Causal-Consistent Reversibility in a Tuple-Based Distributed Language

Francesco Tiezzi
IMT Advanced Studies Lucca

Joint work with Elena Giachino, Ivan Lanese and Claudio Mezzina
Map of the talk

- Reversibility
- Klaim
- Uncontrolled reversibility in Klaim
- Controlling reversibility: roll operator
- Conclusions
What is reversibility?

The possibility of executing a computation both in the standard, forward direction, and in the backward direction, going back to a past state.

Reversibility everywhere
- chemistry/biology
- quantum computing
- state space exploration
- ...
Our aim

- We want to exploit reversibility for programming reliable concurrent/distributed systems
  - To make a system reliable we want to avoid “bad” states
  - If a bad state is reached, reversibility allows to go back to some past state

- Understanding reversibility is the key to
  - Understand existing patterns for programming reliable systems, e.g. checkpointing, rollback-recovery, transactions, …
  - Combine and improve them
  - Develop new patterns
Reverse execution of a sequential program

- Recursively undo the last step
  - Computations are undone in reverse order
  - To reverse A;B reverse B, then reverse A

- We want the Loop Lemma to hold
  - From state S, doing A and then undoing A should lead back to S
  - From state S, undoing A (if A is in the past) and then redoing A should lead back to S
Different approaches to reversibility

● Undoing computational steps, not necessarily easy
  – Computation steps may cause loss of information
  – $X=5$ causes the loss of the past value of $X$

● Considering languages which are reversible
  – Featuring only actions that cause no loss of information

● Taking a language which is not reversible and make it reversible
  – One should save information on the past configurations
  – $X=5$ becomes reversible by recording the old value of $X$
Reversibility and concurrency

- In a sequential setting, recursively undo the last action

- Which is the last action in a concurrent setting?
  - Many possibilities
  - For sure, if an action A caused an action B, A could not be the last one

- **Causal-consistent reversibility**: recursively undo any action whose consequences (if any) have already been undone
Causal-consistent reversibility
Reversibility and concurrency

- Causal-consistent reversibility allows one to distinguish non-determinism from concurrency

- Two sequential actions whose order is chosen nondeterministically should be reversed in reverse order

- Two concurrent actions can be reversed in any order
  - Choosing an interleaving for them is an arbitrary choice
  - It should have no impact on the possible reverse behaviors
Map of the talk

- Reversibility
- Klaim
- Uncontrolled reversibility in Klaim
- Controlling reversibility: roll operator
- Conclusions
Klaim

- Coordination language based on distributed tuple spaces
  - Linda operations for creating and accessing tuples
  - Tuples accessed via pattern-matching
- Klaim nets composed by distributed nodes containing processes and data (tuples)
- We consider a subset of Klaim called $\mu$Klaim
\( \muKlaim \) syntax

(Nets) \( N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
\( \mu \text{Klaim syntax} \)

\begin{align*}
(\text{Nets}) & \quad N :::= 0 \quad | \quad l ::= C \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu l)N \\
(\text{Components}) & \quad C :::= \langle et \rangle \quad | \quad P \quad | \quad C_1 \mid C_2 \\
(\text{Processes}) & \quad P :::= \text{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \\
(\text{Actions}) & \quad a :::= \text{out}(t)@l \quad | \quad \text{eval}(P)@l \quad | \quad \text{in}(T)@l \quad | \quad \text{read}(T)@l \quad | \quad \text{newloc}(l)
\end{align*}
μKlaim syntax

(Nets) \( N ::= 0 \mid l :: C \mid N_1 || N_2 \mid (\nu l)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 | C_2 \)

(Processes) \( P ::= nil \mid a.P \mid P_1 | P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \)
\( \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
μKlaim syntax

(Nets) $N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N$

(Components) $C ::= \langle et \rangle \mid P \mid C_1 \mid C_2$

(Processes) $P ::= \text{nil} \mid a.P \mid P_1 \parallel P_2 \mid A$

(Actions) $a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l)$
\( \mu \text{Klaim syntax} \)

(Nets) \( N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
μKlaim syntax

(Nets) \( N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (vl)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \)
\( \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
\(\mu\text{Klaim syntax}\)

(Nets) \( N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 | C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \)
\( \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
### µKlaim Syntax

| Nets          | 0            | l :: C           | N₁ || N₂            | (νl)N               |
|---------------|--------------|------------------|--------------------|---------------------|
| Components    | C ::= ⟨et⟩   | P                | C₁ | C₂                 |
| Processes     | P ::= nil    | a.P              | P₁ | P₂                 | A                   |
| Actions       | a ::= out(t)@l | eval(P)@l      | in(T)@l            | read(T)@l          | newloc(l)           |

![Diagram]

- P
- Q

$l$

- eval(R)@l
\( \muKlaim \) syntax

\[
\begin{align*}
(Nets) \quad N & ::= 0 \quad \mid \quad l :: C \quad \mid \quad N_1 \parallel N_2 \quad \mid \quad (vl)N \\
(Components) \quad C & ::= \langle et \rangle \quad \mid \quad P \quad \mid \quad C_1 \mid C_2 \\
(Processes) \quad P & ::= \text{nil} \quad \mid \quad a.P \quad \mid \quad P_1 \mid P_2 \quad \mid \quad A \\
(Actions) \quad a & ::= \text{out}(t)@l \quad \mid \quad \text{eval}(P)@l \quad \mid \quad \text{in}(T)@l \quad \mid \quad \text{read}(T)@l \quad \mid \quad \text{newloc}(l)
\end{align*}
\]
\( \mu \text{Klaim syntax} \)

\[
\begin{align*}
\text{(Nets)} \quad N &::= 0 \quad | \quad l :: C \quad | \quad N_1 \parallel N_2 \quad | \quad (vl)N \\
\text{(Components)} \quad C &::= \langle et \rangle \quad | \quad P \quad | \quad C_1 | C_2 \\
\text{(Processes)} \quad P &::= \text{nil} \quad | \quad a.P \quad | \quad P_1 | P_2 \quad | \quad A \\
\text{(Actions)} \quad a &::= \text{out}(t)@l \quad | \quad \text{eval}(P)@l \quad | \quad \text{in}(T)@l \quad | \quad \text{read}(T)@l \quad | \quad \text{newloc}(l)
\end{align*}
\]
μKlaim syntax

\[
\begin{align*}
(Nets) \quad & N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (vl)N \\
\text{(Components)} \quad & C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \\
\text{(Processes)} \quad & P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \\
\text{(Actions)} \quad & a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l)
\end{align*}
\]
μKlaim syntax

(Nets) \( N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \parallel P_2 \mid A \)

(Actions) \( a ::= \text{out}(t)@l \mid \text{eval}(P)@l \mid \text{in}(T)@l \mid \text{read}(T)@l \mid \text{newloc}(l) \)
μKlaim syntax

(Nets)  \( N ::= 0 \ | \ l ::= C \ | \ N_1 || N_2 \ | \ (\nu l)N \)

(Components)  \( C ::= \langle et \rangle \ | \ P \ | \ C_1 \ | \ C_2 \)

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

(Actions)  \( a ::= \text{out}(t)@l \ | \ \text{eval}(P)@l \ | \ \text{in}(T)@l \ | \ \text{read}(T)@l \ | \ \text{newloc}(l) \)
μKlaim syntax

(Nets) \( N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 \parallel C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \parallel P_2 \mid A \)

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

\[ l \]

[Diagram showing processes P, Q, R connected by newloc(l') and nets <et2>, <et3>]
μKlaim syntax

(Nets) \( N ::= 0 \mid l :: C \mid N_1 \parallel N_2 \mid \nu l N \)

(Components) \( C ::= \langle et \rangle \mid P \mid C_1 \mid C_2 \)

(Processes) \( P ::= \text{nil} \mid a.P \mid P_1 \mid P_2 \mid A \)

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

\( l \quad \text{newloc}(l') \quad l' \)
μKlaim semantics

\[
\frac{[t] = et}{l :: \textbf{out}(t)@l'.P \ || \ l' :: \textbf{nil} \ \rightarrow \ l :: P \ || \ l' :: \langle et \rangle} \quad (Out)
\]
\[ [t] = et \]

\[
\frac{\text{Out}}{l :: \text{out}(t)@l'.P \parallel l' :: \text{nil} \rightarrow l :: P \parallel l' :: \langle et \rangle}
\]

\[ \text{match}([T], et) = \sigma \]

\[
\frac{\text{In}}{l :: \text{in}(T)@l'.P \parallel l' :: \langle et \rangle \rightarrow l :: P\sigma \parallel l' :: \text{nil}}
\]
\( [t] = et \)

\[
\frac{}{l :: \text{out}(t)@l'.P \ || \ l' :: \text{nil} \ \mapsto \ l :: P \ || \ l' :: \langle et \rangle \quad (Out)}
\]

\[
match([T], et) = \sigma
\]

\[
\frac{}{l :: \text{in}(T)@l'.P \ || \ l' :: \langle et \rangle \ \mapsto \ l :: P\sigma \ || \ l' :: \text{nil} \quad (In)}
\]

\[
match([T], et) = \sigma
\]

\[
\frac{}{l :: \text{read}(T)@l'.P \ || \ l' :: \langle et \rangle \ \mapsto \ l :: P\sigma \ || \ l' :: \langle et \rangle \quad (Read)}
\]

\[
l :: \text{newloc}(l').P \ \mapsto \ (\nu l')(l :: P \ || \ l' :: \text{nil}) \quad (New)
\]

\[
l :: \text{eval}(Q)@l'.P \ || \ l' :: \text{nil} \ \mapsto \ l :: P \ || \ l' :: Q \quad (Eval)
\]
μKlaim semantics

Evaluation-closed relation
A relation is evaluation closed if it is closed under active contexts
\[ N_1 \leftrightarrow N_1' \implies N_1 \parallel N_2 \leftrightarrow N_1' \parallel N_2 \text{ and } (\nu l) N_1 \leftrightarrow (\nu l)N_1' \]
and under structural congruence
\[ N \equiv M \leftrightarrow M' \equiv N' \implies N \leftrightarrow N' \]

μKlaim semantics
The μKlaim reduction relation \( \leftrightarrow \) is the smallest evaluation-closed relation satisfying the rules in previous slide
Example

\[ l_1 :: \langle \text{foo} \rangle \ || \ l_2 :: \text{read}(\text{foo})@l_1.P \ || \ l_3 :: \text{read}(\text{foo})@l_1.P' \]
Map of the talk

- Reversibility
- Klaim
- Uncontrolled reversibility in Klaim
- Controlling reversibility: roll operator
- Conclusions
Making $\mu$Klaim reversible

- We define $R\mu$Klaim, an extension of $\mu$Klaim allowing:
  - *forward* actions, modeling $\mu$Klaim actions
  - *backward* actions, undoing them

- Some new problems arise

- Read dependencies
  - Two *reads* on the same tuple should not create dependences
  - If the *out* creating the tuple is undone then *reads* on the tuple should be undone too

- Localities
  - Localities are now resources and establish dependences
  - To undo a *newloc* one has to undo all the operations on the created locality
RµKlaim syntax

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

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

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

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

(Memories) $\mu ::= [k : \text{out}(t)@l; k''; k'] \mid [k : \text{in}(T)@l.P; h : \langle et \rangle; k']$

\hspace{1cm} $| [k : \text{read}(T)@l.P; h; k'] \mid [k : \text{newloc}(l); k']$

\hspace{1cm} $| [k : \text{eval}(Q)@l; k''; k']$
RµKlaim semantics – tuple operators

\[ [t] = et \]

\[ \frac{\text{Out}}{\rightarrow_r (\nu k', k'') (l :: k' :: P \mid [k : \text{out}(t) @ l'; k''; k'] \parallel l' :: k'' : \langle et \rangle) } \]
\[ [t] = et \]
\[ l :: k : \texttt{out}(t)@l'.P \parallel l' :: \texttt{empty} \]
\[ \mapsto_r (\nu k', k'')(l :: k' : P \parallel [k : \texttt{out}(t)@l'; k'' ; k'] \parallel l' :: k'' : \langle et \rangle) \]

\[ (\nu k''')(l :: k' : P \parallel [k : \texttt{out}(t)@l'; k'' ; k'] \parallel l' :: k'' : \langle et \rangle) \]
\[ \mapsto_r l :: k : \texttt{out}(t)@l'.P \parallel l' :: \texttt{empty} \quad (OutRev) \]
RµKlaim semantics – tuple operators

\[
[t] = et
\]

(Out)

\[
l :: k : \text{out}(t)@l'.P \parallel l' :: \text{empty} \\
\rightarrow_r (vk',k'') (l :: k' : P | [k : \text{out}(t)@l'; k'' ; k''] \parallel l' :: k'' : \langle et \rangle )
\]

(OutRev)

\[
(\nu k'') (l :: k' : P | [k : \text{out}(t)@l' ; k'' ; k'] \parallel l' :: k'' : \langle et \rangle ) \\
\leadsto_r l :: k : \text{out}(t)@l'.P \parallel l' :: \text{empty}
\]

\[
\text{match}([T], et) = \sigma
\]

(In)

\[
l :: k : \text{in}(T)@l'.P \parallel l' :: h : \langle et \rangle \\
\rightarrow_r (vk') l :: k' : P\sigma | [k : \text{in}(T)@l'.P ; h : \langle et \rangle ; k'] \parallel l' :: \text{empty}
\]

(InRev)

\[
l :: k' : Q | [k : \text{in}(T)@l'.P ; h : \langle et \rangle ; k'] \parallel l' :: \text{empty} \\
\leadsto_r l :: k : \text{in}(T)@l'.P \parallel l' :: h : \langle et \rangle
\]

\[
\text{match}([T], et) = \sigma
\]

(Read)

\[
l :: k : \text{read}(T)@l'.P \parallel l' :: h : \langle et \rangle \\
\rightarrow_r (vk') l :: k' : P\sigma | [k : \text{read}(T)@l'.P ; h ; k'] \parallel l' :: h : \langle et \rangle
\]

(ReadRev)

\[
l :: k' : Q | [k : \text{read}(T)@l'.P ; h ; k'] \parallel l' :: h : \langle et \rangle \\
\leadsto_r l :: k : \text{read}(T)@l'.P \parallel l' :: h : \langle et \rangle
\]
R\(\mu\)Klaim semantics – distribution operators

\[
\begin{align*}
l &::= k : \text{newloc}(l').P \\
&\longrightarrow_r (\nu l') \left( (\nu k') l :: k' : P \mid [k : \text{newloc}(l'); k'] \parallel l' :: \text{empty} \right) \quad \text{(New)}
\end{align*}
\]

\[
\begin{align*}
(\nu l') \left( l :: k' : P \mid [k : \text{newloc}(l'); k'] \parallel l' :: \text{empty} \right) \\
&\leadsto_r l :: k : \text{newloc}(l').P
\end{align*}
\]

\[
\begin{align*}
l &::= k : \text{eval}(Q)@l'.P \parallel l' :: \text{empty} \\
&\longrightarrow_r (\nu k', k'') \left( l :: k' : P \mid [k : \text{eval}(Q)@l'; k''; k'] \parallel l' :: k'' : Q \right) \quad \text{(Eval)}
\end{align*}
\]

\[
\begin{align*}
l &::= k' : P \mid [k : \text{eval}(Q)@l'; k''; k'] \parallel l' :: k'' : Q \\
&\leadsto_r l :: k : \text{eval}(Q)@l'.P \parallel l' :: \text{empty} \quad \text{(EvalRev)}
\end{align*}
\]
Example

\[ l_1 :: k_1 : \langle \text{foo} \rangle \parallel l_2 :: k_2 : \text{read}(\text{foo})@l_1.P \parallel l_3 :: k_3 : \text{read}(\text{foo})@l_1.P' \]

\[ (\nu k'_3) \ (l_1 :: k_1 : \langle \text{foo} \rangle \parallel l_2 :: k_2 : \text{read}(\text{foo})@l_1.P \parallel l_3 :: k'_3 : P' \mid [k_3 : \text{read}(\text{foo})@l_1.P'; k_1; k'_3]) \]

\[ (\nu k'_2) \ (l_1 :: k_1 : \langle \text{foo} \rangle \parallel l_2 :: k'_2 : P \mid [k_2 : \text{read}(\text{foo})@l_1.P; k_1; k'_2] \parallel l_3 :: k_3 : \text{read}(\text{foo})@l_1.P') \]

\[ (\nu k'_2, k'_3) \ (l_1 :: k_1 : \langle \text{foo} \rangle \parallel l_2 :: k'_2 : P \mid [k_2 : \text{read}(\text{foo})@l_1.P; k_1; k'_2] \parallel l_3 :: k'_3 : P' \mid [k_3 : \text{read}(\text{foo})@l_1.P'; k_1; k'_3]) \]
Example

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

\[
(\nu k'_2, k''_2, k'''_2, k'_3, k''_3, k'''_3)(l_1 :: k'''_3 : \langle \text{foo} \rangle \\
\parallel l_2 :: k''_2 : P | [k_2 : \text{in}(\text{foo})@l_1.\text{out}(\text{foo})@l_1.P; k_1 : \langle \text{foo} \rangle; k'_2] \\
\parallel l_3 :: k''_3 : P' | [k_3 : \text{in}(\text{foo})@l_1.\text{out}(\text{foo})@l_1.P'; k''_2 : \langle \text{foo} \rangle; k'_3] \\
\parallel [k'_3 : \text{out}(\text{foo})@l_1; k'''_3; k''_3])
\]
Example

\[
l_1 :: k_1 : \langle \text{foo} \rangle \parallel l_2 :: k_2 : \text{in}(\text{foo})@l_1.\text{out}(\text{foo})@l_1.P \\
\parallel l_3 :: k_3 : \text{in}(\text{foo})@l_1.\text{out}(\text{foo})@l_1.P'
\]

execute **in** in \( l_2 \)

eexecute **out** in \( l_2 \)

execute **in** in \( l_3 \)

eexecute **out** in \( l_3 \)

it needs \( k_2''' : \langle \text{foo} \rangle \) in \( l_1 \) to perform a backward step

\[
(vk_2', k_2'', k_2'''', k_3', k_3'', k_3''') (l_1 :: k_3''' : \langle \text{foo} \rangle \\
\parallel l_2 :: k_2'' : P \parallel [k_2 : \text{in}(\text{foo})@l_1.\text{out}(\text{foo})@l_1.P; k_1 : \langle \text{foo} \rangle; k_2'] \\
\parallel [k_2' : \text{out}(\text{foo})@l_1; k_2''''; k_2''] \\
\parallel l_3 :: k_3'' : P' \parallel [k_3 : \text{in}(\text{foo})@l_1.\text{out}(\text{foo})@l_1.P'; k_2''' : \langle \text{foo} \rangle; k_3'] \\
\parallel [k_3' : \text{out}(\text{foo})@l_1; k_3''''; k_3'''] )
\]
Properties

- The forward semantics of $R\mu\Lambda$ follows the semantics of $\mu\Lambda$

- The Loop Lemma holds
  - i.e., each reduction in $R\mu\Lambda$ has an inverse

- $R\mu\Lambda$ is causally consistent
  - same approach of previous works
  - different technicalities (due to more complex causality structure)
Concurrency in RµKlaim

- Two transitions are concurrent unless
  - They use the same resource
  - At least one transition does not use it in read-only modality

- Resources defined by function $\lambda$ on memories

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

- **Read** uses the tuple in read-only modality

- All primitives but **newloc** use the locality name in read-only modality
Causal consistency

- **Causal equivalence** identifies traces that differ only for
  - swaps of concurrent transitions
  - simplifications of inverse transitions

- **Casual consistency**: there is a unique way to go from one state to another one up to causal equivalence
  - causal equivalent traces can be reversed in the same ways
  - traces which are not causal equivalent lead to distinct nets
Map of the talk

- Reversibility
- Klaim
- Uncontrolled reversibility in Klaim
- Controlling reversibility: roll operator
- Conclusions
Controlling reversibility

- Uncontrolled reversibility is not suitable for programming

- We use reversibility to define a roll operator
  - To undo a given past action
  - And all its consequences

- We call CR\(\mu\)Klaim the extension of \(\mu\)Klaim with roll
CRμKlaim syntax

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

(Components) \( C ::= k : \langle et \rangle \mid k : P \mid C_1 | C_2 \mid \mu \mid k_1 < (k_2, k_3) \)

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

(Actions) \( a ::= \text{out}_\gamma(t)@l \mid \text{eval}_\gamma(P)@l \mid \text{in}_\gamma(T)@l \mid \text{read}_\gamma(T')@l \mid \text{newloc}_\gamma(l) \)

(Memories) \( \mu ::= [k : \text{out}_\gamma(t)@l.P; k''; k'] \mid [k : \text{in}_\gamma(T)@l.P; h : \langle t \rangle; k'] \)
\( \mid [k : \text{read}_\gamma(T')@l.P; h; k'] \mid [k : \text{newloc}_\gamma(l).P; k'] \)
\( \mid [k : \text{eval}_\gamma(Q)@l.P; k''; k'] \)
Example

- From
  \[
  l :: k : \text{out}_\gamma(\text{foo})@l.\text{roll}(\gamma) \parallel l' :: k' : \text{in}(\text{foo})@l.\text{nil}
  \]

- We get
  \[
  (\nu k'' , k''' , k'''' )
  \]
  \[
  (l :: k'' : \text{roll}(k) \mid [k : \text{out}_\gamma(\text{foo})@l.\text{roll}(\gamma); k'''' ; k'']
  \parallel l' :: k'''': \text{nil} \mid [k' : \text{in}(\text{foo})@l.\text{nil}; k''' : \langle \text{foo} \rangle ; k''''])
  \]

- When we undo the \textbf{out} we need to restore the \textbf{in}
CRµKlaim semantics

\[ l :: k : \text{eval}_\gamma(Q)@l'.P \parallel l' :: \text{empty} \]

\[ \rightarrow_c (\nu k', k'')(l :: k' : P[k/\gamma] \mid [k : \text{eval}_\gamma(Q)@l'.P; k''; k'] \parallel l' :: k'' : Q) \]  

\[ M = (\nu \overline{z})l :: k' : \text{roll}(k) \parallel l' :: [k : a.P; \xi] \parallel N \quad k <: M \quad \text{complete}(M) \]

\[ N_t = l'' :: h : \langle t \rangle \text{ if } a = \text{in}_\gamma(T)@l'' \land \xi = h : \langle t \rangle; k'', \text{ otherwise } N_t = 0 \]

\[ N_i = 0 \text{ if } k <: M \quad l, \text{ otherwise } N_i = l :: \text{empty} \]

\[ (\nu \overline{z})l :: k' : \text{roll}(k) \parallel l' :: [k : a.P; \xi] \parallel N \quad \rightarrow_c l' :: k : a.P \parallel N_t \parallel N_i \parallel N_{\overline{z} k} \]

- M is complete and depends on k
- N_t: if the undone action is an in, we should release the tuple
- N_i: we should not consume the roll locality, unless created by the undone computation
- N_{\overline{z} k}: resources consumed by the computation should be released
Results

- CRµKlaim is a controlled version of RµKlaim
- It inherits all its properties
Map of the talk

- Reversibility
- Klaim
- Uncontrolled reversibility in Klaim
- Controlling reversibility: roll operator
- Conclusions
Summary

- We defined uncontrolled and controlled causal-consistent reversibility for \( \mu \text{Klaim} \)

- Two main features taken into account
  - Read dependences
  - Localities
Future work

- Part of HOπ theory not yet transported to μKlaim
  - Encoding of the reversible language in the basic one
    » Would allow to exploit Klaim implementations
  - Low-level controlled semantics
  - Alternatives

- The killer application may be in the field of STM
Thanks!

Questions?
5th International Workshop on Modeling and Simulation of Peer-to-Peer and Autonomic Systems (MOSPAS 2014)

July 21-25, 2014 - Bologna, Italy

Submission Deadline:
March 11, 2014