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 \( \mu \text{KLAIM} \), 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 \( \mu \text{KLAIM} \) read primitive, reading a tuple without consuming it, cannot be matched using channel-based communication. We illustrate the reversible extensions of \( \mu \text{KLAIM} \) 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], [17], \( \pi \)-calculus [4], higher-order \( \pi \) [13] and \( \mu \text{Oz} \) [16]. 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 \( \mu \text{KLAIM} \) [10], a formal language based on distributed tuple spaces derived from the coordination language KLAIM [8]. \( \mu \text{KLAIM} \) 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, \( \mu \text{KLAIM} \) 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 \textbf{in} and \textbf{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 in a simple way properties of our reversibility mechanism. In particular, we show that reversible \( \mu \text{KLAIM} \) (\( R\mu \text{KLAIM} \) for short) is causally consistent (Theorem 1), and that its forward computations correspond to \( \mu \text{KLAIM} \) 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 \( R\mu \text{KLAIM} \) process may always diverge by doing and undoing the same action forever.

To solve this problem, we build on top of \( R\mu \text{KLAIM} \) a language with controlled reversibility, \( CR\mu \text{KLAIM} \) (Section IV). \( CR\mu \text{KLAIM} \) computation normally proceeds forward, but the programmer may ask for a rollback using a dedicated roll operator. The roll 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 reliable applications. We put \( CR\mu \text{KLAIM} \) at work on a practical example about franchising (Section V). Proofs of main results are collected in Appendix.

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 programming distributed systems. Its main benefit with respect
Aℓ, i.e., locality variables and localities, are ranged over by node is a pair (evl)N

\[
\text{(Nets)} \; N \;
\text{(Components)} \; C \;
\text{(Processes)} \; P \;
\text{(Actions)} \; a \;
\text{(Tuples)} \; t \;
\text{(Evaluated tuples)} \; et \;
\text{(Templates)} \; T \;
\]

\[
\begin{align*}
\text{(Nets)} & \quad N := 0 \mid l : C \mid N_1 \mid N_2 \mid (vl)N \\
\text{(Components)} & \quad C := (st) \mid P \mid C_1 \mid C_2 \\
\text{(Processes)} & \quad P := \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \\
\text{(Actions)} & \quad a := \text{out}(t)@t \mid \text{eval}(P)@t \mid \text{in}(t)@t \mid \text{read}(T)@t \mid \text{newloc}(l) \\
\text{(Tuples)} & \quad t := c \mid ℓ \mid t_1, t_2 \\
\text{(Evaluated tuples)} & \quad et := v \mid l \mid et_1, et_2 \\
\text{(Templates)} & \quad T := c \mid ℓ \mid !x \mid !u \mid T_1, T_2
\end{align*}
\]

TABLE I. \( µKLAIM \) SYNTAX

\[
\begin{align*}
\text{(Monoid)} & \quad N \equiv 0 \equiv N \mid N_1 \equiv N_2 \equiv N_1 \\
\text{(RCom)} & \quad (vl_1)(vl_2)N \equiv (vl_2)(vl_1)N \\
\text{(PDef)} & \quad l :: l \equiv P \text{ if } A \equiv P \\
\text{(Ext)} & \quad N_1 \parallel (vl)(N_2) \parallel (vl)(N_1 \parallel N_2) \text{ if } l \notin fn(N_1) \\
\text{(Alpha)} & \quad N \equiv N' \text{ if } N =_α N' \\
\text{(Abs)} & \quad l :: C \equiv l :: (C \parallel \text{nil}) \\
\text{(Clone)} & \quad l :: C_1|C_2 \equiv l :: C_1 \parallel l :: C_2
\end{align*}
\]

TABLE II. \( µKLAIM \) STRUCTURAL CONGRUENCE

_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, cooperation 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 _locations_, ranged over by \( l, l', \ldots \) _locality variables_, ranged over by \( u, u', \ldots \), of _variable names_, ranged over by \( x, x', \ldots \), and of _process identifiers_, ranged over by \( A, B, \ldots \). Localities are the addresses (i.e., network references) of nodes and are the syntactic ingredient used to model administrative domains. In \( µKLAIM \), communicative objects are (evaluated) _tuples_, i.e., sequences of actual fields. Tuple fields may contain expressions, localities or locality variables. The syntax of _expressions_, ranged over by \( e, \ldots \), is deliberately not specified; we just assume that expressions contain _values_ (ranged over by \( v \) and _variable names_). _Names_, i.e., locality variables and localities, are ranged over by \( ℓ, ℓ', \ldots \). We assume each process identifier \( A \) has a single definition \( A \equiv P \), 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 _locality_ \( l \) is the address of the node and \( C \) is the hosted component. In the net \( (vl)N \), the scope of the name \( l \) is restricted to \( N \). \( 0 \) denotes the empty net.

Processes, the \( µKLAIM \) active computational units, are built up from the process \( \text{nil} \), that does not perform any action, using action prefixing \( a.P \), parallel compositions as \( P_1 \mid P_2 \), and process identifiers as \( A \). We may drop trailing \( \text{nils} \). Processes may be executed concurrently either at the same locality or at different localities and can perform five different basic operations, called actions.

Actions _out_, _in_ and _read_ manage data repositories by adding/withdrawing/accessing data. Action _eval_ activates a new thread of execution in a (possibly remote) node. Action _newloc_ permits to create new net nodes. All actions but _newloc_ indicate explicitly the locality where they will act. Actions _in_ and _read_ are blocking and exploit templates as patterns to select data in shared repositories. _Templates_ 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.

Localities and variables can be _bound_ inside processes and nets: _newloc(l).P_ binds name \( l \) in \( P \), and \( (vl)N \) binds \( l \) in \( N \). Prefixes _in(\ldots!x\ldots)@ℓ,P_ and _read(\ldots!x\ldots)@ℓ,P_ bind variable \( _\ell \) 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 \( =_α \), 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 II. Most of the laws are standard, while laws (Abs) and (Clone) are peculiar to this setting. Law (Abs) states that \( nil \) is the identity for \( \cdot \parallel \cdot \). 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 \( \cdot \parallel \cdot \)).

To define the reduction relation, we need an auxiliary pattern-matching function _match(_, __,_ _,_ ), defined by the rules in Table III, to verify the compliance of a tuple w.r.t. a template and to associate localities/values to variables bound in templates. 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 a template \( T \) and a tuple \( t \) do match, _match(T, t)_ 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 \( o \) to denote substitution composition and \( ε \) to denote the empty substitution.

We use function \( [\cdot] \) for evaluating tuples and templates.
match(v,v) = ε  match([2x,v]) = [v/εx]  match([l,l]) = ε
match(T1,t1) = σ1  match(T2,t2) = σ2

match(⟨(T1,T2),(t1,t2)⟩) = σ1 ∘ σ2

TABLE III. μKLAIM MATCHING RULES

| l | = et
| l :: out(t)@l'.P || l' :: nil → l :: P || l' :: (et) | (Out)

match(⟨T⟩,et) = σ

l :: in(T)@l'.P || l' :: (et) → l :: Pσ || l' :: nil

match(⟨T⟩,et) = σ

l :: read(T)@l'.P || l' :: (et) → l :: Pσ || l' :: (et)

l :: newloc(l').P → (l')(l :: P || l' :: nil) (New)

l :: eval(Q)@l'.P || l' :: nil → l :: P || l' :: Q (Eval)

TABLE IV. μKLAIM OPERATIONAL SEMANTICS

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 defined since the exact syntax of expressions is deliberately not specified. Notably, only evaluated tuples, denoted by (et), are stored in tuple spaces.

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 R is evaluation closed if it is closed under active contexts, i.e., N1 R N1' implies (N1 || N2) R (N1' || N2) and (νl)N1 R (νl)N1', and under structural congruence, i.e., N ≡ M R M' ≡ N' R N'.

Definition 2 (μKLAIM semantics): The μKLAIM reduction relation ⇒ is the smallest evaluation-closed relation satisfying the rules in Table IV.

All rules for (possibly remote) actions out, eval, in, and read require the existence of the target node l'. 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 l'. Rules (In) and (Read) require the target node to contain a tuple matching their template argument T. Similarly to out actions, such template must be evaluative. 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 in consumes the matched tuple, while action 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. In words, rμKLAIM nets (which include also information needed to enable reversibility) 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 [13], the technical development is considerably different.

We first present the syntax and operational semantics of rμKLAIM, by also resorting to a simple example that allows us to point out the peculiarity of the causality relationships produced by rμKLAIM constructs. Then, we show that rμKLAIM satisfies the typical properties expected from a reversible formalism.

**Syntax.** rμKLAIM syntax is in Table V. We do not report the syntax of actions, (evaluated) tuples, and templates, which is identical to that of μKLAIM (Table I). The main ingredients of rμKLAIM are keys k, uniquely identifying tuples and processes, memories µ, storing information needed for undoing past actions, and connectors k₁ ~ (k₂, k₃), tracking causality information. More precisely, we have the additional syntactic category of keys, ranged over by k, h, ... We use z to range over keys and localities. Keys uniquely identify processes and tuples. Uniqueness 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 only considering well-formed nets.

**Definition 3 (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 4) 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 five kinds of memories, one for 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 in and read store their original continuation P, since it cannot be recovered from the running one, which has been obtained by applying a substitution - a non reversible transformation.

Also, the memory for out stores the key k'' of the created tuple, while the memory for eval stores the key k'' of the spawned process, and the memory for in stores the consumed tuple h : (et). The memory for read only needs the key h of the read tuple, since the tuple itself is still available in the term and uniquely identified by key h. Connector k₁ ~ (k₂, k₃) recalls that processes with keys k₂ and k₃ originated from the split of process tagged by k₁. Finally, we distinguish empty localities, denoted by l :: empty, containing no information.

\[ N ::= 0 | l :: C | l :: empty | N₁ || N₂ | (νz)N \\
C ::= k :: (et) | k :: P | C₁ || C₂ | µ | k₁ × (k₂, k₃) \\
P ::= nil | a.P | P₁ || P₂ | A \\
µ ::= [k :: out(t)@l'; k''; k'] | [k :: in(T)@l.P; h : (et); k'] \\
     | [k :: read(T)@l.P; h; k'] | [k :: newloc(l); k'] \\
     | [k :: eval(Q)@l'; k''; k']
\]

**TABLE V. rμKLAIM SYNTAX**
from localities \(l : k : \texttt{nil}\) containing a \texttt{nil} process with its key \(k\), which may interact with a memory to perform a backward action.

**Operational semantics.** Structural congruence for \(R\mu\text{KLAIM}\) extends the one for \(\mu\text{KLAIM}\) in Table II to deal with keys: new rules (Garb) and (Split) and updated rules are reported in Table VI. Rules (RCom) and (Ext) now consider also restrictions on keys. Rule (PDef) now applies to process identifiers prefixed by keys. Rule (Abs) now does not create \texttt{nil} terms, but only empty keys (this is possible also in \(\mu\text{KLAIM}\), by combining 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 (R\(\mu\)KLAIM semantics):** The operational semantics of \(R\mu\text{KLAIM}\) consists of a forward reduction relation \(\Rightarrow_f\), and a backward reduction relation \(\Rightarrow_b\). 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 \(\mu\text{KLAIM}\) 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 requiring a parallel term \(l' :: \texttt{empty}\). 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 \(\langle e, t \rangle\), respectively. Also, a memory is created (in the locality where the \texttt{out} prefix was) storing all the relevant information. The corresponding backward rule, (OutRev), may trigger if a memory for \texttt{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 \texttt{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 \texttt{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 \(\mu\text{KLAIM}\) 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 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. 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 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 \(R\mu\text{KLAIM}\) read action and its possible implementations in the other reversible languages in the literature. The other reversible languages we are aware of 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 \(R\mu\text{KLAIM}\) by using an \texttt{in} followed by an \texttt{out}. To avoid introducing other syntaxes and semantics, we present the different behaviors inside \(R\mu\text{KLAIM}\). The difference is striking in a reversible setting, while it is less compelling when only forward actions are considered. Consider a \(R\mu\text{KLAIM}\) net \(N\) with three nodes, \(l_1\) hosting a tuple \(\langle \text{foo} \rangle\), and \(l_2\) and \(l_3\) hosting processes willing to access such tuple:

\[
N' = l_1 :: k_1 : \langle \text{foo} \rangle || l_2 :: k_2 : \text{in}([\text{foo}])@l_1, \text{out}(\text{foo})@l_1, P \quad || l_3 :: k_3 : \text{in}([\text{foo}])@l_1, \text{out}(\text{foo})@l_1, P'
\]

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

\[
\begin{align*}
& (v k'_2, k'_3, k''_{out}, k'_1, k''_{out})(l_1 :: k''_{out} : \langle \text{foo} \rangle) \\
& || l_2 :: k'_2 :: P || (k'_2 : \text{in}([\text{foo}])@l_1, l_2, \text{out}(\text{foo})@l_1, P; k_1 : \langle \text{foo} \rangle; k'_2) \\
& || l_3 :: k'_3 :: P' || (k'_3 : \text{in}([\text{foo}])@l_1, l_3, \text{out}(\text{foo})@l_1, P'; k''_{out} : \langle \text{foo} \rangle; k'_3) \\
\end{align*}
\]

Now, the process in \(l_2\) cannot immediately perform a backward step, since it needs the tuple \(k''_{out} : \langle \text{foo} \rangle\) in \(l_1\), while only \(k''_{out} : \langle \text{foo} \rangle\) is available. The former tuple has been consumed by the \texttt{in} action at \(l_3\) (see the corresponding memory stored in \(l_2\)) and then replaced by the latter by the \texttt{out} action at \(l_3\). This means that to perform the backward step of the process in \(l_2\) one needs first to perform a backward computation of 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 \(\rho\pi\) [13] or an output process in [5], [17], [4].

The problem can be solved in \(R\mu\text{KLAIM}\) using the read primitive. Let us replace in the net above each sequence of \texttt{in} and \texttt{out} with a read:

\[
N = l_1 :: k_1 : \langle \text{foo} \rangle || l_2 :: k_2 : \text{read}(\text{foo})@l_1, P || l_3 :: k_3 : \text{read}(\text{foo})@l_1, P'
\]

By executing the two \texttt{read} actions (the order is now irrelevant), the net \(N\) evolves to:

\[
\begin{align*}
& (v k'_2, k'_3) (l_1 :: k_1 : \langle \text{foo} \rangle) \\
& || l_2 :: k'_2 :: P || (k'_2 : \text{read}(\text{foo})@l_1, P; k_1 ; k'_2) \\
& || l_3 :: k'_3 :: P' || (k'_3 : \text{read}(\text{foo})@l_1, P'; k''_{out} ; k'_3)
\end{align*}
\]

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

\[
\begin{align*}
& (v k'_2, k'_3) (l_1 :: k_1 : \langle \text{foo} \rangle) \\
& || l_2 :: k'_2 : \text{read}(\text{foo})@l_1, P || l_3 :: k'_3 : \text{read}(\text{foo})@l_1, P'; k_1 ; k'_3)
\end{align*}
\]
First introduce some auxiliary definitions.

- \( \mu \) \((vz) \equiv \nu(vz) \) \(N \equiv \nu(vz) \) \(N \) \(N \)

**Definition 5:** Given memories of the shape

\[
[k : \text{out}(\ell)@l]; k'; k], \quad [k : \text{in}(\ell)@l; P; h'; (\ell); k'], \quad [k : \text{read}(T)@l; P; h; k'], \quad [k : \text{eval}(Q)@l; k]; k', k'], \quad [k : \text{newloc}(l); k],
\]

and connectors of the shape \( k \sim (k', k'') \), the head of those memories/connectors is \( k \), while the tail consists of \( k' \) and, when they occur, \( k'' \) or \( h \). Keys \( h \) and \( h' \) occur in an input position.

**Basic properties.** We now show that \( \mu K\text{LAIM} \) respects the \( \mu K\text{LAIM} \) semantics, and that it is causally consistent. We first introduce some auxiliary definitions.

**Lemma 2:** Let \( N \) and \( M \) be two \( \mu K\text{LAIM} \) nets such that \( N \rightarrow_M M \). Then \( \text{erN}(N) \rightarrow_M \text{erN}(M) \).

**Lemma 3:** Let \( R \) and \( S \) be two \( \mu K\text{LAIM} \) nets such that \( R \rightarrow S \). Then for all \( \mu K\text{LAIM} \) nets \( M \) such that \( \text{erN}(M) = R \) there exists a \( \mu K\text{LAIM} \) net \( N \) such that \( M \rightarrow_N N \) and \( \text{erN}(N) \equiv S \).

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

**Lemma 4 (Loop lemma):** For all well-formed \( \mu K\text{LAIM} \) nets \( N \) and \( M \), the following holds: \( N \rightarrow_M M \leftrightarrow_M \rightarrow N \).

We now move to the proof that \( \mu K\text{LAIM} \) 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 \( \mu K\text{LAIM} \).

In a forward reduction \( N \rightarrow_M M \) we call forward memory the memory \( \mu \) created by that reduction, i.e., \( \mu \) does not occur in \( N \) and occurs in \( M \). Similarly, in a backward reduction \( N \rightarrow_M M \) we call backward memory the memory \( \mu \) deleted by that reduction, i.e., \( \mu \) occurs in \( N \) and does not occur in \( M \). We call transition a triplet of the form \( N \xrightarrow{\alpha} M \), or \( N \xleftarrow{\alpha} M \), where \( N, M \) are well-formed nets, and \( \alpha \) is the forward/backward memory of the reduction. We call \( N \) the source of the transition, \( M \) its target. We let \( \eta \) range over labels \( \mu \rightarrow_M \) and \( \mu \rightarrow_M \). If \( \eta = \mu \rightarrow_M \), then \( \eta_0 = \mu \rightarrow_M \), and vice versa. Without loss of generality we restrict our attention to transitions derived without using \( \alpha \)-conversion. We also assume that when structural rule (Split) is applied from left to right creating a connector \( h \sim (k_1, k_2) \), 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 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 $\delta$ range over transitions and $\theta$ range over traces. If $\delta$ is a transition then $\delta^\rightarrow$ denotes its inverse. Notions of source, target and composability extend naturally to traces. We denote with $\epsilon_M$ the empty trace with source $M$, and with $\theta_1;\theta_2$ the composition of two composable traces $\theta_1$ and $\theta_2$. The stamp $\lambda(\mu,\rightarrow)$ of a memory $\mu$ is:

$$\lambda([k : \text{out}(t)@l; k'; k'']) = \{k, k', k'', r(l)\}$$
$$\lambda([k : \text{in}(T)@l; P; k' : (et); k'']) = \{k, k', k'', r(l)\}$$
$$\lambda([k : \text{read}(T)@l; P; k' : k'']) = \{k, r(k'', k', r(l))\}$$
$$\lambda([k : \text{eval}(Q)@l; k; k'']) = \{k, k', k'', r(l)\}$$
$$\lambda([k : \text{newloc}(l); k'']) = \{k, k', l\}$$

We set $\lambda(\mu,\rightarrow) = \lambda(\mu,\rightarrow)$. The stamp of a memory defines the resources used by the corresponding transitions. The notation $r(z)$ highlights that resource $z$ (either key or locality name) is used in read-only modality. Notably, all actions but $\text{newloc}$ use a locality name in a read-only modality. We use $\kappa$ to range over tags $\mathbf{r}(z)$ and $\eta$. We define the closure w.r.t. a net $N$ of a tag $\kappa$ as closure$_N(\kappa) = \{\kappa\} \cup \text{closure}_N(h)$ if $\kappa = k_1$ or $\kappa = k_2$ and $\kappa < k_1$ and $\kappa < k_2$ occurs in $N$, $\{\kappa\}$ otherwise. We define the closure over a set $K$ of tags as closure$_N(K) = \bigcup_{\kappa \in K} \text{closure}_N(\kappa)$. The closure captures that a connector with $h < (k_1, k_2)$ means that resources $k_1$ and $k_2$ are part of resource $h$.

**Definition 7 (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:

1. $z \in \text{closure}_{M || N_1}(\lambda(\eta_1))$ and $z \in \text{closure}_{M || N_2}(\lambda(\eta_2))$.
2. $\mathbf{r}(z) \in \lambda(\eta_1)$ and $\mathbf{r}(z) \in \lambda(\eta_2)$.
3. $z \in \text{closure}_{M || N_1}(\lambda(\eta_1))$ and $\mathbf{r}(z) \in \lambda(\eta_2)$.

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

Essentially, two transitions are in conflict if both use the same resource, and at most one of them uses it in read-only modality. The definition however has to keep into account a few subtleties in the way keys are managed, thus we present a few examples to help clarify it.

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

$$P = \text{out}(\text{foo})@l.(\text{out}(\text{foo}1)@l | \text{out}(\text{foo}2)@l)$$

After a step the net becomes:

$$l :: P \rightarrow ((k', k'')(l :: k : (\text{out}(\text{foo}1)@l | \text{out}(\text{foo}2)@l) | [k : \text{out}(\text{foo})@l; k'; k''] \parallel l :: k' : (\text{foo})) = N$$

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

$$N \rightarrow ((k') (l :: k : P \parallel l :: \text{empty}) \equiv l :: k : P$$

Another option is to execute the action $\text{out}(\text{foo}1)@l$.

$$N \equiv ((k', k'', k_1', k_2') \parallel (l :: k' : (\text{out}(\text{foo}1)@l | \text{out}(\text{foo}2)@l) | k' : (\text{foo}))) | [k : \text{out}(\text{foo})@l; k'; k''])$$

We highlighted in the derivation the use of structural rule (Split). The stamp of the first transition with source $N$ is $\{k, k', k''\}$. The stamp of the second transition with source $N$ is $\{k_1', k_2', k', k''\}$. Only the use of the closure on the second out, adding $k'$ to the resources used by the second transition, allows 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 $\delta_1/\delta_2 = N_2 \xrightarrow{\eta_1} N$.

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

$$\delta_1; \delta_2/\delta_1 \equiv \delta_2/\delta_2 \delta_1/\delta_2 \equiv \epsilon_{\text{source}(\delta)} \delta_1/\delta_2 \equiv \epsilon_{\text{target}(\delta)}$$

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, on one side, that causal equivalent traces can be reversed in the same ways, and, on the other side, that traces which are not causal equivalent lead to distinct nets.

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

## IV. Controlled reversibility

In this section we define CR$\mu$KLAIM, an extension of $\mu$KLAIM featuring an explicit rollback facility to control $R\mu$KLAIM reverting capabilities. This allows us to exploit reversibility to program recovery activities inside $\mu$KLAIM applications. We follow the general approach of [12], but we have to adapt it in order to deal with the interplay of the different $\mu$KLAIM actions.

$$P ::= \text{nil} | a.P | P_1 | P_2 | A | \text{roll}(a)$$

$$a ::= \text{out}_{\gamma}(t)@l | \text{eval}(P)@l | \text{in}_{\delta}(T)@l | \text{read}_{\gamma}(T)@l | \text{newloc}_{\delta}(l)$$

$$\mu ::= [k : \text{out}_{\gamma}(t)@l; P; k'; k''] | [k : \text{in}_{\delta}(T)@l; P; h; k'] | [k : \text{read}_{\gamma}(T)@l; P; h; k'] | [k : \text{newloc}_{\delta}(l); P; k']$$

**TABLE IX.** CR$\mu$KLAIM syntax

CR$\mu$KLAIM syntax extends R$\mu$KLAIM syntax on two respects. First, actions in CR$\mu$KLAIM are labeled by references $\gamma$, which act as variables for keys. Second, CR$\mu$KLAIM introduces
process roll(γ), 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 the memories for actions \texttt{out}, \texttt{eval}, and \texttt{newloc}. Formally, we update the syntax of processes, actions and memories as reported in Table IX. Other syntactic categories are unchanged. At runtime references \( \gamma \) are replaced by keys \( k \), thus we use \( \sigma \) to range over both \( \gamma \) and \( k \). If \( \alpha \gamma \) denotes an action labeled by \( \gamma \), then \( \gamma \) is bound in \( \alpha \gamma P \) with scope \( P \). The definition of initial nets in CRµKLAIM is extended w.r.t. Definition 3, by also requiring that they do not contain any \texttt{roll}(k) (the argument of \texttt{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 \( \xi \) is the additional information (e.g., the remaining keys in an \texttt{out} memory, the read tuple and the continuation key in an \texttt{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:

\[
N \equiv (v\mathcal{Z}) \prod_{l \in \mathcal{L}} \left( l :: \prod_{i \in I} (k_i : P_i) \prod_{j \in J} [k_j : a_j.P_j; \xi_j] \right) \\
\prod_{h \in H} (k_h \prec (k_h^2, k_h^3)) \prod_{x \in X} (k_x : (et_x)) \\
\prod_{w \in W} [k_w^1 : \texttt{in}_w(T_w)@l_w.P_w; k_w^2 : (t_w); k_w^3] \\
\prod_{y \in Y} [k_y^1 : \texttt{read}_y(T_y)@l_y.P_y; k_y^2; k_y^3] 
\]

where action \( a_j \) is neither in nor \texttt{read} for every \( j \in J \).

**Definition 8 (CRµKLAIM semantics):** The operational semantics of CRµKLAIM consists of a forward reduction relation \( \Rightarrow_{f} \) and a backward reduction relation \( \Rightarrow_{b} \). They are the smallest evaluation-closed relations satisfying the rules in Table X.

The forward rules are as for RµKLAIM, except for instantiating \( \gamma \) with the proper key. Backward reductions in CRµKLAIM correspond to executions of the \texttt{roll} operator. Since all the occurrences of references \( \gamma \) are bound, when a \texttt{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 \texttt{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 \texttt{roll}(k) operator should undo all the actions depending on the target action \( k \), and only them. The \emph{all} part is captured by the notion of completeness (Definition 6), and the \emph{only} part by a notion of \( k \)-dependence (written \( \prec \)) 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 \texttt{in}, then also the consumed tuple should be restored. This is the role of \( N_1 \). Also, unless the locality containing the \texttt{roll} has been created by a descendant of \( k \), it has to be preserved. This is the role of \( N_2 \). Finally, resources taken by the computation from the context should be given back to the context. This is the role of \( N_4 \).

We now define formally the notations used in the definition of the semantics, together with examples clarifying it.

**Definition 9 (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 \([k : \texttt{out}(t)@l.P; k_1; k_2], [k : \texttt{eval}(Q}@l.P; k_1; k_2], k < (k_1, k_2) \) occurs in \( N \), with \( k_1 = k' \) or \( k_2 = k' \);
- \( k <_N k' \) if one of \([k_1 : \texttt{in}(T)@l.P; k_2 : (et); k'], [k_1 : \texttt{read}(T}@l.P; k_2; k') \) occurs in \( N \), with \( k_1 = k \) or \( k_2 = k' \);
- \( k <_N \nu \) if \([k : \texttt{newloc}(l.P); k'] \) occurs in \( N \), with \( \nu = l \) or \( \nu = k' \);
- \( l <_N k \) if \( l :: k : P \) or \( l :: k : (et) \) occurs in \( N \).

Note that for action \texttt{out} the continuation and the tuple depend on the action, while for actions \texttt{in} and \texttt{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 10 (k-dependence):** Let \( N \) be a CRµKLAIM net in normal form (see Lemma 6). Net \( N \) is \( k \)-dependent, written \( k \prec N \), by overloading the definition of causal dependence among keys and localities if:

- 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 \( k <_N k_i^1 \) or \( k <_N k_i^2 \);
- for every \( z \in Z \) we have \( k <_N z \).

We now describe the resources taken from the environment that need to be restored. We start by presenting two examples.

**Example 3:** Consider the following net:

\[
l :: k :: \texttt{out}_\gamma(foo)@l.\texttt{in}(foo1)@l.\texttt{roll}(\gamma) || k' :: \langle foo1 \rangle
\]

After two steps the net becomes:

\[
(\forall k'' \in K^{'} : k''' \in K^{''} : (l :: k''' :: \texttt{roll}(k) || k'' :: \langle foo \rangle) \\

\quad || [k :: \texttt{out}_\gamma(foo)@l.\texttt{in}(foo1)@l.\texttt{roll}(\gamma) || k'' :: \langle foo1 \rangle@l.\texttt{roll}(k) || k' :: \langle foo \rangle@l.\texttt{roll}(k)]
\]

Performing \texttt{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 \( N_{4k} \) in rule (Roll), since it is in a memory in \( N \), but \( k' \) does not depend on \( k \).

**Example 4:** Consider the following net:

\[
l :: k :: \texttt{out}_\gamma(foo)@l.\texttt{roll}(\gamma) || l' :: k' :: \texttt{in}(foo)@l
\]
After the out of tuple \( (foo) \) at locality \( l \) followed by the in of the same tuple the net becomes:

\[
\begin{align*}
(\langle l, k''', \text{roll}(k) \rangle & : \text{out}(foo)@l.\text{roll}(\gamma) ; k''' ; k''') \\
\langle l' : k'' : \text{nil} \rangle & : \text{in}(foo)@l.\text{roll}(\gamma)
\end{align*}
\]

Performing \( \text{roll}(k) \) restores the initial net, by releasing the content of the target memory as well as the parallel in, which is generated by \( N_{\not\subseteq k} \) in rule (Roll).

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 11:** Let \( N \) be a net in normal form (see Lemma 6). If \( k \not\subseteq \bar{z} \) then:

\[
N_{\not\subseteq k} \equiv \bigoplus_{l \subseteq L'} (l : \bigoplus_{w \in W'} k'_w : \text{in}_{\gamma_w}(T_w)@l_w.P_w | \bigoplus_{y \in Y'} \bigoplus_{w \in W''} y_u : k'_w : \text{read}_{\gamma_y}(T_y)@l_y.P_y).
\]

where \( L' = \{ l \in L \mid k \not\subseteq l \} \), \( W' = \{ w \in W \mid k \not\subseteq w \} \), \( Y' = \{ y \in Y \mid k \not\subseteq y \} \) and \( W'' = \{ w \in W \mid k \not\subseteq w \} \).

We show now that \( \text{CR} \mu \text{KLAIM} \) is indeed a controlled version of \( \mu \text{KLAIM} \). Let \( \text{erCon} \) be a function from \( \text{CR} \mu \text{KLAIM} \) nets to \( \mu \text{KLAIM} \) nets which is the identity but for replacing \( \text{roll}(k) \) with \( \text{nil} \), and removing continuations inside memories for out, eval and newloc, and references \( \gamma \).

**Theorem 2:** Given a \( \text{CR} \mu \text{KLAIM} \) net \( N \), if \( N \rightleftharpoons M \) then \( \text{erCon}(N) \rightleftharpoons \text{erCon}(M) \) and if \( N \rightleftharpoons M \) then \( \text{erCon}(N) \rightleftharpoons \text{erCon}(M) \) where \( \rightleftharpoons \) is the transitive closure of \( \rightleftharpoons \).

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 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 \( \text{CR} \mu \text{KLAIM} \) as the net in Table XI, where:

\[
\begin{align*}
P_1 & = \text{in}(\text{chgPr})@\text{franchisor1}.\text{roll}(\gamma_1) \\
P_2 & = \text{in}(\text{chgPr})@\text{franchisor2}.\text{roll}(\gamma_2) \\
Q_1 & = \text{in}(\text{chgPr})@\text{franchisee1}.\text{roll}(\gamma_1) \\
Q_2 & = \text{in}(\text{chgPr})@\text{franchisee2}.\text{roll}(\gamma_1)
\end{align*}
\]

The market is a storage of tuples, representing lots of goods, of the form \( \langle \text{lot}, v, l \rangle \), 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 \( \text{price()} \). Then, it waits for a change price request (i.e., a tuple \( \langle \text{chgPr}() \rangle \)) to trigger the rollback of the executed activity (by means of the \( \text{roll} \) 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 \(\text{applyLocalPolicy}(\cdot, \cdot)\)). 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 per unit of goods, 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 \(i\)th and read actions.

If \(\text{franchisee}_1\) needs to change its lot of goods, a tuple (“\(\text{chgPr}\)”) is locally produced and, then, the rollback is triggered by \(\text{roll}(k_0)\). In this way, the memory of the first \(\text{in}\) will be directly restored (and its forward history deleted), rather than undoing action-by-action the forward execution (as in \(\mu\text{KLAIM}\)). As expected, the backward step does not affect \(\text{franchisee}_2\). Instead, if \(\text{franchisor}_1\) wants to change the price stored in \(k'_0\) (“\(\text{suggPrice}\)”), it undoes action \(k_0 : \text{out}(\text{“suggPrice”, price()}, \cdot)\) and by involving the read memories within \(\text{franchisee}_1\) and \(\text{franchisee}_2\), because \(k'_0\) occurs within them, and all the occurrences of \(k'_0\) must be considered to apply rule (Roll). Thus, the projection operation \(\cdot \mid k_0\) 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 \(\text{read}(\text{“suggPrice”}, !x_{pr})@u_r, \text{by in}(\text{“suggPrice”}, !x_{pr})@u_f, \text{out}(\text{“suggPrice”}, x_{pr})@u_r\), 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 franchisors.

It is worth noticing that, by setting processes \(P_i\) and \(Q_i\) to \(\text{nil}\) and removing all references, we obtain a \(\mu\text{KLAIM}\) 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 [15], [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 [17]. 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 \(\pi\)-calculus [4], while reduction semantics has been studied for \(\text{HOr}\) [13] and \(\mu\text{Oz}\) [16]. We are closer to [13], 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 [18], where a non-reversible controller drives a reversible process. For an exhaustive survey on causal-consistent reversibility we refer to [14].
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 [12], 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 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


APPENDIX A
PROOFS OF SECTION III

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

**Proof (sketch).** We prove both (i) and (ii) by induction on the reduction steps performed to obtain $N$ from an initial net. The existence of such a net is guaranteed by Definition 3. If $N$ is the initial net, then the result trivially holds. Otherwise, the proof proceeds by case analysis on the last reduction rule applied.

LEMMA 2. Let $N$ and $M$ be two $\textit{rµKLAIM}$ nets such that $N \xrightarrow{r} M$. Then $\textit{erM}(N) \rightarrow \textit{erM}(M)$.

**Proof (sketch).** Straightforward, by first proving by induction on the derivation of $N \equiv M$ that $N \equiv M' \Rightarrow \textit{erM}(N) \equiv \textit{erM}(M')$, and then by observing that forward $\textit{rµKLAIM}$ rules are just decorated versions of $\mu\textit{KLAIM}$ rules, and decorations are removed by function $\textit{erM}$.

LEMMA 3. Let $R$ and $S$ be two $\mu\textit{KLAIM}$ nets such that $R \rightarrow S$. Then for all $\textit{rµKLAIM}$ nets $M$ such that $\textit{erM}(M) = R$ there exists a $\textit{rµKLAIM}$ net $N$ such that $M \rightarrow R$ and $\textit{erM}(N) \equiv S$.

**Proof (sketch).** Nets such that $\textit{erM}(M) = R$ have the same localities, processes and tuples as $R$. In addition, 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 ($\textit{Split}$ and $\textit{Ext}$). In most of the cases, the result of executing the action is a net $N$ such that $\textit{erM}(N) = S$. However, $\textit{erM}(N)$ may differ from $S$ because $\textit{nil}$ processes cannot be garbage collected in $\textit{rµKLAIM}$, thus some more $\textit{nil}$ processes may occur in $\textit{erM}(N)$. In such cases, anyway, the extra $\textit{nil}$ processes can be removed by applying the $\mu\textit{KLAIM}$ structural congruence.

LEMMA 4 (LOOP LEMMA). For all well-formed $\textit{rµKLAIM}$ nets $N$ and $M$, the following holds:

$$N \xrightarrow{r} M \iff M \xrightarrow{r} N.$$ 

**Proof (sketch).** The proof is by induction on the derivation of $N \xrightarrow{r} M$ 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 $M \xrightarrow{r} N$ for the only if direction. One has to pay attention in rules ($\textit{InRev}$) and ($\textit{ReadRev}$) that the process $Q$ is indeed the instantiation of the stored continuation $P$ with the substitution $\sigma$ resulting from the pattern matching. This always holds for well-formed nets.

LEMMA 5 (SQUARE LEMMA). If $\delta_1 = M \overset{n_1}{\rightarrow} N_1$ and $\delta_2 = M \overset{n_2}{\rightarrow} N_2$ are two coinitial concurrent transitions, then there exist two cofinal transitions $\delta_2/\delta_1 = N_1 \overset{n_2}{\rightarrow} N$ and $\delta_1/\delta_2 = N_2 \overset{n_1}{\rightarrow} N$.

**Proof:** By case analysis on the form of transitions $\delta_1$ and $\delta_2$.

Both $\delta_1$ and $\delta_2$ forward: $\delta_1$ and $\delta_2$ can be any combination of forward reductions, namely we have 15 subcases (($\textit{In}$) and ($\textit{In}$), ($\textit{In}$) and ($\textit{Out}$), ($\textit{In}$) and ($\textit{Read}$), ($\textit{In}$) and ($\textit{Eval}$), ($\textit{In}$) and ($\textit{New}$), ($\textit{Out}$) and ($\textit{Out}$) and ($\textit{Read}$), ($\textit{Out}$) and ($\textit{New}$), ($\textit{Read}$) and ($\textit{Read}$), ($\textit{Read}$) and ($\textit{Eval}$), and ($\textit{New}$) and ($\textit{Eval}$), and ($\textit{Eval}$) and ($\textit{New}$)). The most interesting cases are those concerning the interplay of different reads on the same tuple.

Let us consider the case of two reads on the same tuple.

$$M = l : k : \text{read}(T)@l'P$$

$$\| l' : k' : \langle et \rangle \| l'' : k'' : \text{read}(T')@l''P' \| M'$$

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

$$N_1 = (\nu k_1) \{ l :: k_1 : P_\sigma | [k : \text{read}(T)@l'_P ; k' ; k_1] \}
\| l' :: k' : \langle et \rangle \| l'' :: k'' : \text{read}(T')@l''P' \| M'$$

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

$$N = (\nu k_1) \{ l :: k_1 : P_\sigma | [k : \text{read}(T)@l'_P ; k' ; k_1] \}
\| l' :: k' : \langle et \rangle \| l'' :: k'' : \text{read}(T')@l''P' \| M'$$

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

$$M = l : k : \text{in}(T)@l'P$$

$$\| l' : k' : \langle et \rangle \| l'' : k'' : \text{read}(T')@l''P' \| l''' : k''' : \langle et' \rangle \| M'$$

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

$$N_1 = (\nu k_1) \{ l :: k_1 : P_\sigma | [k : \text{in}(T)@l'_P ; k' ; \langle et \rangle ; k_1] \}
\| l' :: \text{empty} \| l'' :: k'' : \text{read}(T')@l''P' \| l''' :: k''' : \langle et' \rangle \| M'$$

where $\sigma = \text{match}([T], et)$, and $M \rightarrow N_2$ with:

$$N_2 = l : k : \text{in}(T)@l'P \| l' :: k' : \langle et \rangle \| (\nu k_2) \{ l'' :: k_2 : P_\sigma' | [k'' : \text{read}(T')@l''P' ; k''' ; k_2] \}
\| l''' :: k''' : \langle et' \rangle \| M'$$

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

$$N = (\nu k_1) \{ l :: k_1 : P_\sigma | [k : \text{in}(T)@l'_P ; k' ; \langle et \rangle ; k_1] \}
\| l' :: \text{empty} \| (\nu k_2) \{ l'' :: k_2 : P_\sigma' \| [k'' : \text{read}(T')@l''P' ; k''' ; k_2] \}
\| l''' :: k''' : \langle et' \rangle \| M'$$

Both $\delta_1$ and $\delta_2$ backward: $\delta_1$ and $\delta_2$ can be any combination of backward reductions, namely we have 15 subcases (($\textit{InRev}$)
Thus, both pairwise distinct. Then

\[ M \equiv l :: k_0' : Q \mid [k : \text{read}(T)@l'.P; k'; k''] \]

\[ \parallel \parallel l' :: k' : \langle \text{et} \rangle \]

\[ \parallel \parallel l_1 :: k_1' : Q_1 \mid [k_1 : \text{read}(T)@l'.P_1; k'; k_1''] \]

\[ \parallel M' \]

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

\[ N_1 \equiv l :: k : \text{read}(T)@l'.P \parallel l' :: k' : \langle \text{et} \rangle \]

\[ \parallel l_1 :: k_1' : Q_1 \mid [k_1 : \text{read}(T)@l'.P_1; k'; k_1''] \]

\[ \parallel M' \]

and \( M \sim_r N_2 \) with:

\[ N_2 \equiv l :: k'' : Q \mid [k : \text{read}(T)@l'.P; k'; k''] \]

\[ \parallel l' :: k' : \langle \text{et} \rangle \parallel l_1 :: k_1 : \text{read}(T)@l'.P_1 \parallel M' \]

\[ \delta_1 \text{ forward and } \delta_2 \text{ backward}: \] We have 25 subcases, due to the combination of any forward reduction with any backward reduction. The most interesting cases are those concerning the interplay of different reads on the same tuple.

Let us consider the case of a read on a tuple and an undo of a read on the same tuple.

\[ M \equiv l :: k : \text{read}(T)@l'.P \parallel l' :: k' : \langle \text{et} \rangle \parallel (\nu k_2) (l'' :: k_2 : Q \mid k'' : \text{read}(T'@l'.P'; k''; k_2)) \parallel M' \]

Since \( M \) is well formed, by Lemma 1, \( k, k', k'', k_1, k_2 \) are pairwise distinct. Then \( M \sim_r N_1 \) with:

\[ N_1 \equiv (\nu k_1) (l :: k_1 : P\sigma \mid [k : \text{read}(T)@l'.P; k'; k_1]) \]

\[ l' :: k' : \langle \text{et} \rangle \parallel (\nu k_2) (l'' :: k_2 : Q \mid k'' : \text{read}(T'@l'.P'; k''; k_2)) \parallel M' \]

where \( \sigma = \text{match}([T], \text{et}) \), and \( M \sim_r N_2 \) with:

\[ N_2 \equiv l :: k : \text{read}(T)@l'.P \parallel l' :: k' : \langle \text{et} \rangle \parallel l'' :: k'' : \text{read}(T)@l'.P' \parallel M' \]

Thus, both \( N_1 \) and \( N_2 \) evolve to:

\[ N \equiv (\nu k_1) (l :: k_1 : P\sigma \mid l :: k : \text{read}(T)@l'.P; k'; k_1) \]

\[ l' :: k' : \langle \text{et} \rangle \parallel l'' :: k' : \text{read}(T)@l'.P' \parallel M' \]

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

Lemma 7 (Rearranging lemma): Let \( \theta \) be a trace. There exist forward traces \( \theta' \) and \( \theta'' \) such that \( \theta \sim \theta'_\ast \cdot \theta'' \).

Proof: The proof is by lexicographic induction on the length of \( \theta \) and on the distance between the beginning of \( \theta \) and the earliest pair of transitions in \( \theta \) of the form \( \delta ; \delta' \), contradicting the property. If there is no such pair, then we are done. If there is one, we have two possibilities:

- \( \delta \) and \( \delta' \) 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;

- \( \delta \) and \( \delta' \) are in conflict: Let \( \eta_1 \) and \( \eta_2 \) be the forward/backward memories of \( \delta \) and \( \delta' \), respectively. By Definition 7, \( \delta \) and \( \delta' \) are coinitial; let \( M \) be their source and \( N_1 \) and \( N_2 \) their targets, respectively. We have the following possibilities:

  - \( z \in \text{closure}_{M || N_1}(\lambda(\eta_1)) \) and \( z \in \text{closure}_{M || N_2}(\lambda(\eta_2)) \): In this case \( \delta = \delta' \) and then, by applying Lemma 4, we remove \( \delta;\delta' \). Indeed, if they would be 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 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 is not possible because this would mean that the forward in was done before the forward out. If they would be different transitions sharing a locality \( l \), this would mean that two newloc creates the same locality, but this is not possible since names of created localities are bound.

  - \( x(z) \in \lambda(\eta_1) \) and \( x(z) \in \lambda(\eta_2) \): If \( z \) is a key \( k \) then a forward read on a tuple is followed by a backward out on the same tuple: this is not possible because you have to undo the read before undoing the out. The case of a forward read on a 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.

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

\[ \square \]

Lemma 8 (Shortening lemma): Let \( \theta_1 \) and \( \theta_2 \) be coinitial and cofinal traces with \( \theta_2 \) forward. Then, there exists a forward trace \( \theta'_1 \) of length at most that of \( \theta_1 \) s.t. \( \theta'_1 \prec \theta_1 \).

\[ \square \]
Proof: The proof is by induction on the length of $\theta_1$. If $\theta_1$ is forward we are done. Otherwise, by Lemma 7, we can write $\theta_1 = \theta_1; \theta''$, with $\theta$ and $\theta''$ forward. Let $\delta_\theta; \theta''$ be the only two successive transitions in $\theta_1$ with opposite directions with $\mu_1$ belonging to $\delta_\theta$. Since $\mu_1$ is removed by $\delta_\theta$, then $\mu_1$ has to be put back by another forward transition otherwise this difference will stay visible since $\theta_2$ is a forward trace. Let $\delta_1$ be the earliest such transition in $\theta_1$. Since it is able to put back $\mu_1$ it has to be the exact opposite of $\delta_\theta$, so $\delta_1 = \delta$. Now we can swap $\delta_1$ with all the transitions between $\delta_1$ and $\delta_\theta$, in order to obtain a trace in which $\delta_1$ and $\delta_\theta$ 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 $\mu_2$ which is not concurrent to $\delta_1$. Thanks to Lemma 1 (completeness of well-formed nets) and since locality names are fresh, the only possible conflict may be between a $r(z)$ in $\lambda(\mu_1)$ and a $z \in \text{closure}_N(\lambda(\mu_2))$, for an appropriate $N$, or vice versa. A $r(k)$ in $\lambda(\mu_1)$ means $\delta_1$ is a forward read which has some conflicts with an $\text{out}$ or an $\text{in}$ occurring between the previous undo of the read and $\delta_1$. Anyway, it is not possible to have an $\text{out}$ of a tuple (an undo of a read ($\delta_\theta$). It is also not possible to have an $\text{in}$ before a read ($\delta_1 = \delta$). A $r(l)$ in $\lambda(\mu_1)$ 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 $k \in \text{closure}_N(\lambda(\mu_1))$, for an appropriate $N'$, and a $r(k)$ in $\lambda(\mu_2)$ means $\delta_2$ is the undo of an $\text{in}$ (or the undo of an $\text{out}$) and $\delta_2$ is a read. This is impossible because this would mean that a previous $\text{in}$ has happened before the undo of read. The undo of an $\text{out}$ 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 $\theta_1$ and $\theta_2$ be coinitial traces, then $\theta_1 \simeq \theta_2$ if and only if $\theta_1$ and $\theta_2$ are cofinal.

Proof: By construction of $\simeq$, if $\theta_1 \simeq \theta_2$ then $\theta_1$ and $\theta_2$ must be coinitial and cofinal, so this direction of the theorem is verified. Now we have to prove that $\theta_1$ and $\theta_2$ being coinitial and cofinal implies that $\theta_1 \simeq \theta_2$. By Lemma 7 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 $\theta_1$ and $\theta_2$ and on the distance between the end of $\theta_1$ and the earliest pair of transitions $\delta_1$ in $\theta_1$ and $\delta_2$ in $\theta_2$ 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.

- **$\delta_1$ forward and $\delta_2$ backward:** we have $\theta_1 = \theta_1; \delta_1; \theta''$ and $\theta_2 = \theta_1; \delta_2; \theta''$. Moreover we know that $\delta_1; \theta''$ is a forward trace, so we can apply the Lemma 8 to the traces $\delta_1; \theta''$ and $\delta_2; \theta''$ (since $\theta_1$ and $\theta_2$ 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 $\theta_2$ has a shorter equivalent forward trace. We can conclude by induction.

- **$\delta_1$ and $\delta_2$ forward:** by assumption, the two transitions are different. If they are not concurrent then they should conflict on a thread process $k : P$ that they both consume and store in different memories, or a tuple $k : (et)$ 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 $\delta_2$ in $\theta_2$ creating the same memory as $\delta_1$. However, no other process $k : P$ (nor tuple $k : (et)$) is ever created in $\theta_2$ thus this is not possible. The conflict cannot be on a locality either, since the locality name is bound. So we can assume that $\delta_1$ and $\delta_2$ are concurrent. Again let $\delta_2$ be the transition in $\theta_2$ creating the same memory of $\delta_1$. We have to prove that $\delta_2$ is concurrent to all the previous transitions. This holds since no previous transition can remove one of the processes needed for triggering $\delta_2$ and since forward transitions can never conflict on $k$ or $l$. Thus we can repetitively apply Lemma 5 to derive a trace equivalent to $\theta_2$ where $\delta_2$ and $\delta_2'$ are consecutive. We can apply the same transformation to $\theta_1$. Now we can apply Lemma 5 to $\delta_1$ and $\delta_2$ 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.

- **$\delta_1$ and $\delta_2$ backward:** $\delta_1$ and $\delta_2$ cannot remove the same memory. Let $\mu_{\theta_1}$ be the memory removed by $\delta_1$. Since the two traces are cofinal, either there is another transition in $\theta_1$ putting back the memory or there is a transition $\delta_1'$ in $\theta_2$ removing the same memory. In the first case, $\delta_1$ 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 $\theta_2$). 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 $\delta_1; \delta_2 \simeq \epsilon_{\text{target}}(\delta)$ (see definition of $\simeq$) to remove the two transitions, thus shortening the trace. The thesis follows by inductive hypothesis.

Appendix B

Proofs of Section IV

Lemma 6 (Net Normal Form). For any CRUMLAIM net $N$, we have:

$$N \equiv (\nu z) \bigg| \bigg( \nu l \bigg| \bigg( \bigwedge_{i \in I} (k_i : P_i) \bigg) \bigg| \bigg( \bigwedge_{j \in J} [k_j : a_j, P_j, \xi_j] \bigg) \bigg| \bigg( \bigwedge_{k \in K} [k_h \prec (k_{H}, k_{H})] \bigg) \bigg| \bigg( \bigwedge_{z \in Z} (k_z : (et_z)) \bigg) \bigg| \bigg( \bigwedge_{w \in W} [k_{w} : \text{in}_{\tau_w}(T_w)@l_{w}, P_{w}, k_{w}^{2} : (t_w) ; k_{w}^{3}] \bigg) \bigg| \bigg( \bigwedge_{y \in Y} [k_{y}^{1} : \text{read}_{\gamma_y}(T_{y})@l_{y}, P_{y}, k_{y}^{2}, k_{y}^{3}] \bigg) \bigg)$$

where $a_j$ is neither in nor read for every $j \in J$. 
**Proof (sketch).** Trivial, using structural congruence.

**Theorem 2** Given a CRµKaim net \( N \), if \( N \rightarrow_c M \) then \( \text{erCon}(N) \rightarrow_r \text{erCon}(M) \) and if \( N \sim_c M \) then \( \text{erCon}(N) \sim^+_r \text{erCon}(M) \) where \( \sim^+_r \) is the transitive closure of \( \sim_r \).

**Proof (sketch).** The forward case trivially holds. For the backward case we reason by induction on the number \( n \) of memories deleted by the step \( N \sim_c M \). The base case is when \( n = 1 \). Since \( N \) is complete and \( k \)-dependent and since there exists exactly one memory, the continuation of the memory is enabled and the process roll is part of it. Then, the process \( \text{erCon}(N) \) can directly remove this memory and we have \( \text{erCon}(N) \rightarrow_r \text{erCon}(M) \). The correctness of this step can be proved by case analysis on the action \( a \) in the memory.

For the inductive case \( (n > 1) \), note that memories and processes can be seen as a tree, according to the ordering among their keys (for memories, one should consider their head). In this tree, processes are leaves. For the inductive step, we take one process \( P \) in the tree rooted in the memory targeted by the roll. We have to distinguish two cases: either the roll enabling the backward step is part of \( P \), or it is not.

Let us consider the second case. Reasoning as for the base case, \( \text{erCon}(N) \rightarrow_r \text{erCon}(N^-) \) where \( N^- \) does not contain the memory which was the father of \( P \). Since the \( N \) is complete and \( k \)-dependent, one can prove that also \( N^- \) enjoys the same properties and then \( N^- \sim_c M \). By inductive hypothesis we have that \( \text{erCon}(N^-) \sim^+_r \text{erCon}(M) \), and since \( \text{erCon}(N) \rightarrow_r \text{erCon}(N^-) \) we are done. In the first case, one cannot directly apply the same approach, since in \( N^- \) the roll triggering the controlled backward step would not be enabled any more. However, a controlled backward step reaching the same network \( M \) could be triggered by putting the same roll in parallel with the process \( Q \) that was inside the memory. Thus, we consider a net \( N^+ \) which is the same as \( N \) but where the roll is now in parallel with \( Q \). Since \( N \) is \( k \)-dependent, the new key of the roll will be dependent on \( k \) as desired. We then have that \( \text{erCon}(N^+) \rightarrow_r \text{erCon}(N^-) \). Since \( N^- \) is complete and \( k \)-dependent too, we have that \( N^- \sim_c M \), and by applying the inductive hypothesis we also have \( \text{erCon}(N^-) \sim^+_r \text{erCon}(M) \), as desired. ■