Causal-Consistent Reversibility in a Tuple-Based Language

Elena Giachino
and Ivan Lanese
Focus Team, University of Bologna/INRIA, Italy
Email: elena.giachino@unibo.it, ivan.lanese@gmail.com

Claudio Antares Mezzina
SOA Unit
FBK Trento, Italy
Email: mezzina@fbk.eu

Francesco Tiezzi
School of Science and Technology
University of Camerino, Italy
Email: francesco.tiezzi@unicam.it

***Abstract*—*Causal-consistent reversibility* is a natural way of undoing concurrent computations. We study causal-consistent reversibility in the context of μKLAIM, a formal coordination language based on distributed tuple spaces. We consider both *uncontrolled reversibility*, suitable to study the basic properties of the reversibility mechanism, and *controlled reversibility* based on a rollback operator, more suitable for programming applications. The causality structure of the language, and thus the definition of its reversible semantics, differs from all the reversible languages in the literature because of its generative communication paradigm. In particular, the reversible behavior of μKLAIM read primitive, reading a tuple without consuming it, cannot be matched using channel-based communication. We illustrate the reversible extensions of μKLAIM on a simple, but realistic, application scenario.**

I. Introduction

Reversibility is a main ingredient of different kinds of systems, including, e.g., biological systems or quantum systems. We are mainly interested in reversibility as a support for programming reliable concurrent systems. The basic idea is that if a system reaches an undesired state (e.g., an error or deadlock state), reversibility can be used to go back to a past desirable state. Our claim is that the ability to reverse actions is key to understanding and improving existing patterns for programming reliable systems, such as transactions or checkpointing, and to devise new ones.

Studying reversibility in a concurrent setting is particularly tricky. In fact, even the definition of reversibility is different w.r.t. the sequential one, since "recursively undo the last action" is not meaningful in a concurrent scenario, where many actions can be executed at the same time by different threads. This observation led to the concept of *causal-consistent reversibility*: one may undo any action if no other action depending on it has been executed (and not undone). Building on this definition, reversible extensions of many concurrent calculi and languages have been defined, e.g., for CCS [5], [18], π-calculus [4], higher-order π [14] and μOz [17]. However, to figure out how to make a general programming language reversible, the interplay between reversibility and many common language features has still to be understood. In particular, none of the reversible calculi in the literature features tuple-based communication: they all consider channel-based communication.

This paper studies reversibility in the context of μKLAIM [11], a formal language based on distributed tuple spaces derived from the coordination language KLAIM [8]. μKLAIM contrasts on two main points with all the languages whose causal-consistent reversible semantics has been studied in the literature. First, it features localities. Second, it uses tuple-based communication as the interaction paradigm, supported by five primitives. Primitives **out** and **in** respectively insert tuples into and remove them from tuple spaces. Primitives **eval**, to execute a process on a possibly remote location, and **newloc**, creating a new location, support distribution. Finally, μKLAIM features the primitive **read**, which reads a tuple without consuming it. This last primitive allows concurrent processes to access a shared resource while staying independent, thus undoing the actions of one of them has no impact on the others. This behavior, common when manipulating shared data structures, e.g. in software transactional memories, cannot be programmed using only **in** and **out** primitives, nor using channel-based communications, since the resulting causal structure would be different.

In this paper, we first study *uncontrolled reversibility* (Section III), i.e. we define how a process executes forward or backward, but not when it is supposed to do so. This produces a clean algebraic setting, suitable to prove properties of our reversibility mechanism. In particular, we show that reversible μKLAIM (RμKLAIM for short) is causally consistent (Theorem 1), and that its forward computations correspond to μKLAIM computations (Lemmas 2 and 3). However, uncontrolled reversibility is not suitable for programming error recovery activities. In fact, it does not provide a mechanism to trigger a backward computation in case of error: backward actions are always enabled. Even more, a RμKLAIM process may always diverge by doing and undoing the same action forever.

To solve this problem, we build on top of RμKLAIM a language with *controlled reversibility*, CRμKLAIM (Section IV). CRμKLAIM computation normally proceeds forward, but the programmer may ask for a rollback using a dedicated **roll** operator. This operator undoes a given past action, and all its consequences, but it does not affect independent actions. The **roll** operator is based on the uncontrolled reversibility mechanism, but it is much more suitable to exploit reversibility for programming actual applications. We put CRμKLAIM at work on a practical example about franchising (Section V). Proofs and additional examples are available in the companion technical report [10].

From the practical perspective, we believe that the formal approach proposed in this paper is a further step towards the sound development of a real-world reversible language for

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

TABLE I. μKLAIM SYNTAX

(Monoid)	$N \parallel \mathbf{0} \equiv N \quad N_1 \parallel N_2 \equiv N_2 \parallel N_1$ $(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$
(RCom)	$(\nu l_1)(\nu l_2)N \equiv (\nu l_2)(\nu l_1)N$
(PDef)	$l :: A \equiv l :: P$ if $A \triangleq P$
(Ext)	$N_1 \parallel (\nu l)N_2 \equiv (\nu l)(N_1 \parallel N_2)$ if $l \notin fn(N_1)$
(Alpha)	$N \equiv N'$ if $N =_\alpha N'$
(Abs)	$l :: C \equiv l :: (C \mid \mathbf{nil})$
(Clone)	$l :: C_1 \mid C_2 \equiv l :: C_1 \parallel l :: C_2$

TABLE II. μKLAIM STRUCTURAL CONGRUENCE

programming distributed systems. Its main benefit with respect to traditional languages would be to relieve the programmer from coding rollback activities from scratch: they can be easily obtained by applying the rollback operator.

II. 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. The data retrieving mechanism is based on associative pattern-matching. In this paper, to simplify the presentation, we consider a core language of KLAIM, called μKLAIM. We refer to [2] for a detailed account of KLAIM and μKLAIM.

Syntax. The syntax of μKLAIM is in Table I. We assume four disjoint sets: the set of *localities*, ranged over by l, of *locality variables*, ranged over by u, of *value variables*, ranged over by x, and of *process identifiers*, ranged over by A. Localities are the addresses (i.e., network references) of nodes and are the syntactic ingredient used to model administrative domains. In μKLAIM, communicable objects are (evaluated) *tuples*, i.e., sequences of actual fields. Tuple fields may contain expressions, localities or locality variables. We leave the *expressions* e unspecified; we just assume they include *values* (v) and value variables. *Names*, i.e., locality variables and localities, are ranged over by ℓ. We assume each process identifier A has a single definition $A \triangleq P$, available at any locality of the net.

Nets are finite plain collections of nodes where *components*, i.e., processes and evaluated tuples, can be hosted. A *node* is a pair $l :: C$, where l is the address of the node and C is the hosted component. In the net $(\nu l)N$, the scope of the name l is restricted to N. $\mathbf{0}$ denotes the empty net.

Processes, the μKLAIM active computational units, may be inactive as $\mathbf{nil}$, prefixed by an action as $a.P$, parallel compositions as $P_1 \mid P_2$, and process identifiers as A. We may drop trailing $\mathbf{nil}$s. Processes may be executed concurrently either at the same locality or at different localities and can perform actions.

Actions $\mathbf{out}$, $\mathbf{in}$ and $\mathbf{read}$ add/withdraw/access data from repositories. Action $\mathbf{eval}$ activates a new thread of execution in a (possibly remote) node, and $\mathbf{newloc}$ creates new nodes. All actions but $\mathbf{newloc}$ indicate their target locality. Actions $\mathbf{in}$ and $\mathbf{read}$ are blocking and use, as patterns to select data in repositories, *templates*: 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.

Localities and variables can be *bound* inside processes and nets: $\mathbf{newloc}(l).P$ binds name l in P, and $(\nu l)N$ binds l in N. Prefixes $\mathbf{in}(\ldots, !\_, \ldots)@\ell.P$ and $\mathbf{read}(\ldots, !\_, \ldots)@\ell.P$ bind variable _ in P. A locality/variable that is not bound is called *free*. The set $fn(\cdot)$ of free names of a term 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 $\equiv$ is defined as the least congruence closed under the equational laws in Table II. Law *(Abs)* states that $\mathbf{nil}$ 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 $\cdot \mid \cdot$). The other laws are standard.

We define the auxiliary *pattern-matching* function $match(\cdot, \cdot)$ as the smallest function closed under the rules in Table III. Intuitively, a tuple matches a template if they have the same number of fields, and corresponding fields match: two values/localities match only if they are identical, bound value/locality variables match any value/locality, and the matching for free variables always fails. When $match(T, t)$ succeeds, it returns a substitution for the variables in T; otherwise, it is undefined. A *substitution* σ is a function with finite domain from variables to localities/values, and is written as a collection of pairs of the form v/x or l/u. We use $\circ$ to denote substitution composition and ϵ to denote the empty substitution.

We assume the existence of a function $[\![\cdot]\!]$ for evaluating tuples and templates, which computes the value of closed expressions occurring in a tuple/template. Its definition depends on the definition of expressions, which we left unspecified. Only *evaluated tuples* $\langle et \rangle$ are stored in tuple spaces.

We say a relation $\mathcal{R}$ is *evaluation closed* if it is closed under active contexts, i.e. $N_1 \; \mathcal{R} \; N_1'$ implies $(N_1 \parallel N_2) \; \mathcal{R} \; (N_1' \parallel$

$$\frac{match(T_1, t_1) = \sigma_1 \qquad match(T_2, t_2) = \sigma_2}{match((T_1, T_2), (t_1, t_2)) = \sigma_1 \circ \sigma_2}$$

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

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

TABLE III. μKLAIM MATCHING RULES

$$\frac{[\![t]\!] = et}{l :: \mathbf{out}(t)@l'.P \ \| \ l' :: \mathbf{nil} \ \mapsto \ l :: P \ \| \ l' :: \langle et \rangle} \ \textit{(Out)}$$

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

$$\frac{match([\![T]\!], et) = \sigma}{l :: \mathbf{read}(T)@l'.P \ \| \ l' :: \langle et \rangle \ \mapsto \ l :: P\sigma \ \| \ l' :: \langle et \rangle} \ \textit{(Read)}$$

$$l :: \mathbf{newloc}(l').P \ \mapsto \ (\nu l')(l :: P \ \| \ l' :: \mathbf{nil}) \ \textit{(New)}$$

$$l :: \mathbf{eval}(Q)@l'.P \ \| \ l' :: \mathbf{nil} \ \mapsto \ l :: P \ \| \ l' :: Q \ \textit{(Eval)}$$

TABLE IV. μKLAIM OPERATIONAL SEMANTICS

N_2) and $(\nu l)N_1 \ \mathcal{R} \ (\nu l)N_1'$, and under structural congruence, i.e. $N \equiv M \ \mathcal{R} \ M' \equiv N'$ implies $N \ \mathcal{R} \ N'$. The μKlaim reduction relation $\mapsto$ is the smallest evaluation-closed relation satisfying the rules in Table IV. All rules for (possibly remote) actions $\mathbf{out}$, $\mathbf{eval}$, $\mathbf{in}$ and $\mathbf{read}$ require the existence of the target node l'. In rule *(Out)* the action can proceed only if the tuple in its argument is evaluable. If so, the evaluated tuple is released in the target node l'. Rules *(In)* and *(Read)* require the target node to contain a tuple matching their (necessarily evaluable) template argument T. The content of the matched tuple is then used to replace the free occurrences of the variables bound by T in P, the continuation of the process performing the actions. Action $\mathbf{in}$ consumes the matched tuple, while action $\mathbf{read}$ does not. Rule *(New)* creates a new (private) node. Rule *(Eval)* launches a new thread executing process Q on a target node l'.

III. Uncontrolled reversibility

In this section we define RμKlaim, an extension of μKlaim with uncontrolled reversibility. RμKlaim nets allow both *forward* actions, modeling μKlaim actions, and *backward* actions, undoing them, but nothing is specified about whether to prefer forward steps over backward steps, or vice versa. While the general approach follows [14], the technical development is considerably different.

We first present RμKlaim, with a simple example to point out the peculiarity of the causality relationships produced by RμKlaim constructs. Then, we show that the typical properties expected from a reversible formalism hold.

Syntax. RμKlaim syntax is in Table V. Processes, actions, (evaluated) tuples, and templates are the same as in μKlaim (Table I). The main ingredients of RμKlaim are *keys* k, uniquely identifying tuples and processes, *memories* μ, storing information for undoing past actions, and *connectors* $k_1 \prec (k_2, k_3)$, tracking causality information. More precisely, we have the additional syntactic category of keys, ranged over by $k, h, \ldots$. We use z to range over keys and localities. Uniqueness of keys is enforced by using restriction, the only binder for keys (free and bound keys and α-conversion are defined as usual, and from now on $fn(N)$ also includes free keys), and by only considering *well-formed* nets.

$$N ::= \ \mathbf{0} \ | \ l :: C \ | \ l :: \mathbf{empty} \ | \ N_1 \ \| \ N_2 \ | \ (\nu z)N$$

$$C ::= \ k : \langle et \rangle \ | \ k : P \ | \ C_1 \,|\, C_2 \ | \ \mu \ | \ k_1 \prec (k_2, k_3)$$

$$\begin{array}{rl} \mu ::= & [k : \mathbf{out}(t)@l; k''; k'] \ | \ [k : \mathbf{in}(T)@l.P; h : \langle et \rangle; k'] \\ | & [k : \mathbf{read}(T)@l.P; h; k'] \ | \ [k : \mathbf{newloc}(l); k'] \\ | & [k : \mathbf{eval}(Q)@l; k''; k'] \end{array}$$

TABLE V. RμKLAIM SYNTAX

Definition 1 (Initial and well-formed nets): A RμKlaim net is *initial* if it has no memories, no connectors, and all its keys are distinct. A RμKlaim net is *well formed* if it can be obtained by forward or backward reductions (cfr. Definition 2) starting from an initial net.

Keys are needed to distinguish processes/tuples with the same form but different histories, thus allowing for different backward actions. Histories are stored in memories and connectors. A memory keeps track of a past action, thus we have one kind of memoryfor each kind of action. All of them store the prefix giving rise to the action and the fresh key k' generated for the continuation. Furthermore, memories for $\mathbf{in}$ and $\mathbf{read}$ store their original continuation P, since it cannot be recovered from the running one, obtained by applying a substitution - a non reversible transformation[1]. Also, the $\mathbf{out}$ memory stores the key k'' of the created tuple, the $\mathbf{eval}$'s one the key k'' of the spawned process, and the $\mathbf{in}$'s one the consumed tuple $h : \langle et \rangle$. The memory for $\mathbf{read}$ only needs the key h of the read tuple, still available in the term and uniquely identified by key h. Connector $k_1 \prec (k_2, k_3)$ recalls that processes with keys k_2 and k_3 originated from the split of process tagged by k_1. Finally, we distinguish empty localities, $l :: \mathbf{empty}$, containing no information, from localities $l :: k : \mathbf{nil}$ containing a $\mathbf{nil}$ process with its key k, which may interact with a memory to perform a backward action.

Operational semantics. Structural congruence for RμKlaim extends the one for μKlaim in Table II to deal with keys: new rules *(Garb)* and *(Split)* and updated rules are reported in Table VI. Rule *(Garb)* garbage-collects unused keys. Rule *(Split)* splits parallel processes using a connector and generating fresh keys to preserve keys uniqueness.

Definition 2 (RμKlaim semantics): The operational semantics of RμKlaim consists of a *forward reduction relation* $\mapsto_r$ and a *backward reduction relation* $\rightsquigarrow_r$. They are the smallest evaluation-closed relations (now closure under active contexts considers also restriction on keys) satisfying the rules in Table VII.

Forward rules correspond to μKlaim rules, adding the management of keys and memories. We have one backward rule for each forward rule, undoing the forward action. Consider rule *(Out)*. Existence of the target node l' is guaranteed by

[1] One may look for more compact ways to store history information. This issue is considered for reversible μOz in [17], but it is out of the scope of the present paper.

$$\textit{(RCom)}\ (\nu z_1)\,(\nu z_2)\,N \equiv (\nu z_2)\,(\nu z_1)\,N \qquad \textit{(PDef)}\ l :: k : A \equiv l :: k : P \quad \text{if } A \triangleq P \qquad \textit{(Ext)}\ N_1 \parallel (\nu z)\,N_2 \equiv (\nu z)\,(N_1 \parallel N_2) \ \text{ if } z \notin fn(N_1)$$

$$\textit{(Abs)}\ l :: C \equiv l :: C \parallel l :: \mathbf{empty} \qquad \textit{(Garb)}\ (\nu k)\,\mathbf{0} \equiv \mathbf{0} \qquad \textit{(Split)}\ l :: k : P \mid Q \equiv (\nu k_1, k_2)\, l :: k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

TABLE VI. RμKLAIM STRUCTURAL CONGRUENCE

$$\frac{[\![t]\!] = et}{\begin{array}{l} l :: k : \mathbf{out}(t)@l'.P \ \parallel \ l' :: \mathbf{empty} \\ \mapsto_r \ (\nu k', k'')\,(\, l :: k' : P \mid [k : \mathbf{out}(t)@l'; k''; k'] \ \parallel \ l' :: k'' : \langle et \rangle \,) \end{array}}\ \textit{(Out)}$$

$$\begin{array}{l} (\nu k'')\,(\, l :: k' : P \mid [k : \mathbf{out}(t)@l'; k''; k'] \ \parallel \ l' :: k'' : \langle et \rangle \,) \\ \rightsquigarrow_r \ l :: k : \mathbf{out}(t)@l'.P \ \parallel \ l' :: \mathbf{empty} \end{array}\ \textit{(OutRev)}$$

$$\frac{match([\![T]\!], et) = \sigma}{\begin{array}{l} l :: k : \mathbf{in}(T)@l'.P \ \parallel \ l' :: h : \langle et \rangle \\ \mapsto_r \ (\nu k')\, l :: k' : P\sigma \mid [k : \mathbf{in}(T)@l'.P; h : \langle et \rangle; k'] \ \parallel \ l' :: \mathbf{empty} \end{array}}\ \textit{(In)}$$

$$\begin{array}{l} l :: k' : Q \mid [k : \mathbf{in}(T)@l'.P; h : \langle et \rangle; k'] \ \parallel \ l' :: \mathbf{empty} \\ \rightsquigarrow_r \ l :: k : \mathbf{in}(T)@l'.P \ \parallel \ l' :: h : \langle et \rangle \end{array}\ \textit{(InRev)}$$

$$\frac{match([\![T]\!], et) = \sigma}{\begin{array}{l} l :: k : \mathbf{read}(T)@l'.P \ \parallel \ l' :: h : \langle et \rangle \\ \mapsto_r \ (\nu k')\, l :: k' : P\sigma \mid [k : \mathbf{read}(T)@l'.P; h; k'] \ \parallel \ l' :: h : \langle et \rangle \end{array}}\ \textit{(Read)}$$

$$\begin{array}{l} l :: k' : Q \mid [k : \mathbf{read}(T)@l'.P; h; k'] \ \parallel \ l' :: h : \langle et \rangle \\ \rightsquigarrow_r \ l :: k : \mathbf{read}(T)@l'.P \ \parallel \ l' :: h : \langle et \rangle \end{array}\ \textit{(ReadRev)}$$

$$\begin{array}{l} l :: k : \mathbf{newloc}(l').P \\ \mapsto_r \ (\nu l')\,(\,(\nu k')\, l :: k' : P \mid [k : \mathbf{newloc}(l'); k'] \ \parallel \ l' :: \mathbf{empty}\,) \end{array}\ \textit{(New)}$$

$$\begin{array}{l} (\nu l')\,(\, l :: k' : P \mid [k : \mathbf{newloc}(l'); k'] \ \parallel \ l' :: \mathbf{empty}\,) \\ \rightsquigarrow_r \ l :: k : \mathbf{newloc}(l').P \end{array}\ \textit{(NewRev)}$$

$$\begin{array}{l} l :: k : \mathbf{eval}(Q)@l'.P \ \parallel \ l' :: \mathbf{empty} \\ \mapsto_r \ (\nu k', k'')\,(\, l :: k' : P \mid [k : \mathbf{eval}(Q)@l'; k''; k'] \ \parallel \ l' :: k'' : Q\,) \end{array}\ \textit{(Eval)}$$

$$\begin{array}{l} l :: k' : P \mid [k : \mathbf{eval}(Q)@l'; k''; k'] \ \parallel \ l' :: k'' : Q \\ \rightsquigarrow_r \ l :: k : \mathbf{eval}(Q)@l'.P \ \parallel \ l' :: \mathbf{empty} \end{array}\ \textit{(EvalRev)}$$

TABLE VII. RμKLAIM OPERATIONAL SEMANTICS

requiring a parallel term $l' :: \mathbf{empty}$. If locality l' 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 tuple $\langle et \rangle$, respectively. 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 the same locality and a tuple with key k'' in the target locality l'. Requiring that l' contains only the 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 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 tuple. This ensures that **read** actions that have accessed the tuple, whose resulting memory would contain k'', have been undone. The problem of **read** dependencies is peculiar to the μKLAIM setting, and it does not emerge in the other works in the reversibility literature. On the other hand, restricting key k' in rule *(OutRev)* would be redundant since in a well-formed 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. Executing the backward rule *(OutRev)* undoes the effect of the forward rule *(Out)*, as proved by the Loop lemma below. 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 consumed tuple is stored in the memory, while in rule *(Read)* only the key is needed since the 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 l' 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)*.

Example 1: We show an example to clarify the difference between the behavior of a RμKLAIM **read** action and its implementations in the other reversible languages we are aware of. They feature channel-based communication, thus the only way of accessing a resource is consuming it with an input and restoring it with an output. This corresponds to the behavior we obtain in RμKLAIM by using an **in** followed by an **out**. Consider a RμKLAIM net N with three nodes, l_1 hosting a tuple $\langle \mathit{foo} \rangle$, and l_2 and l_3 hosting processes willing to access such tuple:

$$\begin{array}{rcl} N' & = & l_1 :: k_1 : \langle \mathit{foo} \rangle \parallel \ l_2 :: k_2 : \mathbf{in}(\mathit{foo})@l_1.\mathbf{out}(\mathit{foo})@l_1.P \\ & & \qquad\qquad\quad\ \parallel \ l_3 :: k_3 : \mathbf{in}(\mathit{foo})@l_1.\mathbf{out}(\mathit{foo})@l_1.P' \end{array}$$

By executing first the sequence of **in** and **out** in l_2, and then the corresponding sequence in l_3 (the order is relevant), the net evolves to:

$$\begin{array}{l} (\nu k'_2, k''_2, k'''_2, k'_3, k''_3, k'''_3)(\, l_1 :: k'''_3 : \langle \mathit{foo} \rangle \\ \ \parallel l_2 :: k''_2 : P \mid [k_2 : \mathbf{in}(\mathit{foo})@l_1.\mathbf{out}(\mathit{foo})@l_1.P; k_1 : \langle \mathit{foo} \rangle; k'_2] \\ \qquad\qquad\qquad \mid [k'_2 : \mathbf{out}(\mathit{foo})@l_1; k'''_2; k''_2] \\ \ \parallel l_3 :: k''_3 : P' \mid [k_3 : \mathbf{in}(\mathit{foo})@l_1.\mathbf{out}(\mathit{foo})@l_1.P'; k'''_2 : \langle \mathit{foo} \rangle; k'_3] \\ \qquad\qquad\qquad \mid [k'_3 : \mathbf{out}(\mathit{foo})@l_1; k'''_3; k''_3]\,) \end{array}$$

Now, the process in l_2 cannot immediately perform a backward step, since it needs the tuple $k'''_2 : \langle \mathit{foo} \rangle$ in l_1, while only $k'''_3 : \langle \mathit{foo} \rangle$ is available. The former tuple has been consumed by the **in** action at l_3 (see the corresponding memory stored in l_3) and then replaced by the latter by the **out** action at l_3. This means that to perform the backward step of the process in l_2 one needs first to reverse the process in l_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 $\rho\pi$ [14] or an output process in [5], [18], [4].

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**:

$$\begin{array}{rcl} N & = & l_1 :: k_1 : \langle \mathit{foo} \rangle \ \parallel \ l_2 :: k_2 : \mathbf{read}(\mathit{foo})@l_1.P \\ & & \qquad\qquad\quad\ \ \parallel \ l_3 :: k_3 : \mathbf{read}(\mathit{foo})@l_1.P' \end{array}$$

$\mathtt{erN}(\mathbf{0}) = \mathbf{0}$	$\mathtt{erN}(N_1 \parallel N_2) = \mathtt{erN}(N_1) \parallel \mathtt{erN}(N_2)$
$\mathtt{erN}(l :: \mathbf{empty}) = l :: \mathbf{nil}$	$\mathtt{erN}(l :: C) = l :: \mathtt{erC}(C)$
$\mathtt{erN}((\nu k)\ N) = \mathtt{erN}(N)$	$\mathtt{erN}((\nu l)\ N) = (\nu l)\ \mathtt{erN}(N)$
$\mathtt{erC}(k : P) = P$	$\mathtt{erC}(C_1 \mid C_2) = \mathtt{erC}(C_1) \mid \mathtt{erC}(C_2)$
$\mathtt{erC}(k : \langle et \rangle) = \langle et \rangle$	$\mathtt{erC}(k \prec (k_1, k_2)) = \mathbf{nil} \qquad \mathtt{erC}(\mu) = \mathbf{nil}$

TABLE VIII. erN AND erC FUNCTIONS

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

$$\begin{array}{l}(\nu k'_2, k'_3)\ (\,l_1 :: k_1 : \langle \mathit{foo} \rangle \\ \ \parallel\ l_2 :: k'_2 : P\ \mid [k_2 : \mathbf{read}(\mathit{foo})@l_1.P; k_1; k'_2] \\ \ \parallel\ l_3 :: k'_3 : P' \mid [k_3 : \mathbf{read}(\mathit{foo})@l_1.P'; k_1; k'_3]\,)\end{array}$$

Any of the two processes, say l_2, can undo the executed **read** action without affecting the execution of the other one. Thus, applying rule *(ReadRev)* we get:

$$\begin{array}{l}(\nu k'_2, k'_3)\ (\,l_1 :: k_1 : \langle \mathit{foo} \rangle \\ \ \parallel\ l_2 :: k_2 : \mathbf{read}(\mathit{foo})@l_1.P \\ \ \parallel\ l_3 :: k'_3 : P' \mid [k_3 : \mathbf{read}(\mathit{foo})@l_1.P'; k_1; k'_3]\,)\end{array}$$

Basic properties. We now show that RμKLAIM respects the μKLAIM semantics, and that it is causally consistent. We first introduce some auxiliary definitions.

Well-formed nets satisfy the property below, where, given memories of the shape $[k : \mathbf{out}(t)@l; k''; k']$, $[k : \mathbf{in}(T)@l.P; h' : \langle t \rangle; k']$, $[k : \mathbf{read}(T)@l.P; h; k']$, $[k : \mathbf{eval}(Q)@l; k''; k']$ and $[k : \mathbf{newloc}(l); k']$, and connectors of the shape $k \prec (k', k'')$, we call k the *head* of those memories/connector, and k' and, when they occur, k'' or h, the *tail*. Keys h and h' occur in an *input position*.

Definition 3 (Complete net): A net N is complete, written $\mathtt{complete}(N)$, if: (i) for each key k in the tail of a memory/connector of N there exists in N (possibly inside a memory) either a process $k : P$, or a tuple $k : \langle t \rangle$, or a connector $k \prec (h_1, h_2)$ and, unless all the occurrences of k are in input positions, k is bound in N; and (ii) for each memory $[k : \mathbf{newloc}(l); k']$ in N there exists in N a node named l and l is bound in N.

Lemma 1: For each well-formed net N: (i) all keys occurring in N attached to processes or tuples (possibly in a memory) are distinct, and (ii) N is complete.

From a RμKLAIM net we can derive a μKLAIM net by removing history and causality information. This is formalized by function erN (and the auxiliary function erC for components) defined in Table VIII. 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 $N \mapsto_r M$. Then $\mathtt{erN}(N) \mapsto \mathtt{erN}(M)$.

Lemma 3: Let R and S be two μKLAIM nets such that $R \mapsto S$. Then for all RμKLAIM nets M such that $\mathtt{erN}(M) = R$ there exists a RμKLAIM net N such that $M \mapsto_r N$ and $\mathtt{erN}(N) \equiv S$.

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

Lemma 4 (Loop lemma): For all well-formed RμKLAIM nets N and M, the following holds: $N \mapsto_r M \iff M \rightsquigarrow_r N$.

We now move to the proof that RμKLAIM is indeed causally consistent.

In a forward reduction $N \mapsto_r M$ we call *forward memory* the memory μ which does not occur in N and occurs in M. Similarly, in a backward reduction $N \rightsquigarrow_r M$ we call *backward memory* the memory μ which occurs in N and does not occur in M. We call *transition* a triplet of the form $N \xrightarrow{\mu_{\mapsto_r}} M$, or $N \xrightarrow{\mu_{\rightsquigarrow_r}} M$, where N, M are well-formed 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 $\mu_{\mapsto_r}$ and $\mu_{\rightsquigarrow_r}$. If $\eta = \mu_{\mapsto_r}$, then $\eta_\bullet = \mu_{\rightsquigarrow_r}$, 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 $h \prec (k_1, k_2)$, there is a fixed function determining k_1 and k_2 from h, and that different values of h 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 $\mathit{closure}$ 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 pairwise composable transitions 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. The *stamp* $\lambda(\mu_{\mapsto_r})$ of a memory μ is:

$$\begin{array}{lcl}\lambda([k : \mathbf{out}(t)@l; k''; k']) & = & \{k, k', k'', \mathtt{r}(l)\} \\ \lambda([k : \mathbf{in}(T)@l.P; k'' : \langle et \rangle; k']) & = & \{k, k', k'', \mathtt{r}(l)\} \\ \lambda([k : \mathbf{read}(T)@l.P; k''; k']) & = & \{k, \mathtt{r}(k''), k', \mathtt{r}(l)\} \\ \lambda([k : \mathbf{eval}(Q)@l; k''; k']) & = & \{k, k', k'', \mathtt{r}(l)\} \\ \lambda([k : \mathbf{newloc}(l); k']) & = & \{k, k', l\}\end{array}$$

We set $\lambda(\mu_{\rightsquigarrow_r}) = \lambda(\mu_{\mapsto_r})$. The stamp of a memory defines the resources used by the corresponding transitions. The notation $\mathtt{r}(z)$ highlights that resource z is used in read-only modality. All actions but **newloc** use a locality name in a read-only modality. We use κ to range over tags $\mathtt{r}(z)$ and z. We define the *closure* w.r.t. a net N of a tag κ as $\mathit{closure}_N(\kappa) = \{\kappa\} \cup \mathit{closure}_N(h)$ if $\kappa = k_1$ or $\kappa = k_2$ and $h \prec (k_1, k_2)$ occurs in N, $\{\kappa\}$ otherwise. We define the closure over a set K of tags as $\mathit{closure}_N(K) = \bigcup_{\kappa \in K} \mathit{closure}_N(\kappa)$. The closure captures that a connector $h \prec (k_1, k_2)$ means that resources k_1 and k_2 are part of resource h.

Definition 4 (Concurrent transitions): Two coinitial transitions $M \xrightarrow{\eta_1} N_1$ and $M \xrightarrow{\eta_2} N_2$ are *in conflict* if, for some resource z, one of the following holds: (i) $z \in \mathit{closure}_{M \parallel N_1}(\lambda(\eta_1))$ and $z \in \mathit{closure}_{M \parallel N_2}(\lambda(\eta_2))$, (ii) $\mathtt{r}(z) \in \lambda(\eta_1)$ and $z \in \mathit{closure}_{M \parallel N_2}(\lambda(\eta_2))$, or (iii) $z \in \mathit{closure}_{M \parallel N_1}(\lambda(\eta_1))$ and $\mathtt{r}(z) \in \lambda(\eta_2)$. Two coinitial transitions are *concurrent* if they are not in conflict.

Essentially, two transitions are in conflict if they both use the same resource, and at most one of them uses it in read-only modality.

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 $\delta_1/\delta_2 = N_2 \xrightarrow{\eta_1} N$.

Causal equivalence, denoted by $\asymp$, is the least equivalence relation between traces closed under composition that obeys the following rules:

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

Intuitively causal equivalence identifies traces that differ only for swaps of concurrent actions and simplifications of inverse actions. Next result shows that there is a unique way to go from one state to another up to causal equivalence. This means that causal equivalent traces can be reversed in the same ways, and 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.

IV. CONTROLLED REVERSIBILITY

In this section we define CRμKLAIM, an extension of μKLAIM featuring an explicit rollback facility to control RμKLAIM reversing capabilities, allowing us to program recovery activities inside μKLAIM applications. We follow the general approach of [13], but adapted to deal with the interplay of the different μKLAIM actions.

$$\begin{array}{rcl}
P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \quad | \quad \mathbf{roll}(\iota) \\[4pt]
a & ::= & \mathbf{out}_\gamma(t)@\ell \quad | \quad \mathbf{eval}_\gamma(P)@\ell \quad | \quad \mathbf{in}_\gamma(T)@\ell \\
 & | & \mathbf{read}_\gamma(T)@\ell \quad | \quad \mathbf{newloc}_\gamma(l) \\[4pt]
\mu & ::= & [k : \mathbf{out}_\gamma(t)@l.P; k''; k'] \quad | \quad [k : \mathbf{in}_\gamma(T)@l.P; h : \langle t \rangle; k'] \\
 & | & [k : \mathbf{read}_\gamma(T)@l.P; h; k'] \quad | \quad [k : \mathbf{newloc}_\gamma(l).P; k'] \\
 & | & [k : \mathbf{eval}_\gamma(Q)@l.P; k''; k']
\end{array}$$

TABLE IX. CRμKLAIM SYNTAX

CRμKLAIM syntax extends RμKLAIM syntax on two respects. First, actions in CRμKLAIM are labeled by *references* γ, which act as variables for keys. Second, CRμKLAIM introduces process $\mathbf{roll}(\gamma)$, which undoes the action labeled by γ. To simplify the technicalities, we change the syntax of memories as well, recording the continuation process also in **out**, **eval**, and **newloc** memories. Formally, we update the syntax of processes, actions and memories as reported in Table IX. Other syntactic categories are unchanged. At runtime references γ are replaced by keys k, thus we use ι to range over both γ and k. If a_γ denotes an action labeled by γ, then γ is bound in $a_\gamma.P$ with scope P. The definition of initial nets in CRμKLAIM is extended w.r.t. Definition 1, by also requiring that they do not contain any $\mathbf{roll}(k)$ (the argument of **roll** is always a reference), nor free occurrences of references. Well-formedness changes accordingly. Structural congruence coincides with the one of RμKLAIM. For simplicity we denote memories as $[k : a.P; \xi]$, where a is one of the CRμKLAIM actions and ξ is the additional information (e.g., the remaining keys in an **out** memory, the read tuple and the continuation key in an **in** memory, and so on) . For readability's sake we omit references when they are not relevant.

The following result will help us in the definition of CRμKLAIM semantics.

Lemma 6 (Net normal form): For any CRμKLAIM net N, we have:

$$\begin{aligned}
N \equiv (\nu \tilde{z}) \Big\|_{l \in L} \Big(l :: \prod_{i \in I} (k_i : P_i) \mid \prod_{j \in J} [k_j : a_j.P_j; \xi_j] \mid \\
\prod_{h \in H} (k_h \prec (k_h^2, k_h^3)) \mid \prod_{x \in X} (k_x : \langle et_x \rangle) \mid \\
\prod_{w \in W} [k_w^1 : \mathbf{in}_{\gamma_w}(T_w)@l_w.P_w; k_w^2 : \langle t_w \rangle; k_w^3] \mid \\
\prod_{y \in Y} [k_y^1 : \mathbf{read}_{\gamma_y}(T_y)@l_y.P_y; k_y^2; k_y^3] \Big)
\end{aligned}$$

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

Definition 5 (CRμKLAIM semantics): The CRμKLAIM operational semantics consists of a *forward reduction relation* $\mapsto_c$ and a *backward reduction relation* $\rightsquigarrow_c$. The backward reduction relation is the smallest evaluation-closed relation satisfying rule *(Roll)* in Table X. The forward reduction relation is the smallest evaluation-closed relation satisfying the forward rules in Table X. These rules are the forward rules of Table VII extended to instantiate γ with the proper key.

Backward reductions in CRμKLAIM correspond to executions of the **roll** operator. 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 $\mathbf{roll}(k)$. The semantics involves many subtleties, related to the behavior of the different actions. However, we define just one rule, *(Roll)*, capturing all of them.

The $\mathbf{roll}(k)$ operator should undo *all* the actions depending on the target action k, and *only* them. The *all* part is captured by the notion of completeness (Definition 3), 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. As a result of the reduction, M disappears, leaving just the process $k : a.P$ that was inside the memory. If the action a was an **in**, then also the consumed tuple should be restored. This is the role of N_t. Also, unless the locality containing the **roll** has been created by a descendant of k, it has to be preserved. This is the role of N_l. Finally, resources taken by the computation from the context should be given back to the context. This is the role of $N \lightning_k$ (see Definition 8).

We now define formally the notations used in the definition of the semantics. Causal dependence among keys and localities is needed for k-dependence.

Definition 6 (Causal dependence): Let N be a CRμKLAIM net and T_N the set of keys and localities in N. The relation $<:_N$ on T_N is the smallest preorder (i.e., reflexive and transitive relation) satisfying:

- $k <:_N k'$ if one of $[k : \mathbf{out}_\gamma(t)@l.P; k_1; k_2]$, $[k : \mathbf{eval}_\gamma(Q)@l.P; k_1; k_2]$, $k \prec (k_1, k_2)$ occurs in N, with $k' = k_1$ or $k' = k_2$;
- $k <:_N k'$ if one of $[k_1 : \mathbf{in}_\gamma(T)@l.P; k_2 : \langle et \rangle; k']$, $[k_1 : \mathbf{read}_\gamma(T)@l.P; k_2; k']$ occurs in N, with $k = k_1$ or $k = k_2$;

$$\frac{\begin{array}{c} M = (\nu\tilde{z})l :: k' : \mathbf{roll}(k) \parallel l' :: [k : a.P; \xi] \parallel N \qquad k <: M \qquad \texttt{complete}(M) \\ N_t = l'' :: h : \langle t\rangle \text{ if } a = \mathbf{in}_\gamma(T)@l'' \wedge \xi = h : \langle t\rangle ; k'', \text{ otherwise } N_t = \mathbf{0} \qquad N_l = \mathbf{0} \text{ if } k <:_M l, \text{ otherwise } N_l = l :: \mathbf{empty}\end{array}}{(\nu\tilde{z})l :: k' : \mathbf{roll}(k) \parallel l' :: [k : a.P; \xi] \parallel N \rightsquigarrow_c l' :: k : a.P \parallel N_t \parallel N_l \parallel N \not\downarrow_k} \textit{(Roll)}$$

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

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

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

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

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

TABLE X. CRμKLAIM OPERATIONAL SEMANTICS

- $k <:_N z$ if $[k : \mathbf{newloc}_\gamma(l).P; k']$ occurs in N, with $z = l$ or $z = k'$;
- $l <:_N k$ if $l :: k : P$ or $l :: k : \langle et\rangle$ occurs in N.

Note that for action **out** the continuation and the tuple depend on the action, while for actions **in** and **read** the continuation depends on both the action and the tuple. The last clause specifies that tuples and processes depend on the locality where they are. We can now define k-dependence.

Definition 7 (k-dependence): Let N be a CRμKLAIM net in normal form (see Lemma 6). Net N is $k-$dependent, written $k <: N$, if: (i) for every $i \in I \cup J \cup H \cup X$ we have $k <:_N k_i$; (ii) for every $i \in W \cup Y$ we have $k <:_N k_i^1$ or $k <:_N k_i^2$; and (iii) for every $z \in \tilde{z}$ we have $k <:_N z$.

We now describe the resources taken from the environment that need to be restored. We start with a simple example.

Example 2: Consider the following net:

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

After two steps the net becomes:

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

Performing $\mathbf{roll}(k)$ should lead back to the initial state. Releasing only the content of the target memory is not enough, since also the tuple $k' : \langle foo1\rangle$ should be released. This tuple is restored by $N \not\downarrow_k$ in rule *(Roll)*, since it is in a memory in N, but k' does not depend on k.

Projection, defined below, should release the tuples consumed by **in** actions which are undone, and also **in** and **read** actions that accessed a tuple created by an **out** action that is undone. Resources are released only if they do not depend on the key k of the **roll**.

Definition 8 (Projection): Let N be a net in normal form (see Lemma 6). If $k \notin \tilde{z}$ then:

$$\begin{aligned} N \not\downarrow_k \equiv (\nu\tilde{z}) \prod_{l \in L'}^{\parallel} \Big(l :: \prod_{w \in W'} k_w^1 : \mathbf{in}_{\gamma_w}(T_w)@l_w.P_w \ \mid \\ \prod_{y \in Y'} k_y^1 : \mathbf{read}_{\gamma_y}(T_y)@l_y.P_y \Big) \\ \prod_{w \in W''}^{\parallel} \Big(l_w{::}k_w^2 : \langle t_w\rangle \Big) \end{aligned}$$

where $L' = \{l \in L \,|\, k \not<:_N l\}$, $W' = \{w \in W \,|\, k \not<:_N k_w^1\}$, $Y' = \{y \in Y \,|\, k \not<:_N k_y^1\}$ and $W'' = \{w \in W \,|\, k \not<:_N k_w^2\}$.

We show now that CRμKLAIM is indeed a controlled version of RμKLAIM. Let `erCon` be a function from CRμKLAIM nets to RμKLAIM nets which is the identity but for replacing $\mathbf{roll}(\iota)$ with **nil**, and removing continuations inside memories for **out**, **eval** and **newloc**, and references γ.

Theorem 2: Given a CRμKLAIM net N, if $N \mapsto_c M$ then $\texttt{erCon}(N) \mapsto_r \texttt{erCon}(M)$ and if $N \rightsquigarrow_c M$ then $\texttt{erCon}(N) \rightsquigarrow_r^+ \texttt{erCon}(M)$ where $\rightsquigarrow_r^+$ is the transitive closure of $\rightsquigarrow_r$.

V. A FRANCHISING SCENARIO

In this section, we apply our reversible languages to a simplified but realistic franchising scenario, where a number of franchisees affiliate to a franchisor and determine the price of goods to expose to their customers. Each *franchisee* obtains a lot of goods from the *market*, gets the suggested price from the corresponding *franchisor*, possibly modifies it according to some local policy, and then publishes the computed price. In case of errors, e.g., the computed price is not competitive, franchisees can change price and, possibly, franchisor by undoing and performing again the activities described above. Notably, this does not affect the franchisors and the other franchisees. Instead, when a franchisor needs to change the suggested

$$\begin{array}{l}
\mathit{market} :: k_1 : \langle \text{“}\mathit{lot}\text{”}, 100, \mathit{franchisor_1} \rangle \;\mid\; k_2 : \langle \text{“}\mathit{lot}\text{”}, 100, \mathit{franchisor_1} \rangle \;\mid\; \ldots \mid\; k_3 : \langle \text{“}\mathit{lot}\text{”}, 200, \mathit{franchisor_2} \rangle \;\mid\; \ldots \\
\parallel \mathit{franchisor_1} :: k_4 : \mathbf{out}_{\gamma_1}(\text{“}\mathit{suggPrice}\text{”}, \mathit{price}())@\mathit{franchisor_1}.P_1 \parallel \mathit{franchisor_2} :: k_5 : \mathbf{out}_{\gamma_2}(\text{“}\mathit{suggPrice}\text{”}, \mathit{price}())@\mathit{franchisor_2}.P_2 \\
\parallel \mathit{franchisee}_1 :: k_6 : \mathbf{in}_{\gamma_3}(\text{“}\mathit{lot}\text{”}, !x_q, !u_{fr})@\mathit{market}\,.\mathbf{read}(\text{“}\mathit{suggPrice}\text{”}, !x_{pr})@u_{fr}\,.\mathbf{out}(\text{“}\mathit{price}\text{”}, \mathit{applyLocalPolicy_1}(x_{pr}))@\mathit{franchisee}_1.Q_1 \\
\parallel \mathit{franchisee}_2 :: k_7 : \mathbf{in}_{\gamma_4}(\text{“}\mathit{lot}\text{”}, !x_q, !u_{fr})@\mathit{market}\,.\mathbf{read}(\text{“}\mathit{suggPrice}\text{”}, !x_{pr})@u_{fr}\,.\mathbf{out}(\text{“}\mathit{price}\text{”}, \mathit{applyLocalPolicy_2}(x_{pr}))@\mathit{franchisee}_2.Q_2
\end{array}$$

TABLE XI. **CRμKLAIM** SPECIFICATION OF THE FRANCHISING SCENARIO

$$\begin{array}{l}
(\nu k_4'')(\mathit{market} :: \ldots \mid\; k_3 : \langle \text{“}\mathit{lot}\text{”}, 200, \mathit{franchisor_2} \rangle \;\mid\; \ldots \\
\parallel (\nu k_4')\; \mathit{franchisor_1} :: k_4' : P_1[k_4/\gamma_1] \mid k_4'' : \langle \text{“}\mathit{suggPrice}\text{”}, 170 \rangle \mid\; [k_4 : \mathbf{out}_{\gamma_1}(\text{“}\mathit{suggPrice}\text{”}, \mathit{price}())@\mathit{franchisor_1}.P_1; k_4''; k_4'] \\
\parallel (\nu k_5', k_5'')\; \mathit{franchisor_2} :: k_5' : P_2[k_5/\gamma_2] \mid k_5'' : \langle \text{“}\mathit{suggPrice}\text{”}, 160 \rangle \mid\; [k_5 : \mathbf{out}_{\gamma_2}(\text{“}\mathit{suggPrice}\text{”}, \mathit{price}())@\mathit{franchisor_2}.P_2; k_5''; k_5'] \\
\parallel (\nu k_6', k_6'', k_6''', k_6'''')\mathit{franchisee}_1 :: k_6''' : Q_1[k_6/\gamma_3] \;\mid\; k_6'''' : \langle \text{“}\mathit{price}\text{”}, 180 \rangle \\
\qquad\qquad\qquad\qquad \mid\; [k_6 : \mathbf{in}_{\gamma_3}(\text{“}\mathit{lot}\text{”}, !x_q, !u_{fr})@\mathit{market}.Q_1'; k_1 : \langle \text{“}\mathit{lot}\text{”}, 100, \mathit{franchisor_1} \rangle; k_6'] \\
\qquad\qquad\qquad\qquad \mid\; [k_6' : \mathbf{read}(\text{“}\mathit{suggPrice}\text{”}, !x_{pr})@\mathit{franchisor_1}.Q_1''; k_4''; k_6''] \\
\qquad\qquad\qquad\qquad \mid\; [k_6'' : \mathbf{out}(\text{“}\mathit{price}\text{”}, \mathit{applyLocalPolicy_1}(\mathit{170}))@\mathit{franchisee}_1.Q_1; k_6''''; k_6'''] \\
\parallel (\nu k_7', k_7'', k_7''', k_7'''')\mathit{franchisee}_2 :: k_7''' : Q_2[k_7/\gamma_4] \;\mid\; k_7'''' : \langle \text{“}\mathit{price}\text{”}, 185 \rangle \\
\qquad\qquad\qquad\qquad \mid\; [k_7 : \mathbf{in}_{\gamma_4}(\text{“}\mathit{lot}\text{”}, !x_q, !u_{fr})@\mathit{market}.Q_2'; k_2 : \langle \text{“}\mathit{lot}\text{”}, 100, \mathit{franchisor_1} \rangle; k_7'] \\
\qquad\qquad\qquad\qquad \mid\; [k_7' : \mathbf{read}(\text{“}\mathit{suggPrice}\text{”}, !x_{pr})@\mathit{franchisor_1}.Q_2''; k_4''; k_7''] \\
\qquad\qquad\qquad\qquad \mid\; [k_7'' : \mathbf{out}(\text{“}\mathit{price}\text{”}, \mathit{applyLocalPolicy_2}(\mathit{170}))@\mathit{franchisee}_2.Q_2; k_7''''; k_7''']\,)
\end{array}$$

TABLE XII. **CRμKLAIM** NET OF THE FRANCHISING SCENARIO (AFTER A FEW FORWARD STEPS)

price, it performs a backward computation that involves all the affiliated franchisees. For the sake of presentation, hereafter we consider a scenario consisting of the market, two franchisors and two franchisees.

The whole scenario is rendered in CRμKLAIM as the net in Table XI, where:

$$\begin{array}{rcl}
P_1 & = & \mathbf{in}(\text{“}\mathit{chgPr}\text{”})@\mathit{franchisor_1}.\mathbf{roll}(\gamma_1) \\
P_2 & = & \mathbf{in}(\text{“}\mathit{chgPr}\text{”})@\mathit{franchisor_2}.\mathbf{roll}(\gamma_2) \\
Q_1 & = & \mathbf{in}(\text{“}\mathit{chgPr}\text{”})@\mathit{franchisee_1}.\mathbf{roll}(\gamma_3) \\
Q_2 & = & \mathbf{in}(\text{“}\mathit{chgPr}\text{”})@\mathit{franchisee_2}.\mathbf{roll}(\gamma_4)
\end{array}$$

The market is a storage of tuples, representing lots of goods, of the form $\langle \text{“}\mathit{lot}\text{”}, v, l \rangle$, where v indicates a number of items and l the locality of the franchisor providing the lot. Each franchisor is a node executing a process that produces the suggested price, by resorting to a (non specified) function $\mathit{price}()$. Then, it waits for a *change price* request (i.e., a tuple $\langle \text{“}\mathit{chgPr}\text{”} \rangle$) to trigger the rollback of the executed activity (by means of the **roll** operator). Such tuple could be generated by a local process monitoring the selling trend that we leave unspecified and omit. The franchisees are nodes executing processes with the same structure. Each of them first gets a lot from the market, by consuming a lot tuple. Then, it reads the suggested price from the corresponding franchisor and uses it to determine the local price (using the unspecified function $\mathit{applyLocalPolicy}_i(\cdot)$). Finally, similarly to the franchisor process, it waits for a change price request and possibly rolls back.

Consider a net evolution where the two franchisors produce their suggested prices (170 and 160 cents, respectively) and the two franchisees acquire the first two lots, read the suggested price and publish their local prices (by increasing the suggested price by 10 and 15 cents, respectively). The resulting net is shown in Table XII, where Q_i' and Q_i'' denote the continuations of the **in** and **read** actions.

If $\mathit{franchisee}_1$ needs to change its lot of goods, a tuple $\langle \text{“}\mathit{chgPr}\text{”} \rangle$ is locally produced and, then, the rollback is triggered by $\mathbf{roll}(k_6)$. In this way, the memory of the first **in** will be directly restored (and its forward history deleted), rather than undoing action-by-action the forward execution (as in RμKLAIM). As expected, the backward step does not affect $\mathit{franchisee}_2$. Instead, if $\mathit{franchisor_1}$ wants to change the price stored in $k_4'' : \langle \text{“}\mathit{suggPrice}\text{”}, 170 \rangle$, it undoes action $k_4 : \mathbf{out}(\text{“}\mathit{suggPrice}\text{”}, \mathit{price}())@\mathit{franchisor_1}$ by also involving the **read** memories within $\mathit{franchisee}_1$ and $\mathit{franchisee}_2$, because k_4'' occurs within them, and all the occurrences of k_4'' must be considered to apply rule *(Roll)*. Thus, the projection operation $\cdot \lightning_{k_4}$ will restore the **read** actions from their memories. This allows the franchisees affiliated to this franchisor to adjust their local prices.

If we replace the actions $\mathbf{read}(\text{“}\mathit{suggPrice}\text{”}, !x_{pr})@u_{fr}$ by $\mathbf{in}(\text{“}\mathit{suggPrice}\text{”}, !x_{pr})@u_{fr}.\, \mathbf{out}(\text{“}\mathit{suggPrice}\text{”}, x_{pr})@u_{fr}$, the (undesired) side effect already discussed in the simple example shown in Section III arises: if a franchisee changes its lot, it may involve in the undo procedure the other franchisees.

Notably, by setting processes P_i and Q_i to **nil** and removing all references, we obtain a RμKLAIM specification, which exhibits computations as the one above (Theorem 2), but also other computations mixing forward and backward actions in an uncontrolled way that are undesired in this scenario.

VI. RELATED WORK AND CONCLUSION

The history of reversibility in a sequential setting is already quite long [16], [7]. Our work however concerns causal-consistent reversibility, which has been introduced in [5]. This 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, has been proposed in [18]. 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 [4], while reduction semantics has been studied for HOπ [14] and μOz [17]. We are closer to [14], which uses modular memories similar to ours. Controlled reversibility has been studied first in [6], introducing irreversible actions, then in [1], where energy parameters drive the evolution of the process, and in [19], where a non-reversible controller drives a reversible process. For an exhaustive survey on causal-consistent reversibility we refer to [15].

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 [14], [13], which were obtained in the simpler framework of HOπ.

We have not yet transported to μKLAIM all the results in [14], [13], [12]. The main missing results are an encoding from the reversible calculus into the basic one [14], a more low level semantics for controlled reversibility [13], and the introduction of alternatives to avoid repeating the same error after a rollback [12]. A full porting of the results above would need to study the behavioral theory of RμKLAIM and CRμKLAIM, which is left for future work. We outline however below how the issues above can be faced.

The most natural way to add alternatives [12] to CRμKLAIM 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 a tuple behaves as $k : \langle foo\rangle$, but it becomes $k : \langle foo1\rangle$ when it is inside a memory of an $\mathbf{in}$, and a $\mathbf{roll}$ targeting the memory is executed. As in HOπ, such a simple mechanism considerably increases the expressiveness.

A faithful encoding of RμKLAIM and CRμKLAIM into μKLAIM itself would follow the lines of [14]. Its definition would be simpler than that for reversible HOπ [14], since tuple spaces provide a natural storage for memories and connectors. Such encoding will pave the way to the use of KLAVA [3], a framework providing run-time support for KLAIM actions in Java, to experiment with reversible distributed applications.

A low-level semantics for CRμKLAIM, more suitable to an implementation, should follow the idea of [13], based on an exploration of the causal dependences of the memory pointed by the $\mathbf{roll}$. However, one has to deal with read dependences, and at this more concrete level the use of restriction is no more viable. Thus, one should keep in each tuple the keys of processes that have read it.

Acknowledgment

The authors gratefully thank the anonymous referees for their useful remarks. This work was partially supported by Italian MIUR PRIN Project CINA Prot. 2010LHT4KM and by the French ANR project REVER n. ANR 11 INSE 007.

References

[1] G. Bacci, V. Danos, and O. Kammar. On the Statistical Thermodynamics of Reversible Communicating Processes. In *CALCO*, volume 6859 of *LNCS*, pages 1–18. Springer, 2011.

[2] L. Bettini, V. Bono, R. De Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The Klaim Project: Theory and Practice. In *Global Computing*, volume 2874 of *LNCS*, pages 88–150. Springer, 2003.

[3] L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Package for Distributed and Mobile Applications. *Software - Pract. Exper.*, 32(14):1365–1394, 2002.

[4] I. D. Cristescu, J. Krivine, and D. Varacca. A Compositional Semantics for the Reversible Pi-calculus. In *LICS*, pages 388–397. IEEE Press, 2013.

[5] V. Danos and J. Krivine. Reversible Communicating Systems. In *CONCUR*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004.

[6] V. Danos and J. Krivine. Transactions in RCCS. In *CONCUR*, volume 3653 of *LNCS*, pages 398–412. Springer, 2005.

[7] V. Danos and L. Regnier. Reversible, Irreversible and Optimal Lambda-Machines. *Theor. Comput. Sci.*, 227(1-2), 1999.

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

[9] D. Gelernter. Generative Communication in Linda. *ToPLaS*, 7(1):80–112, 1985.

[10] E. Giachino, I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-Consistent Reversibility in a Tuple-Based Language (TR). http://www.cs.unibo.it/~lanese/work/klaimrev-TR.pdf.

[11] D. Gorla and R. Pugliese. Resource Access and Mobility Control with Dynamic Privileges Acquisition. In *ICALP*, volume 2719 of *LNCS*, pages 119–132. Springer, 2003.

[12] I. Lanese, M. Lienhardt, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Concurrent Flexible Reversibility. In *ESOP*, volume 7792 of *LNCS*, pages 370–390. Springer, 2013.

[13] I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling Reversibility in Higher-Order Pi. In *CONCUR*, volume 6901 of *LNCS*, pages 297–311. Springer, 2011.

[14] I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversing Higher-Order Pi. In *CONCUR*, volume 6269 of *LNCS*, pages 478–493. Springer, 2010.

[15] I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-consistent reversibility. *Bulletin of the EATCS*, 114, 2014.

[16] G. Leeman. A Formal Approach to Undo Operations in Programming Languages. *ToPLaS*, 8(1), 1986.

[17] M. Lienhardt, I. Lanese, C. A. Mezzina, and J.-B. Stefani. A Reversible Abstract Machine and Its Space Overhead. In *FMOODS/FORTE*, volume 7273 of *LNCS*, pages 1–17. Springer, 2012.

[18] I. Phillips and I. Ulidowski. Reversing Algebraic Process Calculi. *J. Log. Algebr. Program.*, 73(1-2):70–96, 2007.

[19] I. Phillips, I. Ulidowski, and S. Yuen. A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway. In *RC*, volume 7581 of *LNCS*, pages 218–232. Springer, 2012.