangle \qquad \textit{(InRev)}$$

$$s :: k' : P' \mid [k : \mathbf{read}_\gamma(T)@s'.P; h; k'] \;\parallel\; s' :: h : \langle et \rangle \rightsquigarrow s :: k : \mathbf{read}_\gamma(T)@s'.P \;\parallel\; s' :: h : \langle et \rangle \qquad \textit{(ReadRev)}$$

$$s :: k' : P' \mid [k : \mathbf{eval}_\gamma(Q)@s'.P; k''; k'] \;\parallel\; s' :: k'' : Q \rightsquigarrow\; s :: k : \mathbf{eval}_\gamma(Q)@s'.P \;\parallel\; s' :: \mathbf{empty} \qquad \textit{(EvalRev)}$$

$$(\nu s')\,(\, s :: k' : P' \mid [k : \mathbf{newloc}_\gamma(s').P; k'] \;\parallel\; s' :: \mathbf{empty}\,) \rightsquigarrow\; s :: k : \mathbf{newloc}_\gamma(s').P \qquad \textit{(NewRev)}$$

Table 8: RμKLAIM step-by-step backward semantics

keys and memories. We have one backward rule for each forward rule, undoing the forward action.

In general, a forward RμKLAIM rule has three effects besides the one of the corresponding μKLAIM rule:

1. creating new fresh keys for the continuation of processes;
2. replacing the reference γ with the key of the process which performed the action;
3. creating, in the locality of the process which performed the action, a memory to remember the state prior the execution of the action.

Consider rule *(Out)*. Existence of the target node s' is guaranteed by requiring a parallel term $s' :: \mathbf{empty}$. If locality s' is not empty, such term can be generated by structural rule *(Abs)*. Two fresh keys k' and k'' are created to tag the continuation P and the new evaluated tuple $\langle et \rangle$, respectively. The key k, labeling the component $k : a_\gamma.P$ that executed the action, replaces γ inside process P. In this way, any free occurrence of $\mathbf{roll}(\gamma)$ in P becomes $\mathbf{roll}(k)$, thus requiring to undo action a_γ. Also, a memory is created (in the locality where the **out** prefix was) storing all the relevant information.

The corresponding backward rule, *(OutRev)*, may trigger if a memory for **out** with continuation key k' and with created tuple key k'' finds a process with key k' in its own locality and an evaluated tuple with key k'' in the target locality s'. Requiring that s' contains only the evaluated tuple tagged by k'' is not restrictive, thanks to structural rule *(Clone)*. Note also that all the actions performed by the continuation process $k' : P$ have to be undone beforehand, otherwise no process with key k' would be available at top level (i.e., outside memories). Moreover, the evaluated tuple generated by the **out**, which will be removed by the backward reduction, must bear key k'' as when it was generated. Note the restriction on key k'': this is needed to ensure that all the occurrences of k'' are inside the term, i.e., k'' occurs only in the **out** memory and in the evaluated tuple. This ensures that **read** actions that have accessed the evaluated tuple, whose resulting memory would contain k'', have been undone (the restriction would not be required without **read** actions, since in the case of **in** actions the evaluated tuple is consumed and is no more available outside memories). The problem of **read** dependences is peculiar to the μKLAIM setting, and it does not emerge in the other works in the reversibility literature. Requiring the existence of the restriction on k'' is a compact way of dealing with it. On the other hand, restricting key k' in rule *(OutRev)* would be redundant since in a consistent net it can occur only twice, and both the occurrences are consumed by the rule. Thus, the restriction can be garbage collected by using structural congruence. The execution of the backward rule *(OutRev)* undoes the effect of the forward rule *(Out)*, as proved by the Loop Lemma below. The structure of the other rules is similar. In rule *(Eval)*, k'' labels the spawned process Q. No restriction on k'' is required in rule *(EvalRev)*, since k'' cannot occur elsewhere in the term. In rule *(In)* the evaluated tuple consumed is stored in the memory, while in rule *(Read)* only its key h is needed, since after the **read** is performed the evaluated tuple is still in the term, and its key is unchanged. Rule *(New)* creates a new **empty** locality. In rule *(NewRev)* we again use restriction (now on the name s' of the locality) to ensure that no other locality with the same name exists. This could be possible since localities may be split using structural congruence rules *(Abs)* or *(Clone)*. Finally, it is worth noticing that no (forward or backward) rule is provided for the rollback operator, which means that a **roll** process has the same semantics of **nil**, i.e., it does nothing. This reflects the fact that the step-by-step semantics does not provide any behavior for **roll**. Such a behavior is indeed defined by the roll backward semantics given in Section 4.

Example 1. *We show an example to clarify the difference between the behavior of a* RμKLAIM **read** *action and its possible implementations in the other causal-consistent reversible languages in the literature. In fact, the other languages we are aware of [12, 5, 11, 13, 14] feature message-passing communication, thus the only way of accessing a resource (i.e., a message) is consuming it with an input and possibly restoring it with an output. This corresponds to the behavior we obtain in* RμKLAIM *by using an* **in** *followed by an* **out**. *To avoid introducing other syntaxes and semantics, we present the different behaviors inside* RμKLAIM. *The difference is striking in a reversible setting, while it is less compelling when only forward actions are considered.*

Consider a RμKLAIM *net N with three nodes, s_1 hosting an evaluated tuple $\langle foo \rangle$ containing a single value, and s_2 and s_3 hosting processes willing to access such evaluated tuple (we recall that non-relevant references γ are omitted):*

$$\begin{array}{rcl} N' & = & s_1 :: k_1 : \langle foo \rangle \parallel s_2 :: k_2 : \mathbf{in}(foo)@s_1.\mathbf{out}(foo)@s_1.P \\ & & \qquad\qquad \parallel s_3 :: k_3 : \mathbf{in}(foo)@s_1.\mathbf{out}(foo)@s_1.P' \end{array}$$

By executing first the sequence of **in** *and* **out** *in s_2, and then the corresponding sequence in s_3 (the order is relevant), the net evolves to:*

$$\begin{array}{l} (\nu k_2', k_2'', k_2''', k_3', k_3'', k_3''')(\, s_1 :: k_3''' : \langle foo \rangle \\ \qquad \parallel s_2 :: k_2'' : P \mid [k_2 : \mathbf{in}(foo)@s_1.\mathbf{out}(foo)@s_1.P; k_1 : \langle foo \rangle; k_2'] \\ \qquad\qquad\qquad\quad \mid [k_2' : \mathbf{out}(foo)@s_1.P; k_2'''; k_2''] \\ \qquad \parallel s_3 :: k_3'' : P' \mid [k_3 : \mathbf{in}(foo)@s_1.\mathbf{out}(foo)@s_1.P'; k_2''' : \langle foo \rangle; k_3'] \\ \qquad\qquad\qquad\quad \mid [k_3' : \mathbf{out}(foo)@s_1.P'; k_3'''; k_3''] \,) \end{array}$$

Now, the process in s_2 cannot immediately perform a backward step, since it needs the evaluated tuple $k_2''' : \langle foo \rangle$ in s_1, while only $k_3''' : \langle foo \rangle$ is available. The former evaluated tuple has been consumed by the **in** *action at s_3 (see the corresponding memory stored in s_3) and then replaced by the latter by the* **out** *action at s_3. This means that to perform the backward step of the process in s_2 one needs first to perform a backward computation of the process in s_3. Of course, this is not desired when the processes are accessing a shared resource in read-only modality. This is nevertheless the behavior obtained if the resource is, e.g., a message in $\rho\pi$ [11] or an output process in [5, 12, 13].*

The problem can be solved in RμKLAIM *using the* **read** *primitive. Let us replace in the net above each sequence of* **in** *and* **out** *with a* **read**:

$$\begin{array}{rcl} N & = & s_1 :: k_1 : \langle foo \rangle \parallel s_2 :: k_2 : \mathbf{read}(foo)@s_1.P \\ & & \qquad\qquad \parallel s_3 :: k_3 : \mathbf{read}(foo)@s_1.P' \end{array}$$

By executing the two **read** *actions (the order is now irrelevant), the net N evolves to:*

$$\begin{array}{l}(\nu k_2', k_3')\ (\ s_1 :: k_1 : \langle foo \rangle \\ \qquad\qquad \| \ \ s_2 :: k_2' : P \ \mid [k_2 : \mathbf{read}(foo)@s_1.P; k_1; k_2'] \\ \qquad\qquad \| \ \ s_3 :: k_3' : P' \mid [k_3 : \mathbf{read}(foo)@s_1.P'; k_1; k_3']\ )\end{array}$$

Any of the two processes, say s_2, can undo the executed **read** *action without affecting the execution of the other one. Thus, applying rule (ReadRev) we get:*

$$\begin{array}{l}(\nu k_2', k_3')\ (\ s_1 :: k_1 : \langle foo \rangle \\ \qquad\qquad \| \ \ s_2 :: k_2 : \mathbf{read}(foo)@s_1.P \\ \qquad\qquad \| \ \ s_3 :: k_3' : P' \mid [k_3 : \mathbf{read}(foo)@s_1.P'; k_1; k_3']\ )\end{array}$$

3.3. Basic properties.

We now show that rμKlaim respects the μKlaim semantics (Lemmas 2 and 3), and that it is causally consistent (Theorem 1). All the proofs can be found in Appendix A.

We first introduce some auxiliary definitions.

Definition 5. *We define the set of resources (keys or localities)* consumed*,* read*, and* produced *in a memory μ, denoted respectively* $\mathtt{cons}(\mu)$*,* $\mathtt{read}(\mu)$*, and* $\mathtt{prod}(\mu)$*, as follows. We extend the definition also to connectors.*

μ	$\mathtt{cons}(\mu)$	$\mathtt{read}(\mu)$	$\mathtt{prod}(\mu)$
$[k : \mathbf{out}_\gamma(t)@s.P; k''; k']$	$\{k\}$	$\{s\}$	$\{k', k''\}$
$[k : \mathbf{eval}_\gamma(Q)@s.P; k''; k']$	$\{k\}$	$\{s\}$	$\{k', k''\}$
$[k : \mathbf{in}_\gamma(T)@s.P; h : \langle et \rangle; k']$	$\{k, h\}$	$\{s\}$	$\{k'\}$
$[k : \mathbf{read}_\gamma(T)@s.P; h; k']$	$\{k\}$	$\{s, h\}$	$\{k'\}$
$[k : \mathbf{newloc}_\gamma(s).P; k']$	$\{k\}$	$\emptyset$	$\{s, k'\}$
$k \prec (k', k'')$	$\{k\}$	$\emptyset$	$\{k', k''\}$

Consistent nets are well formed, and satisfy the completeness property below.

Definition 6 (Complete net)**.** *A net N is complete, written* $\mathtt{complete}(N)$*, if:*

– *for each key k produced or read by a memory/connector of N there exists in N (possibly inside a memory) either a process $k : P$, or an evaluated tuple $k : \langle et \rangle$, or a connector $k \prec (k_1, k_2)$ and, unless all the occurrences of k are read by the memory/connector where they occur, k is bound in N;*

– *for each key k occurring at least once as argument of a* **roll**(k) *in N there exists in N a memory with key k;*

– *for each site s produced by a memory of N there exists in N a node named s, and s is bound in N.*

Lemma 1 (Well-formedness)**.** *For each consistent net N:*

1. *all keys occurring in N attached to processes or evaluated tuples (possibly in a memory) are distinct, and*
2. *N is complete.*

We now show that RμKLAIM is a *conservative* extension of μKLAIM. Indeed, keys, references, memories and connectors are just decorations if we only consider forward reductions. As a consequence, from a RμKLAIM net we can derive a μKLAIM net by removing history and causality information. This is formalized by the erasing function ϕ_N (and the auxiliary functions ϕ_C and ϕ_P acting, respectively, on components and processes) defined in Table 9. Let us note that the function ϕ_P deletes all the occurrences of **roll**(ι) processes, and all the references γ attached to the actions.

The following lemmas state the correspondence between RμKLAIM forward semantics and μKLAIM semantics.

Lemma 2. *Let N and M be two* RμKLAIM *nets such that $N \mapsto M$. Then $\phi_N(N) \to \phi_N(M)$.*

Lemma 3. *Let R and S be two μ*KLAIM *nets such that $R \to S$. Then for all consistent* RμKLAIM *nets M such that $\phi_N(M) = R$ there exists a* RμKLAIM *net N such that $M \mapsto N$ and $\phi_N(N) \equiv S$.*

The Loop lemma below shows that each reduction has an inverse.

Lemma 4 (Loop lemma)**.** *For all consistent* RμKLAIM *nets N and M, the following holds:*

$$N \mapsto M \iff M \rightsquigarrow N.$$

$$\phi_N(\mathbf{0}) = \mathbf{0} \qquad \phi_N(N_1 \parallel N_2) = \phi_N(N_1) \parallel \phi_N(N_2)$$
$$\phi_N(s :: \mathbf{empty}) = s :: \mathbf{nil} \qquad \phi_N(s :: C) = s :: \phi_C(C)$$
$$\phi_N((\nu k)\ N) = \phi_N(N) \qquad \phi_N((\nu s)\ N) = (\nu s)\ \phi_N(N)$$

$$\phi_C(k : P) = \phi_P(P) \qquad \phi_C(C_1 \mid C_2) = \phi_C(C_1) \mid \phi_C(C_2)$$
$$\phi_C(k : \langle et \rangle) = \langle et \rangle \qquad \phi_C(k \prec (k_1, k_2)) = \mathbf{nil}$$
$$\phi_C(\mu) = \mathbf{nil}$$

$$\phi_P(P_1 \mid P_2) = \phi_P(P_1) \mid \phi_P(P_2) \qquad \phi_P(a_\gamma.P) = a.\phi_P(P)$$
$$\phi_P(\mathbf{roll}(\iota)) = \mathbf{nil} \qquad \phi_P(A) = A$$
$$\phi_P(\mathbf{nil}) = \mathbf{nil}$$

Table 9: Erasing functions

We now move to the proof that RμKLAIM is indeed causally consistent. While the general strategy follows the approach in [5], the technicalities differ substantially because of the more complex causality structure of RμKLAIM.

In a forward reduction $N \mapsto M$ we call *forward memory* the memory μ created by that reduction, i.e., μ does not occur in N and occurs in M. Similarly, in a backward reduction $N \rightsquigarrow M$ we call *backward memory* the memory μ deleted by that reduction, i.e., μ occurs in N and does not occur in M. We call *transition* a triplet of the form $N \xrightarrow{\mu_\mapsto} M$, or $N \xrightarrow{\mu_\rightsquigarrow} M$, where N and M are consistent nets, and μ is the forward/backward memory of the reduction. We call N the *source* of the transition, M its *target*. We let η range over labels $\mu_\mapsto$ and $\mu_\rightsquigarrow$. If $\eta = \mu_\mapsto$, then $\eta_\bullet = \mu_\rightsquigarrow$, and vice versa. Without loss of generality we restrict our attention to transitions derived without using α-conversion. We also assume that when structural rule *(Split)* is applied from left to right creating a connector $k \prec (k_1, k_2)$, there is a fixed function determining k_1 and k_2 from k, and that different values of k produce different values of k_1 and k_2. This is needed to avoid that the same name is used with different meanings (cfr. the definition of *closure* below). Two transitions are *coinitial* if they have the same source, *cofinal* if they have the same target, and *composable* if the target of the first one is the source of the second one. A sequence of transitions such that each pair of consecutive transitions is composable is called a *trace*. We let δ range over transitions and θ range over traces. If δ is a transition then $\delta_\bullet$ denotes its inverse. Notions of source, target and composability extend naturally to traces. We denote with ϵ_M the empty trace with source M, and with $\theta_1; \theta_2$

the composition of two composable traces θ_1 and θ_2. We define the *closure* w.r.t. a net N of a key k as

$$\mathtt{clos}_N(k) = \begin{cases} \{k\} \cup \mathtt{clos}_N(k_1) \cup \mathtt{clos}_N(k_2) & \text{if } k \prec (k_1, k_2) \text{ occurs in } N, \\ \{k\} & \text{otherwise.} \end{cases}$$

We define the closure of a set K of keys as

$$\mathtt{clos}_N(K) = \bigcup_{k \in K} \mathtt{clos}_N(k).$$

The closure captures the fact that the connector $k \prec (k_1, k_2)$ means that resources k_1 and k_2 are part of resource k.

We extend the definition of sets of resources consumed, read and produced from memories to transition labels as follows:

$$\begin{array}{lll} \mathtt{cons}(\mu_{\mapsto}) = \mathtt{cons}(\mu) & \mathtt{read}(\mu_{\mapsto}) = \mathtt{read}(\mu) & \mathtt{prod}(\mu_{\mapsto}) = \mathtt{prod}(\mu) \\ \mathtt{cons}(\mu_{\rightsquigarrow}) = \mathtt{prod}(\mu) & \mathtt{read}(\mu_{\rightsquigarrow}) = \mathtt{read}(\mu) & \mathtt{prod}(\mu_{\rightsquigarrow}) = \mathtt{cons}(\mu) \end{array}$$

Essentially, forward transitions consume, read and produce sets of resources as specified by their memories, while for backward transitions the sets of consumed and produced resources are swapped w.r.t. their memories.

Definition 7 (Concurrent transitions)**.** *Two coinitial transitions* $M \xrightarrow{\eta_1} N_1$ *and* $M \xrightarrow{\eta_2} N_2$ *are* in conflict *if:*

- $\mathtt{clos}_{M \| N_1 \| N_2}(\mathtt{cons}(\eta_1)) \cap (\mathtt{clos}_{M \| N_1 \| N_2}(\mathtt{cons}(\eta_2)) \cup \mathtt{read}(\eta_2)) \neq \emptyset$ *or,*
- $\mathtt{clos}_{M \| N_1 \| N_2}(\mathtt{cons}(\eta_2)) \cap (\mathtt{clos}_{M \| N_1 \| N_2}(\mathtt{cons}(\eta_1)) \cup \mathtt{read}(\eta_1)) \neq \emptyset$.

Two coinitial transitions are concurrent *if they are not in conflict.*

Essentially, two transitions are in conflict if they both consume or read the same resource, and at most one of them reads it. Consuming must take into account the closure. The use of the closure involves a few subtleties, thus we present an example to clarify it.

Example 2. *Consider a net with a single locality s containing the component $k : P$ where:*

$$P = \mathbf{out}(foo)@s.Q$$
$$Q = \mathbf{out}(foo1)@s \mid \mathbf{out}(foo2)@s$$

The net can evolve as follows:

$$s :: k{:}P \mapsto (\nu k', k'')(s :: k' {:} Q \mid [k : \mathbf{out}(foo)@s.Q; k''; k'] \parallel s :: k'' {:} \langle foo \rangle) = N$$

The resulting net N can, e.g., undo the step just performed.

$$N \rightsquigarrow (\nu k')\,(s :: k : P \parallel s :: \mathbf{empty}) \equiv s :: k : P$$

Another option is to execute the action $\mathbf{out}(foo1)@s$:

$$\begin{aligned}
N \equiv & (\nu k', k'', k_1', k_2') \\
& (s :: k' \prec (k_1', k_2') \mid k_1' : \mathbf{out}(foo1)@s \mid k_2' : \mathbf{out}(foo2)@s \mid k'' : \langle foo \rangle \\
& \quad \mid [k : \mathbf{out}(foo)@s.Q; k''; k']) \\
\mapsto & (\nu k', k'', k_1', k_2', k''', k'''') \\
& (s :: k' \prec (k_1', k_2') \mid k''' : \mathbf{nil} \mid k_2' : \mathbf{out}(foo2)@s \\
& \quad \mid [k : \mathbf{out}(foo)@s.Q; k''; k'] \mid [k_1' : \mathbf{out}(foo1)@s; k''''; k'''] \\
& \parallel s :: k'' : \langle foo \rangle \mid k'''' : \langle foo1 \rangle)
\end{aligned}$$

We highlighted in the derivation the use of structural rule (Split). The sets of consumed resources of the two transitions with source N are, respectively, $\{k', k''\}$ and $\{k_1'\}$. Only the use of the closure on the first set, adding k_1' (as well as k_2') to the set of resources used by the first transition, allows one to find the conflict between the two transitions.

The definition of concurrent transitions is validated by the following lemma.

Lemma 5 (Square lemma)**.** *If $\delta_1 = M \xrightarrow{\eta_1} N_1$ and $\delta_2 = M \xrightarrow{\eta_2} N_2$ are two coinitial concurrent transitions, then there exist two cofinal transitions $\delta_2/\delta_1 = N_1 \xrightarrow{\eta_2} N$ and $\delta_1/\delta_2 = N_2 \xrightarrow{\eta_1} N$.*

In order to show that reversibility in RμKLAIM is causally consistent we define causal equivalence.

Definition 8 (Causal equivalence)**.** Causal equivalence*, denoted by $\asymp$, is the least equivalence relation between traces closed under composition that obeys the following rules:*

$$\delta_1; \delta_2/\delta_1 \asymp \delta_2; \delta_1/\delta_2 \qquad \delta; \delta_\bullet \asymp \epsilon_{\mathrm{source}(\delta)} \qquad \delta_\bullet; \delta \asymp \epsilon_{\mathrm{target}(\delta)}$$

Intuitively, causal equivalence equates traces that differ only for swaps of concurrent actions and simplifications of inverse actions. The next result shows that there is a unique way to go from one state to another up to causal equivalence. This means, on the one hand, that causal equivalent traces can be reversed in the same ways, and, on the other hand, that traces which are not causal equivalent lead to distinct nets.

Theorem 1 (Causal consistency)**.** *Let θ_1 and θ_2 be coinitial traces, then $\theta_1 \asymp \theta_2$ if and only if θ_1 and θ_2 are cofinal.*

Another relevant property of our calculus is backward confluence. In its specification, we denote with $\rightsquigarrow^*$ the reflexive and transitive closure of $\rightsquigarrow$.

Lemma 6 (Backward confluence)**.** *Let M be a consistent net. If $M \rightsquigarrow^* M_1$ and $M \rightsquigarrow^* M_2$ then there exists M' such that $M_1 \rightsquigarrow^* M'$ and $M_2 \rightsquigarrow^* M'$.*

4. Semantics of rollback

In this section we define the *roll backward semantics* of RμKLAIM, which corresponds to the execution of the **roll** operator. Together with the forward semantics of RμKLAIM defined in the previous section, it gives rise to the *roll operational semantics* of RμKLAIM, which, as we will show, satisfies the specification given in the Introduction.

The roll backward semantics builds on the step-by-step backward semantics of RμKLAIM (Table 8), as follows.

Definition 9 (RμKLAIM roll semantics)**.** *The roll operational semantics of* RμKLAIM *consists of the* forward reduction relation $\mapsto$ *defined in Table 7 and the* backward reduction relation $\overset{k}{\rightsquigarrow}_{\mathbf{r}}$*, defined as the smallest evaluation-closed relation satisfying the following rule (we recall that $\rightsquigarrow^*$ is the reflexive and transitive closure of $\rightsquigarrow$) :*

$$\frac{M = (\nu \tilde{z})s :: k' : \mathbf{roll}(k) \,\|\, N \qquad M \rightsquigarrow^* M' \not\rightsquigarrow \qquad k <: M \qquad \mathtt{complete}(M)}{M \;\overset{k}{\rightsquigarrow}_{\mathbf{r}}\; M'} \textit{(Roll)}$$

The reductions derived by rule *(Roll)* feature a label k (which needs to be preserved by closure under contexts and structural congruence). This is just an annotation that helps to study the properties of the semantics, and has no impact on the behavior. We will drop the label when not needed.

A backward reduction corresponds to the execution of a **roll** command. Since all the occurrences of references γ are bound, when a **roll** becomes enabled its argument is always a key k, uniquely identifying the memory created by the action to be undone. Thus, backward reductions are defined by the semantics of **roll**(k). As hinted at in the Introduction, the **roll**(k) operator should undo *all* the actions depending on the target action k to ensure causal-consistency, and *only* them to ensure minimality. The *all* part is captured by the notion of completeness (Definition 6), and the *only* part by a notion of k-dependence (written $<:$) defined below. The term M in rule *(Roll)* captures the part of the net involved in the reduction, thus it contains all the dependences of the target action k. Then, by exploiting the step-by-step undoing facility of the $\rightsquigarrow$ relation, all the actions contained in M depending on k are undone, restoring the net to a point in which the action pointed by k is available again.

We now formally define the k-dependence relation used in the definition of the semantics, together with examples clarifying it. To this end, we need the auxiliary notion of causal dependence among keys and localities, and a normal form result for RμKLAIM nets.

Definition 10 (Causal dependence)**.** *Let N be a* RμKLAIM *net and let T_N be the set of keys and localities in N. The relation $<:_N$ on T_N is the smallest preorder (i.e., reflexive and transitive relation) such that for each memory/connector μ in N and for each pair of keys/localities (z, z') such that $z \in \mathtt{cons}(\mu) \cup \mathtt{read}(\mu)$ and $z' \in \mathtt{prod}(\mu)$, we have $z <:_N z'$.*

Note that for action **out** the continuation and the evaluated tuple produced depend on the action (i.e., on the key identifying the action and on its target locality), while for actions **in** and **read** the continuation depends on both the action and the evaluated tuple. Dependences for action **eval** are similar to the ones for action **out** (with the spawned process replacing the evaluated tuple produced by **out**), while for action **newloc** both the continuation and the created site depend on the action. Finally, the dependences for a connector are as expected: children processes depend on their parent.

We now present the normal form result.

Lemma 7 (Normal form)**.** *For any* rμKlaim *net* N*, we have:*

$$N \equiv (\nu\tilde{z})\Big(\Big\|_{s\in S} s :: \prod_{i\in I}(k_i : P_i) \mid \prod_{j\in J}[k_j : a_j.P_j; \xi_j] \mid \prod_{h\in H}(k_h \prec (k'_h, k''_h)) \mid \prod_{x\in X}(k_x : \langle et_x\rangle) \mid \prod_{w\in W}[k_w : \mathbf{in}_{\gamma_w}(T_w)@s_w.P_w; h_w : \langle t_w\rangle; k'_w] \mid \prod_{y\in Y}[k_y : \mathbf{read}_{\gamma_y}(T_y)@s_y.P_y; h_y; k'_y]\Big)$$

where action a_j *is neither* **in** *nor* **read** *for every* $j \in J$*.*

The proof of the lemma is trivial, using structural congruence.

We can finally define k-dependence.

Definition 11 (k-dependence)**.** *Let* N *be a* rμKlaim *net in normal form. Net* N *is* k*-dependent, written* $k <: N$*, if, using the notation in Lemma 7:*

- *for every* $i \in I \cup J \cup H \cup X$ *we have* $k <:_N k_i$*;*
- *for every* $i \in W \cup Y$ *we have* $k <:_N k_i$ *or* $k <:_N h_i$*;*
- *for every* $z \in \tilde{z}$ *we have* $k <:_N z$*.*

Intuitively, for the given net, the first condition says that keys of active processes, of memories for actions different from **in** and **read**, of connectors, and of available evaluated tuples, depend on the key k. The second condition says that in memories for actions **in** and **read** either the key of the memory or the one of the evaluated tuple accessed depends on k. When looking only at memories/connectors, these two conditions say that at least one of the keys consumed or read by each memory/connector depends on k. Finally, the third condition says that all generated keys and created sites depend on k.

We show now a few examples of rollback.

Example 3. *Consider the following net:*

$$M = s :: k : \mathbf{out}_\gamma(foo)@s.\mathbf{in}(foo1)@s.\mathbf{roll}(\gamma) \mid k' : \langle foo1\rangle$$

The net can perform two forward steps reducing to:

$$\begin{aligned} N = (\nu k'', k''', k'''')\,(s :: k'''' : \mathbf{roll}(k) \mid k''' : \langle foo \rangle \\ \mid [k : \mathbf{out}_\gamma(foo)@s.\mathbf{in}(foo1)@s.\mathbf{roll}(\gamma); k'''; k''] \\ \mid [k'' : \mathbf{in}(foo1)@s.\mathbf{roll}(k); k' : \langle foo1 \rangle; k'''']) \end{aligned}$$

Performing $\mathbf{roll}(k)$ *should lead back to the initial state. Both* $k <: N$ *and* $\texttt{complete}(N)$ *hold, and*

$$\begin{aligned} N \rightsquigarrow (\nu k'', k''')\,(s :: k'' : \mathbf{in}(foo1)@s.\mathbf{roll}(k) \mid k''' : \langle foo \rangle \mid k' : \langle foo1 \rangle \\ \mid [k : \mathbf{out}_\gamma(foo)@s.\mathbf{in}(foo1)@s.\mathbf{roll}(\gamma); k'''; k''] \\ \rightsquigarrow s :: k : \mathbf{out}_\gamma(foo)@s.\mathbf{in}(foo1)@s.\mathbf{roll}(\gamma) \mid k' : \langle foo1 \rangle = M \not\rightsquigarrow \end{aligned}$$

Therefore, we have that $N \rightsquigarrow_{\mathbf{r}} M$ *as desired. Note that the evaluated tuple* $k' : \langle foo1 \rangle$ *is released.*

Example 4. *Consider the following net:*

$$s :: k : \mathbf{out}_\gamma(foo)@s.\mathbf{roll}(\gamma) \parallel s' :: k' : \mathbf{in}(foo)@s$$

After the **out** *of the evaluated tuple* $\langle foo \rangle$ *at locality* s *followed by the* **in** *of the same evaluated tuple the net becomes:*

$$\begin{aligned} &(\nu k'', k''', k'''') \\ &(s :: k'' : \mathbf{roll}(k) \mid [k : \mathbf{out}_\gamma(foo)@s.\mathbf{roll}(\gamma); k'''; k''] \\ &\quad \parallel s' :: k'''' : \mathbf{nil} \mid [k' : \mathbf{in}(foo)@s; k''' : \langle foo \rangle; k'''']) \end{aligned}$$

Performing $\mathbf{roll}(k)$ *restores the initial net, by releasing the content of the target memory as well as the parallel* **in**. *The* **in** *action indeed depends on the* **out** *one as the former has consumed the evaluated tuple produced by the latter; thus, the* **in** *must be undone before undoing the* **out**.

The execution of a given rollback is deterministic. To prove this property we start with an auxiliary lemma.

Lemma 8. *Let net* M *be a subterm of a consistent net. If* $M \rightsquigarrow^* M' \not\rightsquigarrow$ *and* $\texttt{complete}(M)$ *then* M' *contains no memory.*

Proof. We proceed by induction on the length n of the reduction $M \rightsquigarrow^* M'$.

In the base case ($n = 0$), we have that $M \not\rightsquigarrow$ with $\texttt{complete}(M)$. Assume, by contradiction, that there is at least a memory in M. One can order the memories as follows: $\mu_1 \leq \mu_2$ if $\mu_1 = \mu_2$ or μ_1 consumes or reads a resource produced by μ_2. It is easy to see that such a relation is a partial order. Now, take a memory μ_0 which is minimal in this partial order (i.e., there exist no memory $\mu' \neq \mu_0$ such that $\mu' \leq \mu_0$). We have a case analysis on the action inside μ_0. We consider just the case of action **out**, i.e., μ_0 is $[k : \mathbf{out}_\gamma(t)@s.P; k''; k']$, the others being similar. Thanks to completeness, there are resources corresponding to k' and k'', and there is also a restriction for k'' given that the resource is produced (and not only read). In particular, since the memory is minimal and M is a subterm of a consistent net, those resources are at top level and the occurrence of k'' outside the memory is unique. Thanks to consistency, k'' is attached to an evaluated tuple in the target locality of the memory (i.e., s), and k' to a process in the same locality of the memory (it may be attached to a connector, but in this case a process can be rebuilt using structural rule *(Split)*). Then, rule *(OutRev)* can be applied contradicting the hypothesis $M \not\rightsquigarrow$.

Let us consider the inductive case, i.e., $M \rightsquigarrow M_1 \rightsquigarrow^* M' \not\rightsquigarrow$ and $\texttt{complete}(M)$. We only need to show $\texttt{complete}(M_1)$, then the thesis will follow by inductive hypothesis. It is easy to see by inspection on backward rules that only resources produced by the corresponding memory are removed: these resources cannot be produced or read by other memories or connectors. Also, the dropped restrictions concern resources not used elsewhere. Finally, no **roll** could target the memory, since **roll**s targeting the memory can only occur in the continuation of the memory. As a consequence, completeness is preserved by the backward step, and the thesis follows.

□

We can now prove determinism.

Proposition 1 (Determinism)**.** *Given a consistent net N, if $N \overset{k}{\rightsquigarrow}_{\mathbf{r}} N'$ and $N \overset{k}{\rightsquigarrow}_{\mathbf{r}} N''$ then $N' \equiv N''$.*

Proof. $N \overset{k}{\rightsquigarrow}_{\mathbf{r}} N'$ implies that $N = (\nu \tilde{z}')(M \parallel M_r)$, with $M_r = (\nu \tilde{z})s :: k' : \mathbf{roll}(k) \parallel M_1$, $\texttt{complete}(M_r)$ and $k <: M_r$.

We have to show that M_r is uniquely defined up-to structural congruence. Thus, if there are two nets, M_r' and M_r'', satisfying the conditions

above (stated for the net M_r) then $M_r' \equiv M_r''$. Because of completeness a memory with key k should occur in both M_r' and M_r'', and recursively all its consequences. Let us show that these are all the terms allowed by the causal dependence condition. Take one such term with key k'', and take the shortest derivation for $k <:_N k''$. We proceed by induction on the length n of such a derivation. In the base case ($n = 0$), $k = k''$ and the term is the target memory of the **roll**. In the inductive case, by definition of causal dependence, k'' is a resource generated by a memory, and there is a shorter derivation for a resource with key k''' consumed or read by the memory. Hence, the resource with key k''' is in the net, and by completeness also the resource with key k'' is in the net. Since the two nets M_r' and M_r'' contain the same resources, they are structural congruent.

Now, we can safely focus on the subterm M_r of the net N. Let m be the number of memories in M_r. Given that each backward reduction consumes a memory, we have that there exists M_0 such that $M_0 \not\rightsquigarrow$ and $M_r \rightsquigarrow^* M_0$ in at most m steps. Thanks to Lemma 8, the computation has exactly m steps. Assume that there are two such computations. Thanks to Lemma 6 there should be an M' closing the diagram, but given that no further reductions are possible the two final nets should be the same, up to the fact that the two reductions may choose different representatives inside the equivalence class defined by structural congruence. The thesis is preserved by closure under context and structural congruence.

□

We now prove that the different requirements of the specification given in the Introduction are satisfied. To this end, we need to formalize them in a precise way. When a **roll** is executed, its argument is a key k. Therefore, we decide to state the requirements in terms of keys k, instead of references γ as done in the Introduction (where, for the sake of presentation, keys had not been introduced yet). However, given the fact that each key corresponds to a unique γ (the one labeling the action inside the memory with key k) this will not make any difference. Also, note that it is important to restrict to computations that avoid α-conversion and similar renamings due to rule *(Split)*, since both keys and references are bound and could otherwise change. Finally, in the formalization of the requirements, we denote with $\mapsto^*$ the reflexive and transitive closure of $\mapsto$.

Proposition 2 (Redoability)**.** *Let M be a consistent net. If $M \overset{k}{\rightsquigarrow}_{\mathbf{r}} M'$ then M' contains a process with key k which is executable.*

Proof. The hypothesis $M \overset{k}{\rightsquigarrow}_{\mathbf{r}} M'$ implies, by Definition 9, that $M = (\nu\tilde{z}')(N' \parallel M_r)$ with $M_r = (\nu\tilde{z})s :: k' : \mathbf{roll}(k) \parallel N$ and $M_r \rightsquigarrow^* M_r' \not\rightsquigarrow$, and both $k <: M_r$ and $\texttt{complete}(M_r)$, where, by evaluation closure, M_r' is a subterm of M'. The completeness of M_r and the fact that a process $\mathbf{roll}(k)$ occurs in M_r guarantee the existence of a memory in M_r with key k (*cfr.* Definition 6). The consistency of M guarantees that no other process or memory with key k occurs in M_r, given that there is a memory with the same k. Now we show that such a memory is removed by the sequence of backward reductions, and in particular by the last reduction in the sequence. Notice that the removal of a memory which consumes or read a resource k' may be performed only if such a resource occurs in M_r (outside any memory), thus no memory with the same key occurs. Namely, every memory tagged with k' such that $k <:_{M_r} k'$ must be removed before removing a memory tagged with k. The k-dependence of M_r, i.e., $k <: M_r$, implies then that every other memory in M_r must be removed before removing the memory tagged by k. By Lemma 8 we know that the application of the *(Roll)* rule causes indeed the removal of all the memories. Thus, the last backward reduction causes the removal of the memory tagged by k. As the removal of the memory tagged by k implies the release of the corresponding consumed resources, then we have that M_r' contains a process with key k. As M_r' is a subterm of M', we obtain that M' contains such a process as well. Finally, by the Loop Lemma, the undone action can be redone, thus the process with key k is executable. □

Proposition 3 (Causal consistency)**.** *Let M be a consistent net. If $M \overset{k}{\rightsquigarrow}_{\mathbf{r}} M'$ then M' contains no consequences of k.*

Proof. The hypothesis $M \overset{k}{\rightsquigarrow}_{\mathbf{r}} M'$ implies, by Definition 9, that $k <: N$ and $\texttt{complete}(N)$, where N is the subterm of M to which rule *(Roll)* was applied. By Definition 11 and completeness of N, all the consequences of k were in the term N to which rule *(Roll)* was applied. Take a resource in N with key k'. Consider the shortest derivation for $k <:_N k'$. We will show that such a resource is consumed, unless it has key k. If the key is not k, but, say k', the derivation has length at least 1. The last step of the derivation implies that there is a memory or connector μ in N such that k' is produced by the memory/connector. If it is a connector, the resource is removed by applying structural rule *(Split)* (this needs to be done, otherwise the step corresponding to the memory with key k cannot be undone). If it is a memory, the step

removing this memory (which is performed thanks to Lemma 8) consumes the resource as required. □

Proposition 4 (Correctness)**.** *Let M be a consistent net. If $M \rightsquigarrow_{\mathbf{r}} M'$ then $M' \mapsto^* M$.*

Proof. From the hypothesis $M \rightsquigarrow_{\mathbf{r}} M'$ and Definition 9 (in particular from the premise of rule *(Roll)* concerning the sequence of step-by-step reductions and from the evaluation closure), we have that $M \rightsquigarrow^* M'$. Then, the thesis follows from the Loop Lemma. □

Proposition 5 (Minimality)**.** *Let M be a consistent net. If $M \overset{k}{\rightsquigarrow}_{\mathbf{r}} M'$ then there is no consistent net M'' which enjoys redoability (it contains a process with key k which is executable), causal consistency (it contains no consequences of k), and correctness ($M'' \mapsto^* M$), and such that $M'' \mapsto^* M$ is shorter than $M' \mapsto^* M$.*

Proof. Assume that there is such an M''. Thanks to *correctness* $M'' \mapsto^* M$. Thanks to the Loop Lemma $M \rightsquigarrow^* M''$. Thanks to *redoability* the action with key k needs to be undone during the computation. In fact, at any time there is at most one process with key k, and in order to be executed it needs to be extracted from the memory targeted by the **roll**, hence the corresponding action needs to be undone. Thanks to *causal consistency*, all its consequences need to be undone too. Now, from the hypothesis $M \overset{k}{\rightsquigarrow}_{\mathbf{r}} M'$, by Definition 9, we have that $M_r \rightsquigarrow^* M'_r$ with $M = (\nu\tilde{z})(N \parallel M_r)$, $M' = (\nu\tilde{z})(N \parallel M'_r)$ and $k <: M_r$. Since $k <: M_r$ then all the actions in M_r are consequences of k and thus need to be undone, hence at least a number n of backward steps equal to the number of memories in M_r need to be done in $M \rightsquigarrow^* M''$. Thanks to the Loop Lemma, this is also the number of steps in $M'' \mapsto^* M$. However, thanks to Lemma 8 this is exactly the number of steps in $M_r \rightsquigarrow^* M'_r$ (given that each step removes exactly one memory). Due to evaluation closure, this is also the exact number of steps in $M \rightsquigarrow^* M'$. Now, thanks to the Loop Lemma, this is also the number of steps in $M' \mapsto^* M$. Therefore, $M'' \mapsto^* M$ is not shorter than $M' \mapsto^* M$, which is a contradiction. □

To sum up, the roll operational semantics that we defined for RμKLAIM satisfies the specification given in the Introduction.

5. A speculative execution scenario

In this section, we apply our reversible language to a simplified but realistic producer-consumers scenario, inspired by [16], that involves a form of speculative execution. This kind of scenario is particularly relevant in our setting. In fact, on the one hand, speculative execution can be significantly effective in improving the performance of a system with distributed, concurrent and shared-memory nature. Indeed, the most highly parallel and fastest discrete event simulation benchmark executed till 2013 has made essential use of it [17]. On the other hand, reversible execution permits to relieve programmers from the burden of properly undoing actions depending on an incorrect speculation.

In the considered scenario, a *producer* provides values to a number of speculative *consumers* (all of them read each value, for possibly different purposes) that, in their own turn, pass the values (or predictions of them) to the corresponding final *users*. More specifically, the production of values requires a fairly long time, thus the producer provides a partial information that is used by consumers to predict the values ahead of time. These predictions are passed to the users in order to enable their execution speculatively and concurrently with the execution of the producer. Once the production completes, the actual value and the predicted ones are compared; if a prediction is precise enough, we gain performance because the execution of the producer and the corresponding user overlapped in time, otherwise rollback is used to move consumers and users back to a correct state, in order to re-execute them.

The scenario informally described above, where for simplicity's sake we consider just two consumer-user pairs, is rendered in RμKLAIM as follows:

$$
\begin{array}{l}
\mathit{producer} \;::\; (k : A_P \;\mid\; k' : \langle \text{“}\mathit{counter}\text{”}, 0 \rangle) \\
\parallel\; \mathit{consumer}_1 \;::\; (k_1 : A_{C_1} \;\mid\; k'_1 : \langle \text{“}\mathit{counter}\text{”}, 0 \rangle) \;\parallel\; \mathit{user}_1 \;::\; (k_3 : A_{U_1}) \\
\parallel\; \mathit{consumer}_2 \;::\; (k_2 : A_{C_2} \;\mid\; k'_2 : \langle \text{“}\mathit{counter}\text{”}, 0 \rangle) \;\parallel\; \mathit{user}_2 \;::\; (k_4 : A_{U_2})
\end{array}
$$

Essentially, each participant in the scenario corresponds to a RμKLAIM node running a specific process according to its role. To guarantee that values are read by each consumer in the same order as they were produced, producer and consumers are equipped with counters (that in RμKLAIM are, naturally, represented as evaluated tuples).

The process describing the producer's behavior is defined as:

$$\begin{aligned} A_P \triangleq\ & \mathbf{in}(\text{“}\mathit{counter}\text{”}, !c)@\mathit{producer}. \\ & \mathbf{out}(\text{“}\mathit{value}\text{”}, \mathit{false}, p(c), c)@\mathit{producer}. \\ & \mathbf{out}(\text{“}\mathit{value}\text{”}, \mathit{true}, r(c), c)@\mathit{producer}. \\ & \mathbf{out}(\text{“}\mathit{counter}\text{”}, c+1)@\mathit{producer}.\ A_P \end{aligned}$$

The producer, after retrieving the current value of its counter, makes available a partial information concerning the value on production. This is a local evaluated tuple of the form $\langle \text{“}\mathit{value}\text{”}, \mathit{false}, v_{\mathit{partial}}, v_c \rangle$ where the first field is just a string, the second field is a boolean indicating that the status of the data value is not final, the third field is the partial value, and the fourth is the counter value. The computation of the partial data value is abstracted by means of function $p(i)$, which returns the partial information of the i-th value. When the producer has terminated the computation, it makes available in its tuple space the final result, i.e., a “*value*” evaluated tuple with second field set to *true*; notably, such evaluated tuple is correlated to the corresponding partial value by means of the counter value, which is the same for the two evaluated tuples. Again, the computation of the data is abstracted by means of a function, that is $r(\cdot)$. Afterwards, the counter is increased and the process restarts in order to produce a new value.

The following is the definition of the i-th consumer process:

$$\begin{aligned} A_{C_i} \triangleq\ & \mathbf{in}(\text{“}\mathit{counter}\text{”}, !c)@\mathit{consumer}_i. \\ & \mathbf{read}_\gamma(\text{“}\mathit{value}\text{”}, !\mathit{final}, !d, c)@\mathit{producer}. \\ & \mathbf{if}\ (\mathit{final})\ \mathbf{then}\ \{\ \mathbf{out}(\text{“}\mathit{value}\text{”}, d)@\mathit{user}_i. \\ & \qquad\qquad \mathbf{out}(\text{“}\mathit{counter}\text{”}, c+1)@\mathit{consumer}_i.\ A_{C_i}\ \} \\ & \qquad \mathbf{else}\ \{\ p\ :=\ \mathit{predict}(d). \\ & \qquad\quad \mathbf{out}(\text{“}\mathit{value}\text{”}, p)@\mathit{user}_i. \\ & \qquad\quad \mathbf{read}(\text{“}\mathit{value}\text{”}, \mathit{true}, !r, c)@\mathit{producer}. \\ & \qquad\quad \mathbf{if}\ (p \not\approx r)\ \mathbf{then}\ \{\ \mathbf{roll}(\gamma)\ \} \\ & \qquad\qquad\qquad \mathbf{else}\ \{\ \mathbf{out}(\text{“}\mathit{counter}\text{”}, c+1)@\mathit{consumer}_i. \\ & \qquad\qquad\qquad\quad A_{C_i}\ \}\ \} \end{aligned}$$

A consumer starts by retrieving the current value of its counter and, then, reading the corresponding information provided by the producer. Notably, pattern-matching on c is used here to ensure that values are read in the correct order. If the evaluated tuple read corresponds to a final result, then the consumer sends it to its user, increases the counter and restarts. Otherwise,

it predicts the result (via function *predict*$(\cdot)$), sends such prediction to the user and, while the user executes, the consumer waits for the final result and compares it with the predicted value. Again, pattern-matching on the counter value is used to ensure that the read final result corresponds to the partial value previously read. If the prediction is not accurate enough (which is checked by means of operator $\not\approx$), then the consumer executes a **roll** action to go back to the status before the execution of the initial **read** action has taken place; otherwise, the consumer simply increases the counter and restarts in order to consume a new value. Notably, the **roll** action has the effect of forcing the rollback of the actions speculatively executed by the user. Instead, the **roll** has no effect on the producer and on the other consumers, which may have read the same evaluated tuple, as the undone action is a **read**.

We omit the definition of the user processes, as they are application-specific and may be different one from the other, depending on the use of the produced values each of them has to do. Anyway, considering the illustrative purpose of this scenario, their specification is not necessary.

Notice that here we have exploited two linguistic constructs that are not provided by the RμKLAIM syntax: if-then-else and assignment. Their meaning is the standard one. They are used in this scenario just to simplify the presentation, as they can be easily expressed in RμKLAIM by means of fresh names and proper combinations of **out** and **in** actions. It is also worth noticing that, after the rollback, the consumer will find available in the producer's tuple space both the evaluated tuple with the partial information and the evaluated tuple containing the final result. Since both evaluated tuples match the template of the consumer's **read** action, one of the two will be nondeterministically selected. This means that the consumer again could read the partial data value and predict the final result, rather than directly reading the final result. This situation cannot be avoided, due to the nature of reversibility embedded in RμKLAIM, which precisely restores a past state (see Loop Lemma), thus forgetting that an unsuccessful computation has been performed. In this specific case, this forces the system to go back to a configuration where the evaluated tuple with the partial data value is still present in the producer's tuple space. To avoid repeating the same erroneous computation after a rollback, an approach of reversibility allowing to specify *alternatives* should be considered. Such an approach has been studied in the context of HOπ in [18], and we discuss in Section 7 how it could be applied to RμKLAIM.

We conclude the section by considering an extension of the scenario, where also the producer performs some checks on the partial information it provides to consumers. The refined definition of the producer is as follows:

$$
\begin{array}{l}
A'_P \triangleq \mathbf{in}(\text{“}counter\text{”}, !c)@producer. \\
\qquad \mathbf{out}_\gamma(\text{“}value\text{”}, false, p(c), c)@producer. \\
\qquad \mathbf{if}\ (check(p))\ \mathbf{then}\ \{\ \mathbf{out}(\text{“}value\text{”}, true, r(c), c)@producer. \\
\qquad\qquad\qquad\qquad\qquad \mathbf{out}(\text{“}counter\text{”}, c+1)@producer.\ A'_P\ \} \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \{\ \mathbf{roll}(\gamma)\ \}
\end{array}
$$

In this case, if the check on the partial data value is negative, i.e., function $check(\cdot)$ returns *false*, a **roll** action is executed in order to undo the consequences caused by the wrong estimate of the value under production. In particular, this implies that the consumers that have read this partial data value, and hence the corresponding users, must undo what they have already done, because the data value is invalidated by the producer itself.

6. Discussion

We have defined a rollback operator for μKLAIM satisfying the specification given in the Introduction. We defined it in two steps: first we defined a step-by-step backward semantics, and then we showed how these steps can be composed to obtain the semantics of the rollback operator.

Indeed, this two-steps approach can be applied to other concurrent calculi and languages as well, and the first step has already been done for a few different calculi in the literature. More precisely, a step-by-step causal-consistent semantics has been defined for CCS in [5], for calculi in a subformat of the path format in [12], for π-calculus in [13], for HOπ in [11], and for the functional concurrent language μOz in [14]. Note that [5, 12, 13] deal with a labeled transition semantics instead of a reduction semantics, but a reduction semantics can easily be obtained by considering just the τ actions. By applying our approach to the step-by-step semantics of HOπ defined in [11] one would get the same operator which is defined by the naive semantics in [19] (the high-level semantics in [19] could be obtained by changing just redoability to consider sets of actions). However, the two definitions differ in many respects. The one in [19] is the direct equivalent of the definition we gave for μKLAIM in the conference version of the present paper [7]. Those definitions concentrate more on the result of the rollback, while the definition we give in the present paper is more related to how such a result can be computed in

terms of step-by-step undo of given actions. The current presentation is more modular and easy to adapt to any calculus, while the previous one was more tightly related to the specific calculus and tend to become more difficult to work out as far as the structure of the calculus gets more complex. Indeed, it is reasonably simple for HOπ, but less so for μKLAIM.

7. Related work and conclusion

Our rollback primitive is able to rollback the entire system to a past state where an action indicated by the programmer, and its consequences, are undone. This is somehow akin to a checkpointing and rollback schema. In distributed systems, checkpointing and rollback (also known as checkpoint-recovery) is a technique of backward recovery (see [20]) for creating fault tolerant systems. Among the recent works that have considered undo or rollback capabilities for concurrent program execution, we can single out: [21] where logging primitives are combined with process grouping to serve as basis for fault-tolerant systems, [22] where the actor model is extended with constructs to create globally-consistent checkpoints, and [23] which introduces a functional language extended with a checkpoint abstraction. Thanks to the fine-grained causality tracking implied by our reversible substrate (i.e., every action is logged), our calculus does not suffer from uncontrolled cascading rollbacks (domino effect) which may arise with [23]. In [22], checkpointing is possible only under certain conditions, and not in general cases, while we provide a built-in guarantee that, in failure-free computations, rollback is always possible (Loop Lemma) and reaches a consistent past state. Even though [21] provides several abstractions for fault-tolerant systems, whose combination may encode different recovery mechanisms, it does not provide automatic support for undoing the effects of groups of processes that abort, while our calculus directly provides a primitive to undo all the effects of a certain action.

While the aims of our rollback primitive is similar to the ones of the works above, the technical development falls in the line of research on reversibility.

The history of reversibility in a sequential setting is already quite long [24, 25], with [26] being the first work advocating the use of reversible computing as a means to achieve fail-safety in a sequential setting. Our work however concerns causal-consistent reversibility, which has been introduced in [5]. That work considered causal-consistent reversibility for CCS, introducing histories for threads to track causality information. A generalization

of the approach, based on the transformation of dynamic operators into static ones, has been proposed in [12]. Both the works are in the setting of uncontrolled reversibility, and they consider labeled semantics. Labeled semantics for uncontrolled reversibility has been also studied for π-calculus [13], while reduction semantics has been studied for HOπ [11, 15] and μOz [14]. We are closer to [11], which uses modular memories similar to ours. Controlled reversibility has been studied first in [27], introducing irreversible actions. Other approaches have been presented in [28], where energy parameters drive the evolution of the process, in [29], where a non-reversible controller drives a reversible process, and in [30] using composite actions. Our choice of a rollback operator is particularly suited to exploit reversibility to ensure reliability, since it allows one to rollback only in case of error, and by undoing the minimal amount of computation needed to restore a safe state. For an extensive survey on causal-consistent reversibility we refer to [31].

From a technical point of view, the main novelty of our work concerns the analysis of the interplay between reversibility on the one hand, and tuple-based communication on the other hand. The results we discussed correspond to some of the results in [11, 19], which were obtained in the simpler framework of HOπ.

We have not yet transported to μKLAIM all the results obtained for HOπ in [11, 19, 18]. The main missing results are an encoding from the reversible calculus into the basic one [11], a more low-level semantics for the rollback operator [19], and the introduction of alternatives to avoid repeating the same error after a rollback [18]. A full porting of the results above would need to study the behavioral theory of RμKLAIM, which is left for future work. We outline however below how the issues above can be faced.

The most natural way to add alternatives [18] to RμKLAIM is to attach them to tuples. For instance, $k : \langle foo \rangle \% \langle foo1 \rangle$ would mean "try $\langle foo \rangle$, then try $\langle foo1 \rangle$". Such an evaluated tuple behaves as $k : \langle foo \rangle$, but it becomes $k : \langle foo1 \rangle$ when it is inside a memory of an **in**/**read** action which is the target of a **roll**. The execution of such a **roll** will restore $k : \langle foo1 \rangle$ instead of the original evaluated tuple. As in HOπ, such a simple mechanism would considerably increase the expressiveness, allowing one to remember previous failed computations and avoiding cycles. For instance, in our example scenario in Section 5, when a consumer undoes the reading of an evaluated tuple containing a partial value that led to an incorrect prediction, such evaluated tuple could be replaced by an empty one. In this way, when the consumer will re-execute the **read** action, only the matching with the evaluated tuple

containing the final result will succeed, thus avoiding to repeat unnecessary predictions.

A faithful encoding of RμKLAIM into μKLAIM itself would follow the lines of [11, 15]. Its definition would be simpler than that for reversible HOπ [11, 15], since tuple spaces provide a natural storage for memories and connectors. Such encoding will pave the way to the use of KLAVA [32], a framework providing run-time support for KLAIM actions in Java, to experiment with reversible distributed applications.

A low-level semantics for RμKLAIM, more suitable to an implementation, should follow the idea of [19], based on an exploration of the causal dependences of the memory pointed by the **roll**. However, one has to deal with read dependences, and at this more concrete level the use of restriction as done in rules *(OutRev)* and *(NewRev)* is no more viable. In the first case the problem could be solved by keeping in each evaluated tuple the set of keys of processes which read the evaluated tuple. In the second case one should check that no other component with the created site exists.

Debugging is another context where reversibility plays a natural role [33]. Being able to debug a program by rewinding the execution can be very handy. In [34] we described a causal-consistent reversible debugger for μOz, a functional language based on message passing. We can exploit the results obtained in the present paper about the interplay between causality, reversibility and shared memory, to extend our debugger with shared-memory features such as mutables.

Acknowledgment

This work was partially supported by Italian MIUR PRIN Project CINA Prot. 2010LHT4KM, by the French ANR 11 INSE 007 project REVER, and by the COST Action IC1405. We thank the anonymous reviewers for their helpful comments.

References

[1] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, D. B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Comput. Surv. 34 (3) (2002) 375–408.

[2] Apple, Time Machine (OS X).
URL `https://en.wikipedia.org/wiki/Time_Machine_(OS_X)`

[3] P. A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.

[4] B. Randell, System structure for software fault tolerance, IEEE Transactions on Software Engineering 1 (2) (1975) 220–232.

[5] V. Danos, J. Krivine, Reversible communicating systems, in: P. Gardner, N. Yoshida (Eds.), CONCUR, Vol. 3170 of Lecture Notes in Computer Science, Springer, 2004, pp. 292–307.

[6] D. Gorla, R. Pugliese, Resource access and mobility control with dynamic privileges acquisition, in: J. C. M. Baeten, J. K. Lenstra, J. Parrow, G. J. Woeginger (Eds.), ICALP, Vol. 2719 of Lecture Notes in Computer Science, Springer, 2003, pp. 119–132.

[7] E. Giachino, I. Lanese, C. A. Mezzina, F. Tiezzi, Causal-consistent reversibility in a tuple-based language, in: M. Daneshtalab, M. Aldinucci, V. Leppänen, J. Lilius, M. Brorsson (Eds.), PDP, IEEE, 2015, pp. 467–475.

[8] R. De Nicola, G. Ferrari, R. Pugliese, KLAIM: A Kernel Language for Agents Interaction and Mobility, IEEE Transactions on Software Engineering 24 (5) (1998) 315–330.

[9] D. Gelernter, Generative communication in linda, ACM Trans. Program. Lang. Syst. 7 (1) (1985) 80–112.

[10] L. Bettini, V. Bono, R. De Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, B. Venneri, The Klaim Project: Theory and practice, in: C. Priami (Ed.), Global Computing, Vol. 2874 of Lecture Notes in Computer Science, Springer, 2003, pp. 88–150.

[11] I. Lanese, C. A. Mezzina, J.-B. Stefani, Reversing higher-order pi, in: P. Gastin, F. Laroussinie (Eds.), CONCUR, Vol. 6269 of Lecture Notes in Computer Science, Springer, 2010, pp. 478–493.

[12] I. Phillips, I. Ulidowski, Reversing algebraic process calculi, Journal of Logic and Algebraic Programming 73 (1-2) (2007) 70–96.

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

[14] M. Lienhardt, I. Lanese, C. A. Mezzina, J.-B. Stefani, A reversible abstract machine and its space overhead, in: H. Giese, G. Rosu (Eds.), FMOODS/FORTE, Vol. 7273 of Lecture Notes in Computer Science, Springer, 2012, pp. 1–17.

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

[16] P. Prabhu, G. Ramalingam, K. Vaswani, Safe programmable speculative parallelism, in: B. G. Zorn, A. Aiken (Eds.), PLDI, ACM, 2010, pp. 50–61.

[17] P. D. Barnes Jr., C. D. Carothers, D. R. Jefferson, J. M. LaPre, Warp speed: executing time warp on 1,966,080 cores, in: M. L. Loper, G. A. Wainer (Eds.), SIGSIM-PADS, ACM, 2013, pp. 327–336.

[18] I. Lanese, M. Lienhardt, C. A. Mezzina, A. Schmitt, J.-B. Stefani, Concurrent flexible reversibility, in: M. Felleisen, P. Gardner (Eds.), ESOP, Vol. 7792 of Lecture Notes in Computer Science, Springer, 2013, pp. 370–390.

[19] I. Lanese, C. A. Mezzina, A. Schmitt, J.-B. Stefani, Controlling reversibility in higher-order pi, in: J. Katoen, B. König (Eds.), CONCUR, Vol. 6901 of Lecture Notes in Computer Science, Springer, 2011, pp. 297–311.

[20] A. Avizienis, J.-C. Laprie, B. Randell, C. E. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Sec. Comput. 1 (1) (2004) 11–33.

[21] T. Chothia, D. Duggan, Abstractions for fault-tolerant global computing, Theor. Comput. Sci. 322 (3) (2004) 567–613.

[22] J. Field, C. A. Varela, Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments, in: J. Palsberg, M. Abadi (Eds.), POPL, ACM, 2005, pp. 195–208.

[23] L. Ziarek, S. Jagannathan, Lightweight checkpointing for concurrent ML, J. Funct. Program. 20 (2) (2010) 137–173.

[24] G. Leeman, A formal approach to undo operations in programming languages, ACM Trans. Program. Lang. Syst. 8 (1) (1986) 50–87.

[25] V. Danos, L. Regnier, Reversible, irreversible and optimal lambda-machines, Theor. Comput. Sci. 227 (1-2) (1999) 79–97.

[26] P. G. Bishop, Using reversible computing to achieve fail-safety, in: ISSRE, IEEE Computer Society, 1997, pp. 182–191.

[27] V. Danos, J. Krivine, Transactions in RCCS, in: M. Abadi, L. de Alfaro (Eds.), CONCUR, Vol. 3653 of Lecture Notes in Computer Science, Springer, 2005, pp. 398–412.

[28] G. Bacci, V. Danos, O. Kammar, On the statistical thermodynamics of reversible communicating processes, in: A. Corradini, B. Klin, C. Cîrstea (Eds.), CALCO, Vol. 6859 of Lecture Notes in Computer Science, Springer, 2011, pp. 1–18.

[29] I. Phillips, I. Ulidowski, S. Yuen, A reversible process calculus and the modelling of the ERK signalling pathway, in: R. Glück, T. Yokoyama (Eds.), RC, Vol. 7581 of Lecture Notes in Computer Science, Springer, 2012, pp. 218–232.

[30] S. Kuhn, I. Ulidowski, Towards modelling of local reversibility, in: J. Krivine, J.-B. Stefani (Eds.), RC, Vol. 9138 of Lecture Notes in Computer Science, Springer, 2015, pp. 279–284.

[31] I. Lanese, C. A. Mezzina, F. Tiezzi, Causal-consistent reversibility, Bulletin of the EATCS 114 (2014) 121–139.

[32] L. Bettini, R. De Nicola, R. Pugliese, Klava: a Java package for distributed and mobile applications, Software - Pract. Exper. 32 (14) (2002) 1365–1394.

[33] J. Engblom, A review of reverse debugging, in: System, Software, SoC and Silicon Debug, IEEE, 2012, pp. 1–6.

[34] E. Giachino, I. Lanese, C. A. Mezzina, Causal-consistent reversible debugging, in: S. Gnesi, A. Rensink (Eds.), FASE, Vol. 8411 of Lecture Notes in Computer Science, Springer, 2014, pp. 370–384.

Appendix A. Proofs of Section 3

Lemma 1 (Well-formedness)**.** *For each consistent net N:*

1. *all keys occurring in N attached to processes or evaluated tuples (possibly inside a memory) are distinct, and*
2. *N is complete.*

Proof. Being N consistent, it can be obtained by forward or backward reductions from an initial net. Then we prove the thesis by induction on the number n of reduction steps from the initial net to N.

The base case is $n = 0$, namely N is an initial net. Then item 1 holds by definition of initial net, and item 2 is trivially true since an initial net contains no memories and no $\mathbf{roll}(k)$.

For the inductive case ($n > 0$), let N' be a consistent net satisfying items 1 and 2, such that either $N' \mapsto N$ or $N' \rightsquigarrow N$. Then we proceed by case analysis on the reduction rule applied. If the last applied rule is *(Out)* (or *(Eval)*), two new distinct keys k' and k'' are created, thus the validity of item 1 is unaltered. Also a memory is created, with produced keys k'' and k'. Key k' is the key of the reduct process (the continuation of the process that performed the *(Out)* action), while k'' is the key of the new evaluated tuple. Moreover, both keys are bound in N. Finally, if a $\mathbf{roll}(k)$ is created, k is the key of the created memory, thus the completeness conditions for N are satisfied. In rules *(In)* and *(Read)* a new key k' and a new memory are created. The only produced key for the memory is k', which is bound and tags the reduct process. Again, the completeness conditions for N are satisfied. Rule *(New)* creates a new key k', a new node s', and a new memory. The only produced key for the memory is k', which is bound and tags the reduct process. Moreover, the node s' mentioned in the memory exists and it is bound in N, satisfying the completeness conditions for N.

If the last applied rule is a backward rule, then the result follows trivially from the fact that backward rules do not add any memories, keys or $\mathbf{roll}(k)$, but at most remove them. □

Lemma 2. *Let N and M be two* RμKLAIM *nets such that $N \mapsto M$. Then $\phi_N(N) \to \phi_N(M)$.*

Proof (SKETCH). Straightforward, by first proving by induction on the derivation of $N \equiv M$ that $N \equiv M \;\Rightarrow\; \phi_N(N) \equiv \phi_N(M)$, and then by observing that forward RμKLAIM rules are just decorated versions of μKLAIM rules, and decorations are removed by function ϕ_N. □

Lemma 3. *Let R and S be two μ*KLAIM *nets such that $R \to S$. Then for all consistent* RμKLAIM *nets M such that $\phi_N(M) = R$ there exists a* RμKLAIM *net N such that $M \mapsto N$ and $\phi_N(N) \equiv S$.*

Proof (SKETCH). Nets such that $\phi_N(M) = R$ have the same localities, processes and evaluated tuples as R. In addition, evaluated tuples and processes have keys, and M also contains memories and connectors. If an action is enabled in R then the corresponding action is enabled in M, possibly after some application of structural congruence rules *(Split)* and *(Ext)*. In most of the cases, the result of executing the action is a net N such that $\phi_N(N) = S$. However, $\phi_N(N)$ may differ from S because **nil** processes cannot be garbage collected in RμKLAIM, thus some more **nil** processes may occur in $\phi_N(N)$. In such cases, anyway, the extra **nil** processes can be removed by applying the μKLAIM structural congruence. □

Lemma 4 (Loop lemma)**.** *For all consistent* RμKLAIM *nets N and M, the following holds:*

$$N \mapsto M \iff M \rightsquigarrow N.$$

Proof (SKETCH). The proof is by induction on the derivation of $N \mapsto M$ for the *if* direction. The structural congruence rule *(Garb)* is needed to garbage-collect the unused keys.

The proof is by induction on the derivation of $M \rightsquigarrow N$ for the *only if* direction. One has to pay attention in rules *(InRev)* and *(ReadRev)* that the process Q is indeed the instantiation of the stored continuation P with the substitution σ resulting from the pattern matching. This always holds for consistent nets. □

Lemma 5 (Square lemma)**.** *If $\delta_1 = M \xrightarrow{\eta_1} N_1$ and $\delta_2 = M \xrightarrow{\eta_2} N_2$ are two coinitial concurrent transitions, then there exist two cofinal transitions $\delta_2/\delta_1 = N_1 \xrightarrow{\eta_2} N$ and $\delta_1/\delta_2 = N_2 \xrightarrow{\eta_1} N$.*

Proof. By case analysis on the form of transitions δ_1 and δ_2.

Both δ_1 and δ_2 forward: δ_1 and δ_2 can be any combination of forward reductions, namely we have 15 subcases (*(In)* and *(In)*, *(In)* and *(Out)*, *(In)* and *(Read)*, *(In)* and *(Eval)*, *(In)* and *(New)*, *(Out)* and *(Out)*, *(Out)* and *(Read)*, *(Out)* and *(Eval)*, *(Out)* and *(New)*, *(Read)* and *(Read)*, *(Read)* and *(Eval)*, *(Read)* and *(New)*, *(Eval)* and *(Eval)*, *(Eval)* and *(New)*, *(New)* and *(New)*). The most interesting case is the one of two **read**s on the same evaluated tuple.

$$\begin{aligned} M \equiv\ & s :: k : \mathbf{read}(T)@s'.P \\ & \parallel s' :: k' : \langle et \rangle \parallel s'' :: k'' : \mathbf{read}(T')@s'.P' \ \parallel M' \end{aligned}$$

Since M is well formed, by Lemma 1, k, k', k'' are pairwise distinct. Then $M \mapsto N_1$ with:

$$\begin{aligned} N_1 \equiv\ & (\nu k_1)\ (s :: k_1 : P\sigma \mid [k : \mathbf{read}(T)@s'.P; k'; k_1]) \\ & \parallel s' :: k' : \langle et \rangle \parallel \ s'' :: k'' : \mathbf{read}(T')@s'.P' \ \parallel M' \end{aligned}$$

where $\sigma = match([\![T]\!], et)$, and $M \mapsto N_2$ with:

$$\begin{aligned} N_2 \equiv\ & s :: k : \mathbf{read}(T)@s'.P \ \parallel \ s' :: k' : \langle et \rangle \\ & \parallel (\nu k_2)\ (s'' :: k_2 : P'\sigma' \mid [k'' : \mathbf{read}(T')@s'.P'; k'; k_2]) \parallel M' \end{aligned}$$

where $\sigma' = match([\![T']\!], et)$. Now, both N_1 and N_2 evolve to:

$$\begin{aligned} N \equiv\ & (\nu k_1)\ (s :: k_1 : P\sigma \mid [k : \mathbf{read}(T)@s'.P; k'; k_1]) \\ & \parallel s' :: k' : \langle et \rangle \\ & \parallel (\nu k_2)\ (s'' :: k_2 : P'\sigma' \mid [k'' : \mathbf{read}(T')@s'.P'; k'; k_2]) \parallel M' \end{aligned}$$

The other cases, since they are about actions on different evaluated tuples, are all similar. Let us consider as an example the case of a **read** and an **in** on different evaluated tuples.

$$\begin{aligned} M \equiv\ & s :: k : \mathbf{in}(T)@s'.P \ \parallel \ s' :: k' : \langle et \rangle \\ & \parallel s'' :: k'' : \mathbf{read}(T')@s'''.P' \\ & \parallel s''' :: k''' : \langle et' \rangle \ \parallel M' \end{aligned}$$

Since M is well formed, by Lemma 1, k, k', k'', k''' are pairwise distinct. Then $M \mapsto N_1$ with:

$$\begin{aligned} N_1 \equiv\ & (\nu k_1)(s :: k_1 : P\sigma \mid [k : \mathbf{in}(T)@s'.P; k' : \langle et \rangle; k_1]) \\ & \parallel s' :: \mathbf{empty} \\ & \parallel s'' :: k'' : \mathbf{read}(T')@s'''.P' \ \parallel \ s''' :: k''' : \langle et' \rangle \\ & \parallel M' \end{aligned}$$

where $\sigma = match([\![T]\!], et)$, and $M \mapsto N_2$ with:

$$\begin{array}{rl} N_2 \equiv & s :: k : \mathbf{in}(T)@s'.P \ \parallel \ s' :: k' : \langle et \rangle \\ & \parallel (\nu k_2)\ (s'' :: k_2 : P'\sigma' \mid [k'' : \mathbf{read}(T')@s'''.P'; k'''; k_2]) \\ & \parallel s''' :: k''' : \langle et' \rangle \ \parallel M' \end{array}$$

where $\sigma' = match([\![T']\!], et')$. Thus, both N_1 and N_2 evolve to:

$$\begin{array}{rl} N \equiv & (\nu k_1)(s :: k_1 : P\sigma \mid [k : \mathbf{in}(T)@s'.P; k' : \langle et \rangle; k_1]) \\ & \parallel s' :: \mathbf{empty} \\ & \parallel (\nu k_2)(s'' :: k_2 : P'\sigma' \mid [k'' : \mathbf{read}(T')@s'''.P'; k'''; k_2]) \\ & \parallel s''' :: k''' : \langle et' \rangle \ \parallel M' \end{array}$$

Both δ_1 and δ_2 backward: δ_1 and δ_2 can be any combination of backward reductions, namely we have 15 subcases (*(InRev)* and *(InRev)*, *(InRev)* and *(OutRev)*, *(InRev)* and *(ReadRev)*, *(InRev)* and *(EvalRev)*, *(InRev)* and *(NewRev)*, *(OutRev)* and *(OutRev)*, *(OutRev)* and *(ReadRev)*, *(OutRev)* and *(EvalRev)*, *(OutRev)* and *(NewRev)*, *(ReadRev)* and *(ReadRev)*, *(ReadRev)* and *(EvalRev)*, *(ReadRev)* and *(NewRev)*, *(EvalRev)* and *(EvalRev)*, *(EvalRev)* and *(NewRev)*, *(NewRev)* and *(NewRev)*).

The most interesting case is the one of two concurrent undos of two **read**s on the same evaluated tuple.

$$\begin{array}{rl} M \equiv & s :: k'' : Q \mid [k : \mathbf{read}(T)@s'.P; k'; k''] \\ & \parallel s' :: k' : \langle et \rangle \\ & \parallel \ s_1 :: k_1'' : Q_1 \mid [k_1 : \mathbf{read}(T)@s'.P_1; k'; k_1''] \\ & \parallel \ M' \end{array}$$

Since M is well formed, by Lemma 1, k, k', k'', k_1, k_1'' are pairwise distinct. Then $M \rightsquigarrow N_1$ with:

$$\begin{array}{rl} N_1 \equiv & s :: k : \mathbf{read}(T)@s'.P \ \parallel \ s' :: k' : \langle et \rangle \\ & \parallel \ s_1 :: k_1'' : Q_1 \mid [k_1 : \mathbf{read}(T)@s'.P_1; k'; k_1''] \\ & \parallel M' \end{array}$$

and $M \rightsquigarrow N_2$ with:

$$\begin{array}{rl} N_2 \equiv & s :: k'' : Q \mid [k : \mathbf{read}(T)@s'.P; k'; k''] \\ & \parallel s' :: k' : \langle et \rangle \parallel \ s_1 :: k_1 : \mathbf{read}(T)@s'.P_1 \parallel M' \end{array}$$

Thus, both N_1 and N_2 evolve to:

$$\begin{array}{rl} N \equiv & s :: k : \mathbf{read}(T)@s'.P \;\parallel\; s' :: k' : \langle et \rangle \\ & \parallel s_1 :: k_1 : \mathbf{read}(T)@s'.P_1 \parallel M' \end{array}$$

δ_1 **forward and** δ_2 **backward:** We have 25 subcases, due to the combination of any forward reduction with any backward reduction. The most interesting case is the one of a **read** on an evaluated tuple and an undo of a **read** on the same evaluated tuple.

$$\begin{array}{rl} M \equiv & s :: k : \mathbf{read}(T)@s'.P \;\parallel\; s' :: k' : \langle et \rangle \\ & \parallel (\nu k_2)\ (s'' :: k_2 : Q \mid [k'' : \mathbf{read}(T')@s'.P'; k'; k_2]) \\ & \parallel M' \end{array}$$

Since M is well formed, by Lemma 1, k, k', k'' and k_2 are pairwise distinct. Then $M \mapsto N_1$ with:

$$\begin{array}{rl} N_1 \equiv & (\nu k_1)(s :: k_1 : P\sigma \mid [k : \mathbf{read}(T)@s'.P; k'; k_1]) \\ & \parallel s' :: k' : \langle et \rangle \\ & \parallel (\nu k_2)\ (s'' :: k_2 : Q \mid [k'' : \mathbf{read}(T')@s'.P'; k'; k_2]) \parallel M' \end{array}$$

where $\sigma = match([\![T]\!], et)$, and $M \rightsquigarrow N_2$ with:

$$\begin{array}{rl} N_2 \equiv & s :: k : \mathbf{read}(T)@s'.P \;\parallel\; s' :: k' : \langle et \rangle \\ & \parallel\; s'' :: k'' : \mathbf{read}(T)@s'.P' \parallel M' \end{array}$$

Thus, both N_1 and N_2 evolve to:

$$\begin{array}{rl} N \equiv & (\nu k_1)\ (s :: k_1 : P\sigma \parallel s :: [k : \mathbf{read}(T)@s'.P; k'; k_1]) \\ & \parallel s' :: k' : \langle et \rangle \\ & \parallel\; s'' :: k'' : \mathbf{read}(T)@s'.P' \parallel M' \end{array}$$

□

In order to prove Theorem 1 we first have to prove two auxiliary lemmas.

Lemma 9 (Rearranging lemma)**.** *Let θ be a trace. There exist forward traces θ' and θ'' such that $\theta \asymp \theta'_\bullet; \theta''$.*

Proof. The proof is by lexicographic induction on the length of θ and on the distance between the beginning of θ and the earliest pair of transitions in θ of the form $\delta'; \delta_\bullet$ contradicting the property. If there is no such pair, then we are done. If there is one, we have two possibilities:
δ and δ' are concurrent: They can be swapped by Lemma 5, resulting in a later earliest contradicting pair, and by induction the result follows, since swapping transitions keeps the total length constant;
δ and δ' are in conflict: Let η_1 and η_2 be the forward/backward memories of δ and $\delta'_\bullet$, respectively. By Definition 7, δ and δ' are coinitial; let M be their source and N_1 and N_2 their targets, respectively. We have the following possibilities:

- $z \in \mathtt{clos}_{M\|N_1\|N_2}(\mathtt{cons}(\eta_1))$ and $z \in \mathtt{clos}_{M\|N_1\|N_2}(\mathtt{cons}(\eta_2))$: In this case $\delta = \delta'$ and then, by applying Lemma 4, we remove $\delta; \delta_\bullet$. Indeed, if they were different transitions sharing a key k, the only possibility, by Lemma 1 (completeness of well-formed nets), would be that they correspond to two actions on the same evaluated tuple. But this would mean having: i) a forward **in** followed by a backward **out**: this is not possible because first we have to undo the **in** in order to undo the **out**; ii) a forward **out** and a backward **in**: again this is not possible because this would mean that the forward **in** was done before the forward **out**. If they were different transitions sharing a locality l, this would mean that two **newloc**s create the same locality, but this is not possible since names of created localities are bound.

- $z \in \mathtt{read}(\eta_1)$ and $z \in \mathtt{clos}_{M\|N_1\|N_2}(\mathtt{cons}(\eta_2))$: If z is a key k then a forward **read** on an evaluated tuple is followed by a backward **out** on the same evaluated tuple: this is not possible because you have to undo the **read** before undoing the **out**. The case of a forward **read** on an evaluated tuple followed by a backward **in** is impossible as well, because this would mean that a forward **in** has happened before the forward **read**. If z is a locality then a forward action on a locality l is followed by the undo of the creation of the locality, but this is not possible since only **empty** localities can be removed.

- $z \in \mathtt{clos}_{M\|N_1\|N_1}(\mathtt{cons}(\eta_1))$ and $z \in \mathtt{read}(\eta_2)$: If z is a key k then a forward **out** on an evaluated tuple is followed by a backward **read** on the same evaluated tuple: this is not possible because this would mean a forward **read** has occurred before the **out**. The case of a forward

in on an evaluated tuple followed by a backward **read** is impossible as well, because you have to undo the **in** before undoing the **read**. If z is a locality then a forward **newloc** on a locality is followed by the undo of an action on that locality, but this is not possible since no action on this locality has been done yet.

□

Lemma 10 (Shortening lemma)**.** *Let θ_1 and θ_2 be coinitial and cofinal traces with θ_2 forward. Then, there exists a forward trace θ_1' of length at most that of θ_1 s.t. $\theta_1' \asymp \theta_1$.*

Proof. The proof is by induction on the length of θ_1. If θ_1 is forward we are done. Otherwise, by Lemma 9, we can write θ_1 as $\theta_\bullet; \theta'$, with θ and θ' forward. Let $\delta_\bullet; \delta'$ be the only two successive transitions in θ_1 with opposite directions with μ_1 backward memory of $\delta_\bullet$. Since μ_1 is removed by $\delta_\bullet$, then μ_1 has to be put back by another forward transition otherwise this difference will stay visible since θ_2 is a forward trace. Let δ_1 be the earliest such transition in θ_1. Since it is able to put back μ_1 it has to be the exact opposite of $\delta_\bullet$, so $\delta_1 = \delta$. Now we can swap δ_1 with all the transitions between δ_1 and $\delta_\bullet$, in order to obtain a trace in which δ_1 and $\delta_\bullet$ are adjacent. To do so we use Lemma 5, since all the transitions in between are concurrent. Assume, in fact, that there is a transition involving memory μ_2 which is not concurrent to δ_1. Thanks to Lemma 1 (completeness of well-formed nets) and since locality names are fresh, the only possible conflict may be between a $z \in \mathtt{read}(\mu_1)$ and a $z \in \mathtt{clos}_N(\mathtt{cons}(\mu_2))$, for an appropriate N, or vice versa. A k in $\mathtt{read}(\mu_1)$ means δ_1 is a forward **read** which has some conflicts with an **out** or an **in** occurring between the previous undo of the **read** and δ_1. Anyway, it is not possible to have an **out** of an evaluated tuple after (an undo of) a **read** ($\delta_\bullet$). It is also not possible to have an **in** before a **read** ($\delta_1 = \delta$). An l in $\mathtt{read}(\mu_1)$ means that there is the undo of an operation on a locality that has not been created yet. This is impossible since the name of the new locality is bound. In case of the opposite conflict $k \in \mathtt{clos}_{N'}(\mathtt{cons}(\mu_1))$, for an appropriate N', and a $k \in \mathtt{cons}(\mu_2)$ means δ_1 is the undo of an **in** (or the undo of an **out**) and δ_2 is a **read**. This is impossible because this would mean that a previous **in** has happened before the undo of **read**. The undo of an **out** combined with a **read** is impossible as well. Finally, if z is a locality this means that there is an operation targeting a locality which has been removed, but this is not possible since all the occurrences of the locality name have been removed. □

Theorem 1 (Causal consistency)**.** *Let θ_1 and θ_2 be coinitial traces, then $\theta_1 \asymp \theta_2$ if and only if θ_1 and θ_2 are cofinal.*

Proof. By Definition 8, if $\theta_1 \asymp \theta_2$ then θ_1 and θ_2 must be coinitial and cofinal, so this direction of the theorem is verified. Now we have to prove that θ_1 and θ_2 being coinitial and cofinal implies that $\theta_1 \asymp \theta_2$. By Lemma 9 we know that the two traces can be written as composition of a backward trace and a forward one. The proof is by lexicographic induction on the sum of the lengths of θ_1 and θ_2 and on the distance between the end of θ_1 and the earliest pair of transitions δ_1 in θ_1 and δ_2 in θ_2 which are not equal. If all the transitions are equal then we are done. Otherwise, we have to consider three cases depending on the direction of the two transitions.

- δ_1 **forward and** δ_2 **backward:** we have $\theta_1 = \theta_\bullet; \delta_1; \theta'$ and $\theta_2 = \theta_\bullet; \delta_2; \theta''$. Moreover we know that $\delta_1; \theta'$ is a forward trace, so we can apply Lemma 10 to the traces $\delta_1; \theta'$ and $\delta_2; \theta''$ (since θ_1 and θ_2 are coinitial and cofinal by hypothesis, also $\delta_1; \theta'$ and $\delta_2; \theta''$ are coinitial and cofinal) and we obtain that $\delta_2; \theta''$ has a shorter equivalent forward trace and so also θ_2 has a shorter equivalent forward trace. We can conclude by induction.

- δ_1 **and** δ_2 **forward:** by assumption, the two transitions are different. If they are not concurrent then they should conflict on a process $k : P$ that they both consume and store in different memories, or an evaluated tuple $k : \langle et \rangle$ one consumes and the other either reads or consumes, or on a locality l that one creates and the other uses. Since the two traces are cofinal there should be δ_2' in θ_2 creating the same memory as δ_1. However, no other process $k : P$ (nor evaluated tuple $k : \langle et \rangle$) is ever created in θ_2 thus this is not possible. The conflict cannot be on a locality either, since the locality name is bound. So we can assume that δ_1 and δ_2 are concurrent. Again let δ_2' be the transition in θ_2 creating the same memory of δ_1. We have to prove that δ_2' is concurrent to all the previous transitions. This holds since no previous transition can remove one of the processes needed for triggering δ_2' and since forward transitions can never conflict on k or l. Thus we can repetitively apply Lemma 5 to derive a trace equivalent to θ_2 where δ_2 and δ_2' are consecutive. We can apply a similar transformation to θ_1. Now we can apply Lemma 5 to δ_1 and δ_2 to have two traces of the same

length as before but where the first pair of different transitions is closer to the end. The thesis follows by inductive hypothesis.

– δ_1 **and** δ_2 **backward:** δ_1 and δ_2 cannot remove the same memory. Let μ_1 be the memory removed by δ_1. Since the two traces are cofinal, either there is another transition in θ_1 putting back the memory or there is a transition δ_1' in θ_2 removing the same memory. In the first case, δ_1 is concurrent to all the backward transitions following it, but the ones that consume processes generated by it. All the transitions of this kind have to be undone by corresponding forward transitions (since they are not possible in θ_2). Consider the last such transition: we can use Lemma 5 to make it the last backward transition. The forward transition undoing it should be concurrent to all the previous forward transitions (the reason is the same as in the previous case). Thus we can use Lemma 5 to make it the first forward transition. Finally we can apply the simplification rule $\delta_\bullet; \delta \asymp \epsilon_{\mathtt{target}(\delta)}$ (see Definition 8) to remove the two transitions, thus shortening the trace. The thesis follows by inductive hypothesis.

□

Lemma 6 (Backward confluence)**.** *Let M be a consistent net. If $M \rightsquigarrow^* M_1$ and $M \rightsquigarrow^* M_2$ then there exists M' such that $M_1 \rightsquigarrow^* M'$ and $M_2 \rightsquigarrow^* M'$.*

Proof. Thanks to the Loop Lemma (Lemma 4) from $M \rightsquigarrow^* M_2$ we have $M_2 \mapsto^* M$. Thanks to Lemma 9 since $M_2 \mapsto^* M \rightsquigarrow^* M_1$ there exists M' such that $M_2 \rightsquigarrow^* M' \mapsto M_1$. Using again the Loop Lemma (Lemma 4) $M_1 \rightsquigarrow M'$ as desired. □