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], [18], \(\pi\)-calculus [4], higher-order \(\pi\) [14] and \(\mu\text{Oz}\) [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 \(\mu\text{KLAIM}\) [11], 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 \textit{out} and in respectively insert tuples into and remove them from tuple spaces. Primitives \textit{eval}, to execute a process on a possibly remote location, and \textit{newloc}, creating a new location, support distribution. Finally, \(\mu\text{KLAIM}\) features the primitive \textit{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 \textit{in} and \textit{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 \(\mu\text{KLAIM}\) (\(R\mu\text{KLAIM}\) 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 \textit{roll} operator. This operator undoes a given past action, and all its consequences, but it does not affect independent actions. The \textit{roll} operator is based on the uncontrolled reversibility mechanism, but it is much more suitable to exploit reversibility for programming actual applications. We put \(CR\mu\text{KLAIM}\) 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
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) $N ::= \emptyset \mid l :: C \mid N_1 \parallel N_2 \mid (vl)N$ |
| (Components) $C ::= (et) \mid P \mid C_1 | C_2$ |
| (Processes) $P ::= \text{nil} \mid a.P \mid P_1 | P_2 \mid A$ |
| (Actions) $a ::= \text{out}(t)@l \mid \text{eval}(P)@l$ |
| (Tuples) $t ::= e \mid \ell \mid t_1, t_2$ |
| (Evaluated tuples) $et ::= v \mid l \mid et_1, et_2$ |
| (Templates) $T ::= e \mid \ell \mid !x \mid !u \mid T_1, T_2$ |

### TABLE I. µKLAIM SYNTAX

| (Monoid) $N \parallel N \equiv N \mid N_1 \parallel N_2 \equiv N_2 \parallel N_1$ |
| (RCom) $(vl_1)(vl_2)N \equiv (vl_2)(vl_1)N$ |
| (PDef) $l :: A \equiv l :: P$ if $A \triangleq P$ |
| (Ext) $N_1 \parallel (vl_1)N_2 \equiv (vl_2)(vl_1)N$ if $l \notin \text{fn}(N_1)$ |
| (Alpha) $N \equiv N'$ if $N =_\alpha N'$ |
| (Abs) $l :: C \equiv (C)\text{nil}$ |
| (Clone) $l :: C_1\ | C_2 \equiv l :: C_1 \parallel l :: C_2$ |

### TABLE II. µKLAIM STRUCTURAL CONGRUENCE

C is the hosted component. In the net $(vl)N$, the scope of the name $l$ is restricted to $N$. $\emptyset$ denotes the empty net.

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

Actions $\text{out}$, $\text{in}$ and $\text{read}$ add/withdraw/access data from repositories. Action eval activates a new thread of execution in a (possibly remote) node, and $\text{newloc}$ creates new nodes. All actions but $\text{newloc}$ indicate their target locality. Actions $\text{in}$ and $\text{read}$ 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: $\text{newloc}(l).P$ binds name $l$ in $P$, and $(vl)N$ binds $l$ in $N$. Prefixes $\text{in} \{l_1, l_2, \ldots \}@P$ and $\text{read} \{l_1, l_2, \ldots \}@P$ bind variable $l$ in $P$. A locality/variable that is not bound is called free. The set $\text{fn}(\cdot)$ of free names of a term is defined accordingly. As usual, we say that two terms are $\alpha$-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 $\text{nil}$ is the identity for $\cdot | \cdot$. 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 $\cdot | \cdot$). The other laws are standard.

We define the auxiliary pattern-matching function $\text{match}(\cdot, \cdot)$ 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 $\text{match}(T, t)$ succeeds, it returns a substitution for the variables in $T$; otherwise, it is undefined. A substitution $\sigma$ 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 $\sigma$ to denote substitution composition and $e$ to denote the empty substitution.

We assume the existence of a function $\llbracket \cdot \rrbracket$ 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 $(et)$ are stored in tuple spaces.

We say a relation $R$ is evaluation closed if it is closed under active contexts, i.e. $N_1 R N_2$ implies $(N_1 \parallel N_2) R (N'_1 \parallel N'_2)$. 
match(T₁, t₁) = σ₁  
match(T₂, t₂) = σ₂

\[\text{match}(T₁, T₂), (t₁, t₂) = \sigma₁ \circ \sigma₂\]

\[\text{match}(v, v) = \epsilon \quad \text{match}(\langle x, v \rangle) = [v/x]\]

\[\text{match}(l, l) = \epsilon \quad \text{match}(\langle u, l \rangle) = [l/u]\]

**TABLE III.** \(\mu\text{CLAIM MATCHING RULES}\)

\[
\begin{align*}
[t] &= \epsilon \\
1 : \text{out}(t) &@\tilde{t}.P \mid \tilde{t} : \text{nil} \mapsto 1 : P \parallel \tilde{t}' : (\epsilon t) &\quad \text{(Out)} \\
\text{match}(T], et) &= \sigma \\
1 : \text{in}(T) &@\tilde{t}.P \mid \tilde{t}' : (\epsilon t) \mapsto 1 : P\sigma \parallel \tilde{t}' : \text{nil} &\quad \text{(In)} \\
\text{match}(T], et) &= \sigma \\
1 : \text{read}(T) &@\tilde{t}'.P \mid \tilde{t}' : (\epsilon t) \mapsto 1 : P\sigma \parallel \tilde{t}' : (\epsilon t) &\quad \text{(Read)} \\
1 : \text{newloc}(\tilde{t}).P &\mapsto (\tilde{t}')(l : P \parallel \tilde{t} : \text{nil}) &\quad \text{(New)} \\
1 : \text{eval}(Q) &@\tilde{t}'.P \mid \tilde{t}' : \text{nil} \mapsto 1 : P \parallel \tilde{t}' : Q &\quad \text{(Eval)}
\end{align*}
\]

**TABLE IV.** \(\mu\text{CLAIM OPERATIONAL SEMANTICS}\)

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 \(\alpha\)-conversion are defined as usual, and from now on \(f_o(N)\) also includes free keys), and by only considering well-formed nets.

**Definition 1 (Initial and well-formed nets):** A \(\mu\text{CLAIM}\) net is initial if it has no memories, no connectors, and all its keys are distinct. A \(\mu\text{CLAIM}\) 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 memory 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 \(\text{in}\) and \(\text{read}\) 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 \(\text{out}\) memory stores the key \(k''\) of the created tuple, the \(\text{eval}'\)'s one the key \(k''\) of the spawned process, and the \(\text{in}'\)'s one the consumed tuple \(h : (\epsilon t)\). The memory for \(\text{read}\) 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 the process tagged by \(k_1\). Finally, we distinguish empty localities, \(l : \text{empty}\), containing no information, from localities \(l : k : \text{nil}\) containing a \(\text{nil}\) process with its key \(k\), which may interact with a memory to perform a backward action.

**Operational semantics.** Structural congruence for \(\rho\mu\text{CLAIM}\) extends the one for \(\mu\text{CLAIM}\) 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 (\(\rho\mu\text{CLAIM}\) semantics):** The operational semantics of \(\rho\mu\text{CLAIM}\) 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 \(\mu\text{CLAIM}\) 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

\[N := 0 \mid l : C \mid l : \text{empty} \mid N_1 || N_2 \mid (\nu z)N\]

\[C := k : (et) \mid k : P \mid C₁ || C₂ \mid \mu \mid k₁ \prec (k₂, k₃)\]

\[\mu := [k : \text{out}(t)@\tilde{t}; k'' ; k'] \mid [k : \text{in}(T)@\tilde{t}; P; h : (\epsilon t); k'] \mid [k : \text{read}(T)@\tilde{t}; P; h ; k'] \mid [k : \text{newloc}(l); k'] \mid [k : \text{eval}(Q)@\tilde{t}; k'' ; k']\]

**TABLE V.** \(\rho\mu\text{CLAIM SYNTAX}\)

\(^1\)One may look for more compact ways to store history information. This issue is considered for reversible \(\mu\text{Oz}\) in [17], but it is out of the scope of the present paper.
(RCom) \((\nu e_1)(\nu e_2)N \equiv (\nu e_2)(\nu e_1)N\)  
(Prefl) \(l : k : A \equiv l : k : P \text{ if } A \models P\)  
(Ext) \(N_1 \equiv (\nu z)N_2 \equiv (\nu z)(N_1 \parallel N_2) \text{ if } z \notin \text{fn}(N_1)\)  
(Abs) \(l : C \equiv l : C \parallel l : \text{empty}\)  
(Garb) \((\nu k)0 \equiv 0\)  
(Split) \(l : k : P \mid Q \equiv (\nu k_1, k_2)l : k : (k_1, k_2) \mid k_1 : P \mid k_2 : Q\)  

<table>
<thead>
<tr>
<th>Table VI.</th>
<th>(\text{(R\mu\text{CLAIM structural congruence})})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\vdash t = \text{et})</td>
<td>(\rightarrow (\nu k', k'') (l : k : P \mid [k : \text{out}(t)@l'; k''] \parallel l' : k'' : (\text{et})))</td>
</tr>
<tr>
<td>(\vdash t : k : \text{in}(T)@l', P \parallel l' : h : (\text{et}))</td>
<td>(\rightarrow r (\nu k') l : k' : P' \parallel [k : \text{in}(T)@l'; h : (\text{et})]; l' \parallel l' : \text{empty})</td>
</tr>
<tr>
<td>(\vdash t : k : \text{newloc}(l'), P \parallel l' : \text{empty})</td>
<td>(\rightarrow r (\nu l') (\nu (k') l : k' : P \parallel [k : \text{newloc}(l')]; l') \parallel l' : \text{empty})</td>
</tr>
<tr>
<td>(\vdash t : k : \text{eval}(Q)@l', P \parallel l' : \text{empty})</td>
<td>(\rightarrow r (\nu k', k'') (l : k' : P \mid [k : \text{eval}(Q)@l'; k''] \parallel l' : k'' : Q))</td>
</tr>
<tr>
<td>(\vdash t : k : \text{out}(t)@l', P \parallel l' : \text{empty})</td>
<td>(\rightarrow r (\nu k') l : k' : P \mid [k : \text{out}(t)@l'; k''] \parallel l' : k'' : (\text{et}))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table VII.</th>
<th>(\text{(R\mu\text{CLAIM operational semantics})})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\vdash t : k : \text{in}(T)@l', P \parallel l' : h : (\text{et}))</td>
<td>(\rightarrow r (\nu k') l : k' : P' \parallel [k : \text{in}(T)@l'; h : (\text{et})]; l' \parallel l' : \text{empty})</td>
</tr>
<tr>
<td>(\vdash t : k : \text{newloc}(l'), P \parallel l' : \text{empty})</td>
<td>(\rightarrow r l : k : \text{newloc}(l'), P)</td>
</tr>
<tr>
<td>(\vdash t : k : \text{eval}(Q)@l', P \parallel l' : \text{empty})</td>
<td>(\rightarrow r l : k : \text{eval}(Q)@l', P \parallel l' : \text{empty})</td>
</tr>
</tbody>
</table>

retrieval, for a term of type \(P\) to be generated by rule \((\nu k)\), \(k\) must be unique in the term. This means that a tuple \(k\) which contains \(k\) as a key in the tuple \(k\) is not allowed to occur elsewhere in the term. In rule \((\nu k)\), \(k\) labels the spawned process \(Q\). No restriction on \(k\) is required in rule \((\nu k\text{'}\)\), since \(k\) cannot occur elsewhere in the term. In rule \((\nu k\text{'}\)\) only the key is needed since the tuple is still in the term, and its key is unchanged. Rule \((\nu k\text{'}\)\) creates a new, empty, locality. In rule \((\nu k\text{'}\)\) 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 \(R\mu\text{CLAIM read}\) 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 \(R\mu\text{CLAIM}\) by using an in followed by an out.

Consider a \(R\mu\text{CLAIM}\) net \(N\) with three nodes, \(l_1\) hosting a tuple \((\text{foo},\text{bar})\), and \(l_2\) and \(l_3\) hosting processes willing to access such tuple:

\[\begin{align*}
N' & = l_1 :: k_1 : (\text{foo}) \parallel l_2 :: k_2 : \text{in}(\text{foo})@l_1, \text{out}(\text{foo})@l_1, P \\
& \parallel l_3 :: k_3 : \text{in}(\text{foo})@l_1, \text{out}(\text{foo})@l_1, P'.
\end{align*}\]

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:

\[\begin{align*}
& (\nu k_1, k_2)l_1 :: k'' : (\text{foo}) \parallel l_2 :: k_1 : \text{in}(\text{foo})@l_1, \text{out}(\text{foo})@l_1, P; k_2 : (\text{foo}) \parallel k_3 : \text{out}(\text{foo})@l_1, k'' : (\text{foo}) \parallel l_3 :: k'' : P'; k_1 : \text{in}(\text{foo})@l_1, P; k_2 : \text{out}(\text{foo})@l_1, k'' : (\text{foo}) \parallel k_3 : \text{out}(\text{foo})@l_1, k'' : (\text{foo})
\end{align*}\]

Now, the process in \(l_2\) cannot immediately perform a backward step, since it needs the tuple \(k'' : (\text{foo})\) in \(l_1\), while only \(k'' : (\text{foo})\) 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 \(\text{proc} [14]\) or an output process in [5], [18], [4].

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

\[\begin{align*}
N = l_1 :: k_1 : (\text{foo}) \parallel l_2 :: k_2 : \text{read}(\text{foo})@l_1, P \\
& \parallel l_3 :: k_3 : \text{read}(\text{foo})@l_1, P'.
\end{align*}\]
By executing the two read actions (the order is now irrelevant), the net $N$ evolves to:

$$
(\nu k_2', k_3') (l_1 :: k_3 : \{ \text{foo} \}) \\
| l_2 :: k_3 : P | (k_2 : \text{read}(\text{foo}) @ l_1, P; k_1; k_2) \\
| l_3 :: k_3 : P' | (k_3 : \text{read}(\text{foo}) @ l_1, P'; k_1; k_2')
$$

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:

$$
(\nu k_2', k_3') (l_1 :: k_3 : \{ \text{foo} \}) \\
| l_2 :: k_3 : P | (k_2 : \text{read}(\text{foo}) @ l_1, P) \\
| l_3 :: k_3 : P' | (k_3 : \text{read}(\text{foo}) @ l_1, P'; k_1; k_2')
$$

**Basic properties.** We now show that $R_{\mu} L A I M$ respects the $\mu L A I M$ 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 $[k : \text{out}(t)@l; k'; k'']$, $[k : \text{in}(T)@l; P; k' : \{ \tau \}; k']$, $[k : \text{read}(\text{foo})@l_1, P; h; k']$, $[k : \text{eval}(Q)@l; k''; k']$, and $[k : \text{newloc}(l); k']$, and connectors of the shape $k \prec (k'; k'')$, we call $k$ the head of those memories/connector, and $k'$ and $k''$ 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 $\text{complete}(N)$, 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 $k : \{ \tau \}$, 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 $[k : \text{newloc}(l); k']$ 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 $R_{\mu} L A I M$ net we can derive a $\mu L A I M$ net by removing history and causality information. This is formalized by function $erH$ (and the auxiliary function $erC$ for components) defined in Table VIII. The following lemmas state the correspondence between $R_{\mu} L A I M$ forward semantics and $\mu L A I M$ semantics.

**Lemma 2:** Let $N$ and $M$ be two $R_{\mu} L A I M$ nets such that $N \rightarrow_{\nu} M$. Then $erH(N) \rightarrow_{\nu} erH(M)$.

**Lemma 3:** Let $R$ and $S$ be two $\mu L A I M$ nets such that $R \rightarrow_{\nu} S$. Then for all $R_{\mu} L A I M$ nets $M$ such that $erH(M) = R$ there exists a $R_{\mu} L A I M$ net $N$ such that $M \rightarrow_{\nu} N$ and $erH(N) \equiv S$.

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

**Lemma 4 (Loop lemma):** For all well-formed $R_{\mu} L A I M$ nets $N$ and $M$, the following holds: $N \rightarrow_{\nu} M \iff M \rightarrow_{\nu} N$.

We now move to the proof that $R_{\mu} L A I M$ is indeed causally consistent.

**In a forward reduction $N \rightarrow_{\mu} M$ we call forward memory the memory $\mu$ which does not occur in $N$ and occurs in $M$. Similarly, in a backward reduction $N \rightarrow_{\nu} M$ we call backward memory the memory $\mu$ which occurs in $N$ and does not occur in $M$. We call transition a triplet of the form $N \rightarrow_{\nu} M$, or $N \rightarrow_{\mu} M$, where $N, M$ are well-formed nets, and $\mu$ 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 $\mu_{\rightarrow_{\nu}}$ and $\mu_{\rightarrow_{\mu}}$. If $\eta = \mu_{\rightarrow_{\nu}}$, then $\eta = \mu_{\rightarrow_{\mu}}$, 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 $h \prec (k_1, k_2)$, 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 closure below). Two transitions are coinital 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 $\delta'$ denotes its inverse. Notions of source, target and compositibility extend naturally to traces. We denote with $e_M$ 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 $\lambda(\mu_{\rightarrow_{\nu}})$ of a memory $\mu$ is:

$$
\lambda([k : \text{out}(t)@l; k'; k'']) = \{ k, k', k'', \tau(l) \} \\
\lambda([k : \text{in}(T)@l; P; k' : \{ \tau \}; k']) = \{ k, k', k'', \tau(l) \} \\
\lambda([k : \text{read}(\text{foo})@l_1, P; h; k']) = \{ k, k', k'', \tau(l) \} \\
\lambda([k : \text{eval}(Q)@l; k''; k']) = \{ k, k', k'' \} \\
\lambda([k : \text{newloc}(l); k']) = \{ k, k' \}
$$

We set $\lambda(\mu_{\rightarrow_{\mu}}) = \lambda(\mu_{\rightarrow_{\nu}})$. The stamp of a memory defines the resources used by the corresponding transitions. The notation $\tau(z)$ 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 $\kappa$ to range over tags $\tau(z)$ and $z$. We define the closure w.r.t. a net $N$ of a tag $\kappa$ as $\text{closure}_{\nu}(N)(\kappa) = \{ \kappa \} \cup \text{closure}_{\nu}(h)$ if $\kappa = k_1$ or $\kappa = k_2$ and $h \prec (k_1, k_2)$ occurs in $N$, $\emptyset$ otherwise. We define the closure over a set $K$ of tags as $\text{closure}_{\nu}(K) = \bigcup_{\kappa \in K} \text{closure}_{\nu}(\kappa)$. 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 coinital transitions $M \rightarrow_{\nu} N_1$ and $M \rightarrow_{\nu} N_2$ are in conflict if, for some resource $z$, one of the following holds: (i) $z \in \text{closure}_{\mu}(N_1)(\lambda(\eta_1))$ and $z \notin \text{closure}_{\mu}(N_2)(\lambda(\eta_2))$, (ii) $\tau(z) \in \lambda(\eta_1)$ and $z \notin \text{closure}_{\mu}(N_2)(\lambda(\eta_2))$, or (iii) $z \in \text{closure}_{\mu}(N_1)(\lambda(\eta_1))$ and $\tau(z) \in \lambda(\eta_2)$. Two coinital 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 them 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 cofinal concurrent transitions, then there exist two cofinal transitions $\delta_2/\delta_1 = N_1 \xrightarrow{\eta} N$ and $\delta_1/\delta_2 = N_2 \xrightarrow{\eta} N$.

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

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

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 initial traces, then that traces which are not causal equivalent lead to distinct nets.

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

IV. CONTROLLED REVERSIBILITY

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

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

$a ::= \text{out}_i(t)@t^l \mid \text{eval}_i(P)@t^l \mid \text{in}_i(T)@t^l \mid \text{read}_i(T)@t^l \mid \text{newloc}_i(t)$

$\mu ::= [k : \text{out}_i(t)@t^l.P; k'; k'] \mid [k : \text{in}_i(T)@t^l.P; h: (t); k'] \mid [k : \text{read}_i(T)@t^l.P; h; k'] \mid [k : \text{newloc}_i(t).P; k']$

TABLE IX. CRµKLAIM SYNTAX

CRµKLAIM syntax extends $\mu$KLAIM syntax on two respects. First, actions in CRµKLAIM are labeled by references $\gamma$, which act as variables for keys. Second, CRµKLAIM introduces process $\text{roll}(\gamma)$, which undoes the action labeled by $\gamma$. To simplify the technicalities, we change the syntax of memories as well, recording the continuation process also in $\text{out}$, $\text{eval}$, and $\text{newloc}$ memories. Formally, we update the syntax of processes, actions and memories as reported in Table IX. Other syntactic categories are unchanged. At runtime references $\gamma$ are replaced by keys $k$, thus we use $i$ to range over both $\gamma$ and $k$. If $a_\gamma$ denotes an action labeled by $\gamma$, then $\gamma$ 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 $\text{roll}(k)$ (the argument of $\text{roll}$ is always a reference), nor free occurrences of references. Well-formedness changes accordingly. Structural congruence coincides with the one of $\mu$KLAIM. 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 $\text{out}$ memory, the read tuple and the continuation key in an $\text{in}$ 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:

$N \equiv \sum_{\mu \in L} \left( l : \prod_{i \in I} (k_i : P_i) \mid \prod_{j \in J} (k_j : a_j.P_j; \xi_j) \mid \prod_{h \in H} (k_h \prec (k_h^2; k_h^3)) \mid \prod_{x \in X} (k_x : \langle et \rangle x) \mid \prod_{w \in W} (k_w : \text{in}_w(T_w)@l_w.P_w; k_w^2 : \langle t \rangle w; k_w^3) \mid \prod_{y \in Y} (k_y : \text{read}_y(T_y)@l_y.P_y; k_y^2 : \langle k \rangle y) \right)$

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 $\rightarrow_\gamma$ and a backward reduction relation $\leftarrow_\gamma$. 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 $\gamma$ with the proper key.

Backward reductions in CRµKLAIM correspond to executions of the roll operator. Since all the occurrences of references $\gamma$ are bound, when a roll becomes enabled its argument is always a key $k$, uniquely identifying the memory created by the action to be undone. Thus, backward reductions are defined by the semantics of roll($k$). 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 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 $\prec$) 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 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 $N_r$. Finally, resources taken by the computation from the context should be given back to the context. This is the role of $N_{\not\leftarrow k}$ (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 $\prec : N \times T_N$ is the smallest preorder (i.e., reflexive and transitive relation) satisfying:

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

TABLE IX. CRµKLAIM SYNTAX
\[
M = (\nu \bar{z} \ldots) \mapsto k' : \text{roll}(k) \mapsto \bar{t}' \:: \:\{ k : a.P; \xi \} \:: N \quad k < \mu M
\]
\[
N_k = l' : h : \langle t \rangle \quad \text{if } a = \text{in}, (T)@t' \land \xi = h : \langle t \rangle@k'; \text{ otherwise } N_k = \emptyset
\]
\[
N_k = 0 \text{ if } k < \mu M \lambda l, \text{ otherwise } N_k = l :: \emptyset
\]

(\nu \bar{z}) \mapsto k' :: \text{roll}(k) \mapsto \bar{t}' :: \{ k : a.P; \xi \} :: N \quad \mu N_k = 0 \quad \mu N_k = l :: \emptyset

(Roll)

\[\]
\[l :: \text{out}_s(t)@l'.P \quad \bar{t}' :: \emptyset \rightarrow_c (o(k', k'')) \quad (l :: \text{out}_s(t)@l'.P : k'' ; k') \quad \bar{t}' :: \emptyset \rightarrow_c (\langle t \rangle) \]

(Out)

\[\]
\[l :: \text{in}_s(T)@l'.P \quad \bar{t}' :: \langle t \rangle \rightarrow_c (o(k') \quad (l :: \text{in}_s(T)@l'.P : k' ; \xi) \quad \bar{t}' :: \langle t \rangle \]

(In)

\[\]
\[l :: \text{read}_s(T)@l'.P \quad \bar{t}' :: \langle t \rangle \rightarrow_c (o(k') \quad (l :: \text{read}_s(T)@l'.P : k' ; k) \quad \bar{t}' :: \langle t \rangle \]

(Read)

\[\]
\[l :: \text{eval}_s(Q)@l'.P \quad \bar{t}' :: \emptyset \rightarrow_c (o(k', k'') \quad (l :: \text{eval}_s(Q)@l'.P : k'' ; k') \quad \bar{t}' :: \emptyset \rightarrow_c (Q) \]

(Eval)

\[
\begin{array}{c}
\text{TABLE X.} \quad \text{CRLA@M operational semantics}\end{array}
\]

\[\bullet \quad k < : N \quad z \text{ if } \{ k : \text{newloc}_\gamma(l).P ; k' \} \text{ occurs in } N, \text{ with } z = l \text{ or } z = k'; \\
\bullet \quad l < : N \quad k \text{ if } l :: k : P \quad \text{or } l :: k : \langle t \rangle \text{ occurs in } N.
\]

Note that for action \text{out} the continuation and the tuple depend on the action, while for actions \text{in} and \text{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 \((k\text{-dependence})\): Let \( N \) be a CRLA@M net in normal form (see Lemma 6). Net \( N \) is \( k \)-dependent, written \( k < : N \), if: (i) for every \( i \in \text{I} \cup \text{J} \cup \text{H} \cup \text{X} \) we have \( k < : N \; k_i \); (ii) for every \( i \in \text{W} \cup \text{Y} \) we have \( k < : N \; k_i^1 \) or \( k < : N \; k_i^2 \); and (iii) for every \( z \in \bar{z} \) 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:

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

After two steps the net becomes:

\[
(\nu k'', k''') \mapsto \text{roll}(k) \quad (k' :: \langle \text{foo} \rangle \quad (l :: \text{out}_s(\text{foo})@l.l(\text{in}_s(\text{foo})@l.l(\text{roll}(\gamma))) : k'' ; k''') \quad (l :: \text{in}_s(\text{foo})@l.l(\text{roll}(\gamma))) : k' :: \langle \text{foo1} ; k''' \rangle)
\]

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

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

Definition 8 (Projection): Let \( N \) be a net in normal form (see Lemma 6). If \( k \not\in \bar{z} \) then:

\[
N_{k'} \equiv \left( \nu \bar{z} \right) \left( l :: \left( \prod_{w \in W'} k_1 : \text{in}_{\gamma}(T_w)@l_w.P_w \right) \prod_{y \in Y'} k_2 : \text{read}_{\gamma}(T_y)@l_y.P_y \right) \right) \left( l :: k_0 : \langle t_w \rangle \right)
\]

where \( L' = \{ l \in L \mid k \not\in N \; l \} \), \( W' = \{ w \in W \mid k \not\in N \; k_1 \} \), \( Y' = \{ y \in Y \mid k \not\in N \; k_2 \} \) and \( W'' = \{ w \in W \mid k \not\in N \; k_2 \} \).

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

Theorem 2: Given a CRLA@M net \( N \), if \( N \rightarrow_c M \) then erCon\( (M) \rightarrow_{erCon} \) erCon\( (M) \) and if \( N \rightarrow_{erCon} M \) then erCon\( (N) \rightarrow_{erCon} \) erCon\( (M) \) where \( \rightarrow_{erCon} \) is the transitive closure of \( \rightarrow_{erCon} \).

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 \texttt{CRKLAIM} as the net in Table XI, where:

\[
P_1 = \text{in}('chgPr', \mufranchisee, \text{roll}(\gamma))
\]

\[
P_2 = \text{in}('chgPr', \mufranchisee, \text{roll}(\gamma))
\]

\[
Q_1 = \text{in}('chgPr', \mufranchisee, \text{read}(\gamma))
\]

\[
Q_2 = \text{in}('chgPr', \mufranchisee, \text{read}(\gamma))
\]

The market is a storage of tuples, representing lots of goods, of the form \texttt{("lot", v, l)}, 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 \texttt{price()}. Then, it waits for a \texttt{change price} request (i.e., a tuple \texttt{("chgPr")}) to trigger the rollback of the executed activity (by means of the \texttt{roll} operator). Such tuple could be generated by a local process monitoring the selling trend that we left 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 \texttt{applyLocalPolicy()}). 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 \(Q'_1\) and \(Q''_1\) denote the continuations of the \texttt{in} and \texttt{read} actions.

If \(\mufranchisee_1\) needs to change its lot of goods, a tuple \texttt{("chgPr")} is locally produced and, then, the rollback is triggered by \texttt{roll(\mufranchisee_1)}. In this way, the memory of the first \texttt{in} will be directly restored (and its forward history deleted), rather than undoing action-by-action the forward execution (as in \texttt{RKLAIM}). As expected, the backward step does not affect \(\mufranchisee_2\). Instead, if \(\mufranchisee_1\) wants to change the price stored in \(k^4 = \{\text{"suggPrice","170\}"}\), it undoes action \(k_4 = \text{out}('suggPrice', price()) \mufranchisee_1\) by also involving the \texttt{read} memories within \(\mufranchisee_1\) and \(\mufranchisee_2\), because \(k^4\) occurs within them, and all the occurrences of \(k^4\) must be considered to apply rule (Roll). Thus, the projection operation \(\tilde{k}_4\) 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 \texttt{read(\texttt{"suggPrice", \text{\textit{price}}})@ufr, by \texttt{in(\texttt{"suggPrice", \text{\textit{price}}})@ufr, \texttt{out(\texttt{\textit{price}}})@ufr, 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 \(P_1\) and \(Q_1\) to \texttt{nil} and removing all references, we obtain a \texttt{RKLAIM} 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.

\section{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 Hôtőr [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. The results we discussed correspond to some of the results in [14], [13], which were obtained in the simpler framework of Hôtőr.
We have not yet transported to \(\mu\text{KLAIM}\) 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 \(R\mu\text{KLAIM}\) and \(CR\mu\text{KLAIM}\), 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 \(CR\mu\text{KLAIM}\) is to attach them to tuples. For instance, \(k : \langle \texttt{foo}\rangle\%\langle \texttt{foo1}\rangle\) would mean “try \(\texttt{foo}\), then try \(\texttt{foo1}\)”. Such a tuple behaves as \(k : \langle \texttt{foo}\rangle\), but it becomes \(k : \langle \texttt{foo1}\rangle\) when it is inside a memory of an \(in\), and a \texttt{roll}\ targeting the memory is executed. As in HO\(\pi\), such a simple mechanism considerably increases the expressiveness.

A faithful encoding of \(R\mu\text{KLAIM}\) and \(CR\mu\text{KLAIM}\) into \(\mu\text{KLAIM}\) itself would follow the lines of [14]. Its definition would be simpler than that for reversible HO\(\pi\) [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 \(CR\mu\text{KLAIM}\), 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 \texttt{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**


