Causal-Consistent Reversibility in a Tuple-Based Distributed Language

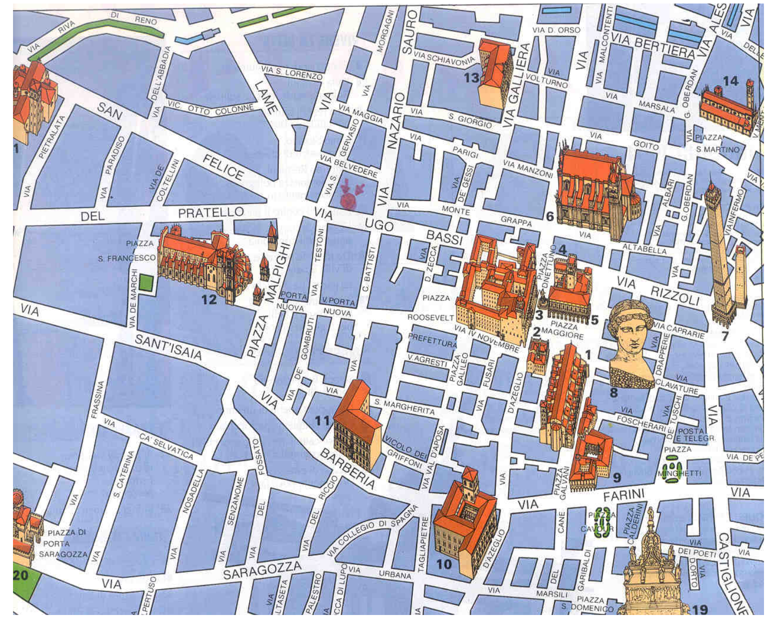
Francesco Tiezzi
IMT Advanced Studies Lucca

CINA
2nd Gen. Meeting
19 February 2014
Bologna

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

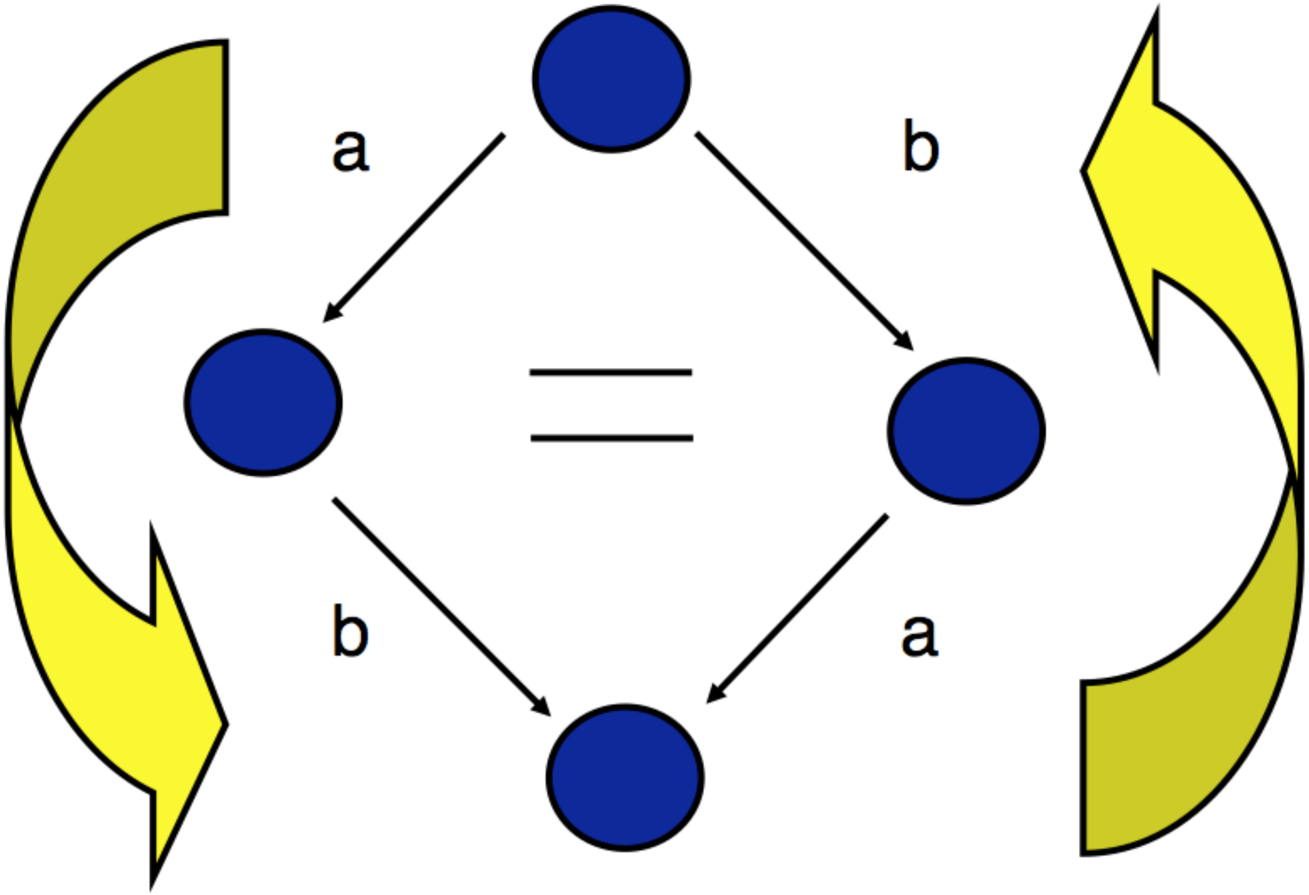
Different approaches to reversibility

- 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

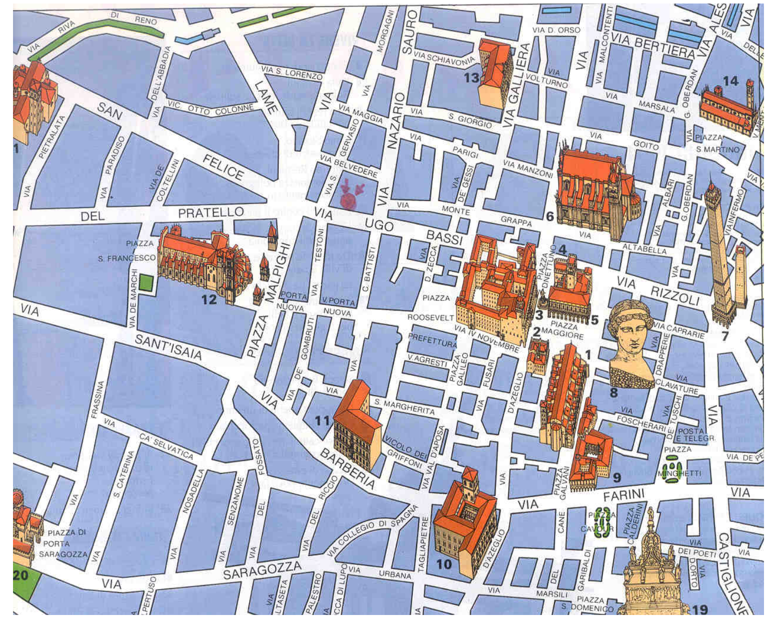
- Causal-consistent reversibility allows one to distinguish non-determinism from concurrency

- Two sequential actions whose order is chosen nondeterminstically 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

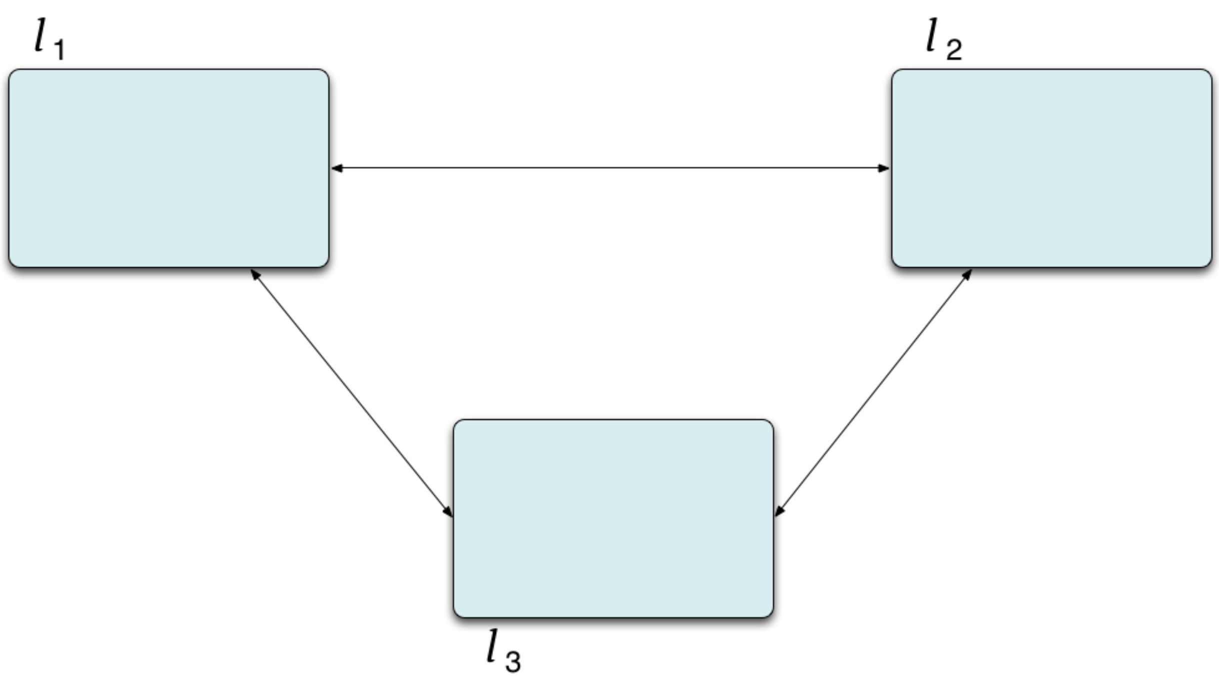
- Coordination language based on distributed tuple spaces
  - Linda operations for creating and accessing tuples
  - Tuples accessed via pattern-matching
- Klaim nets composed by distributed nodes containing processes and data (tuples)
- We consider a subset of Klaim called μKlaim

μKlaim syntax

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

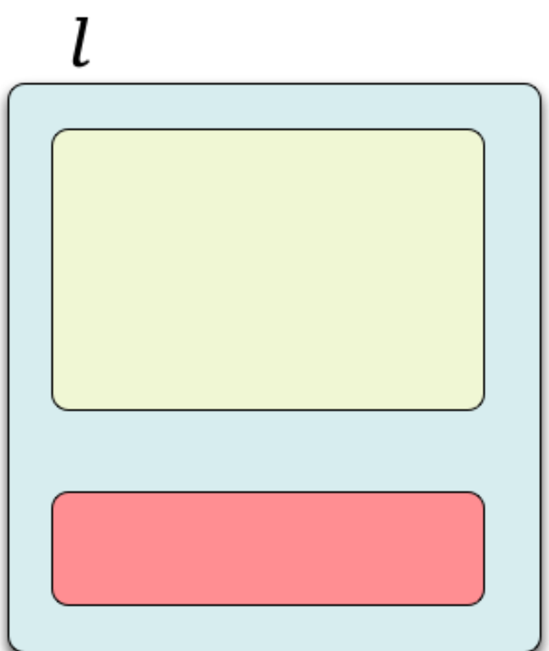
µKlaim syntax

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



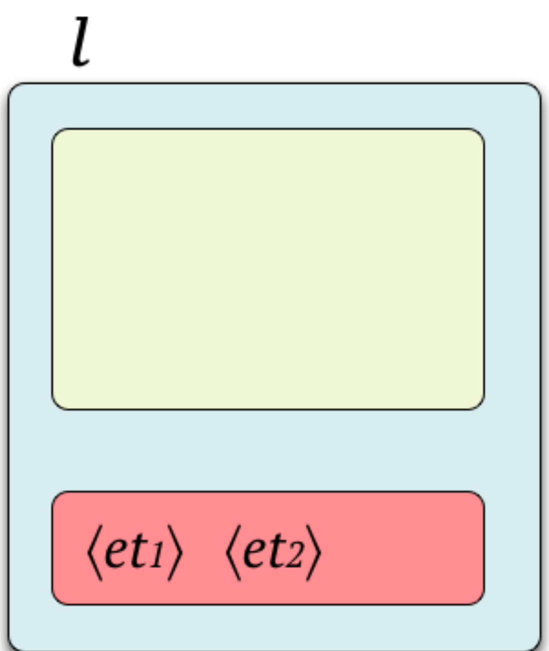
μKlaim syntax

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



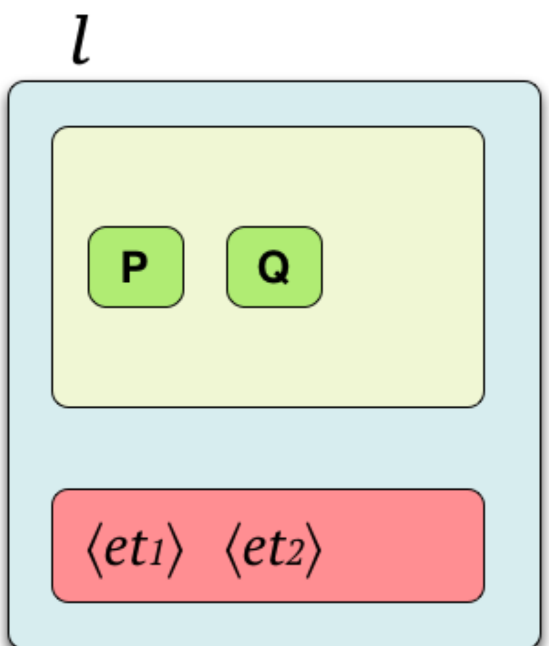
μKlaim syntax

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



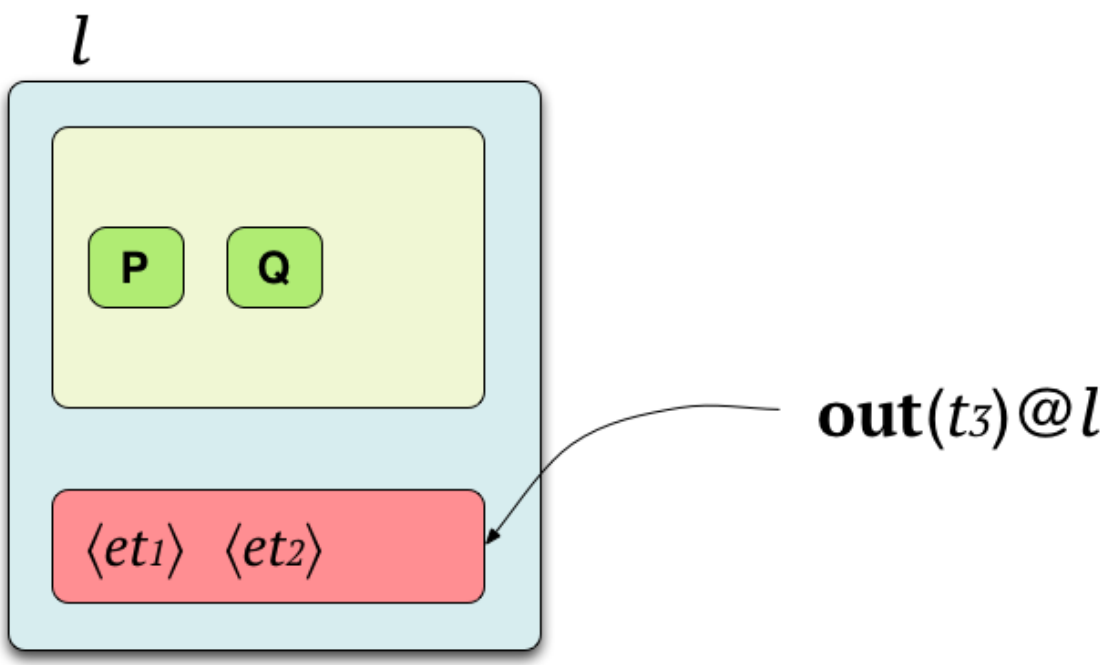
μKlaim syntax

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



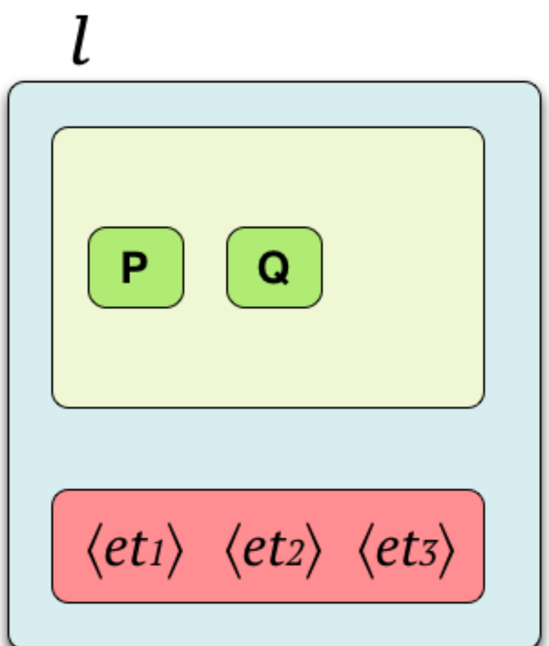
μKlaim syntax

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



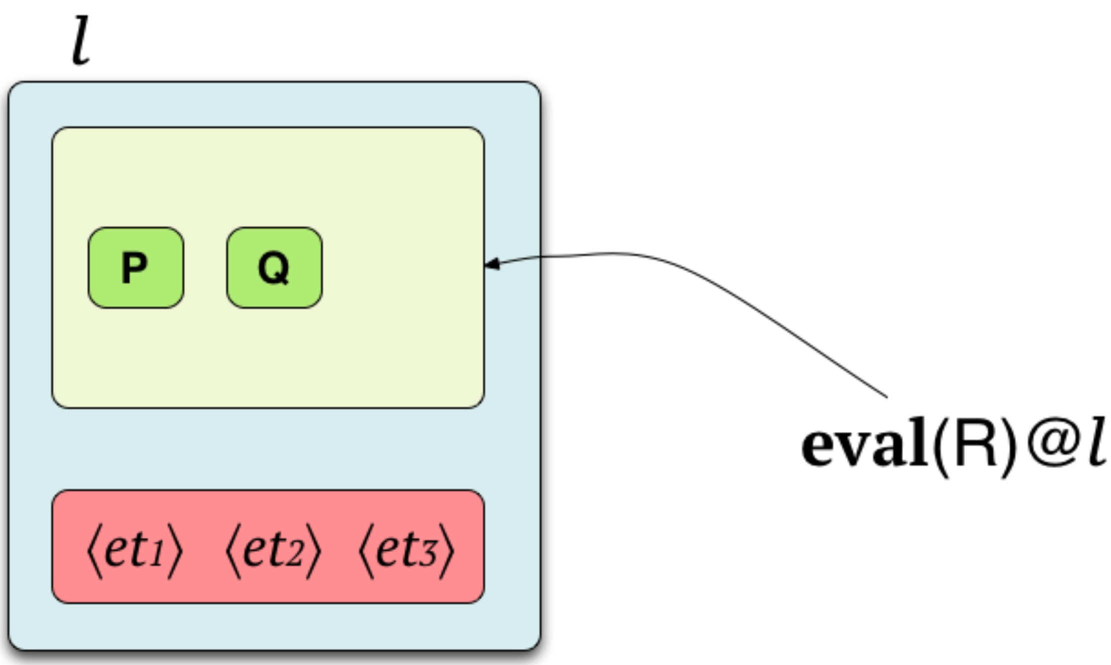
μKlaim syntax

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



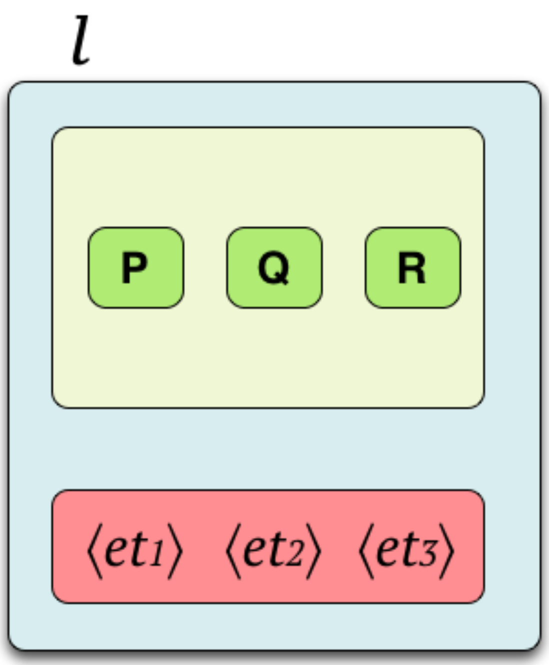
μKlaim syntax

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



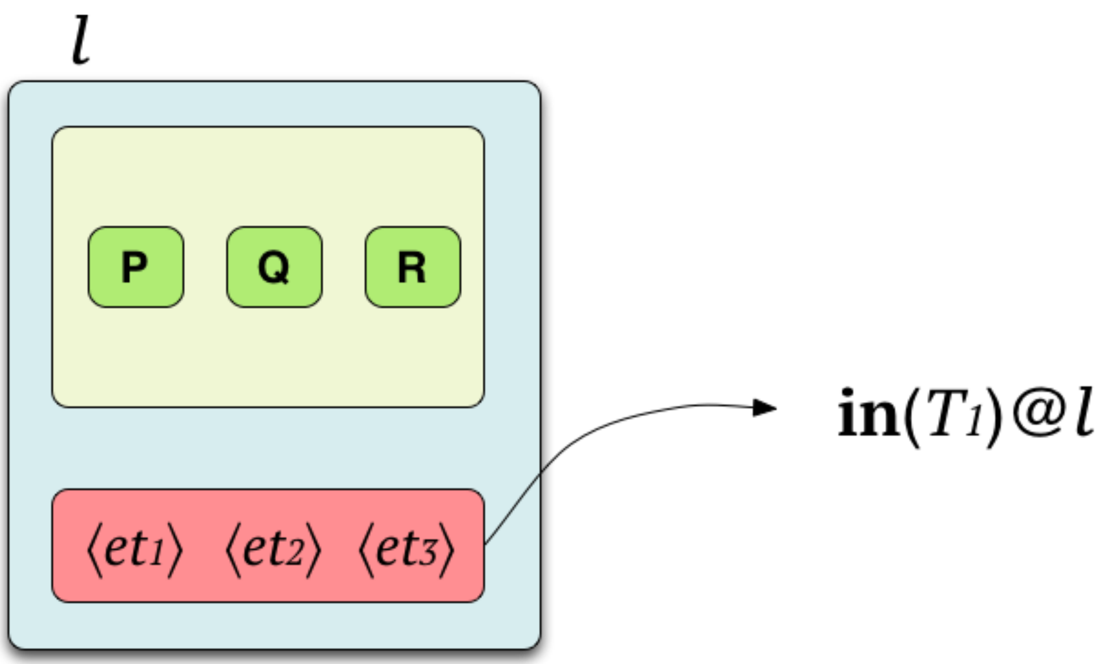
μKlaim syntax

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



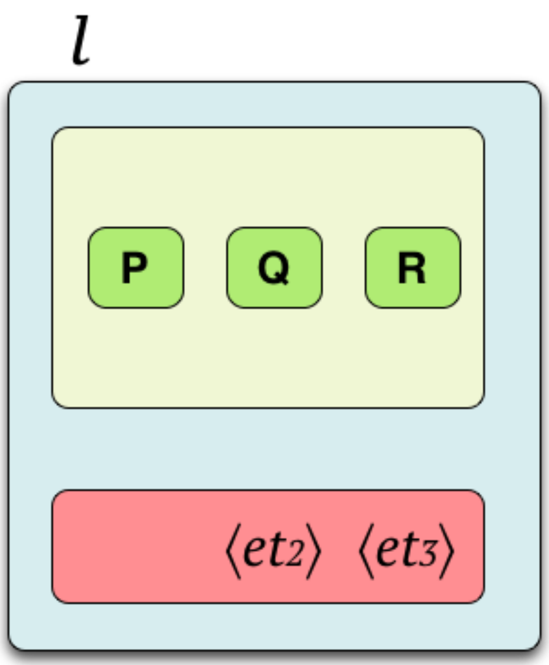
μKlaim syntax

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



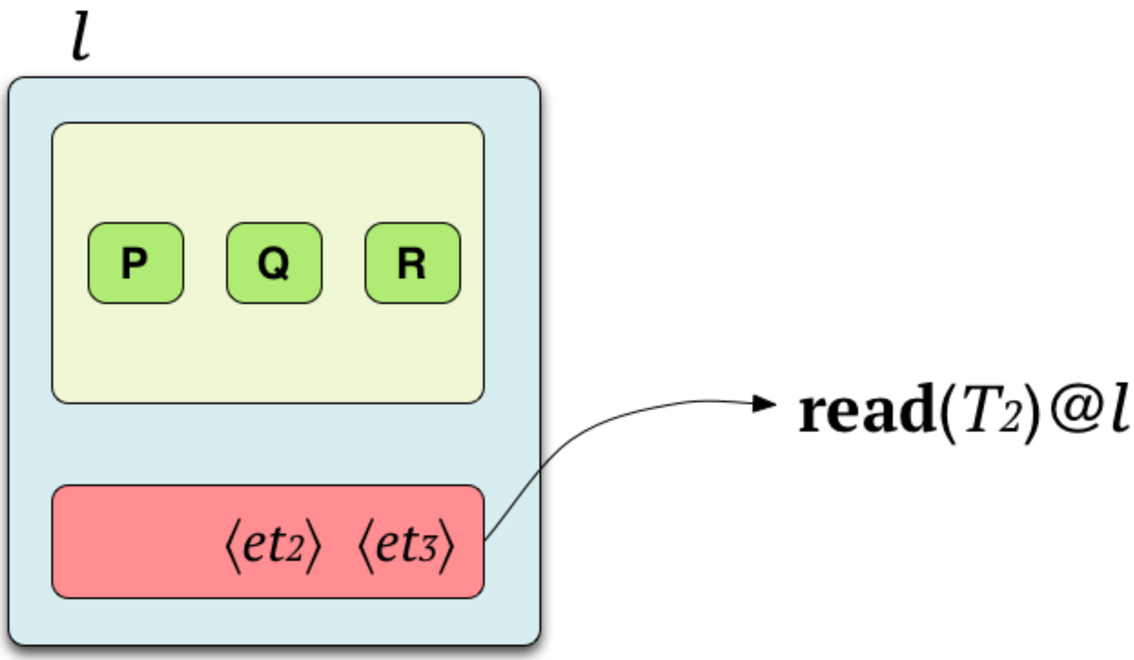
µKlaim syntax

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



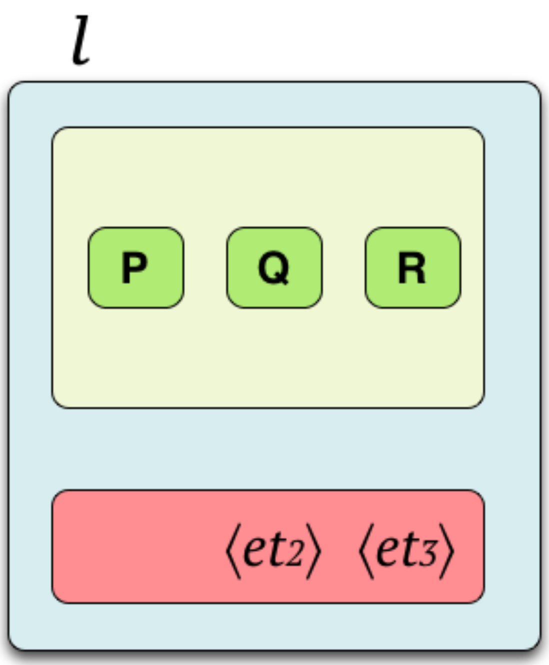
μKlaim syntax

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



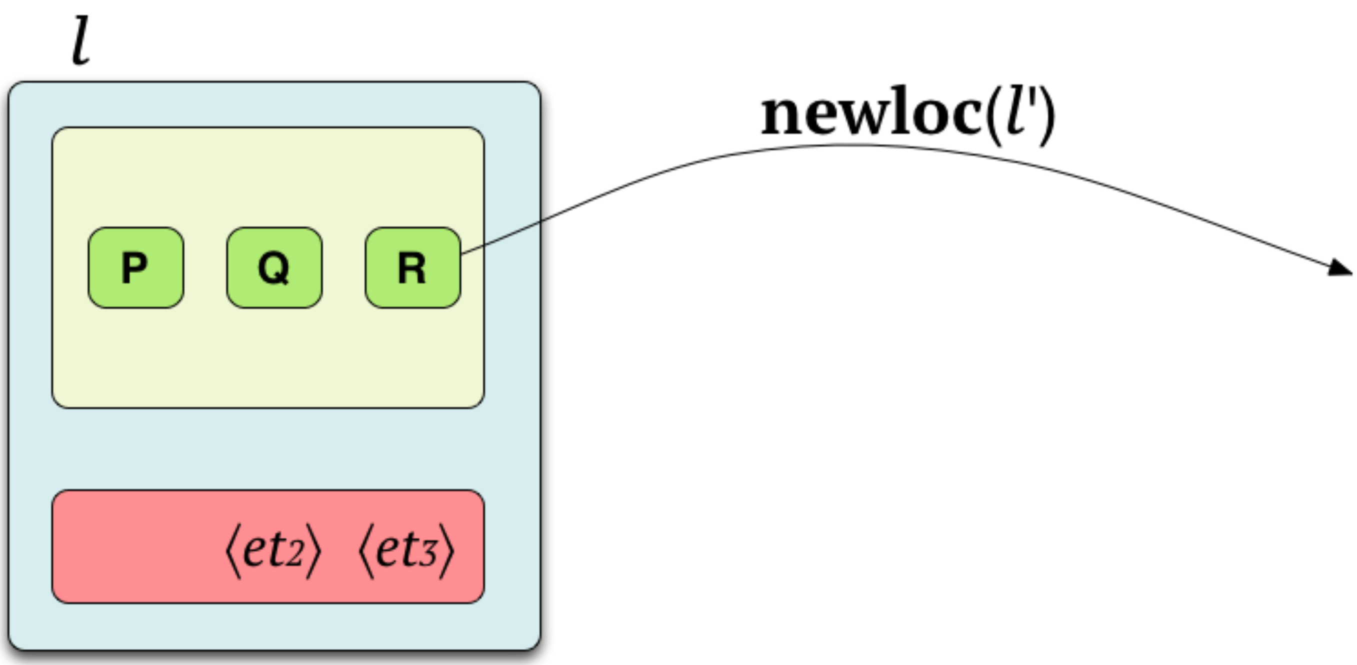
µKlaim syntax

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



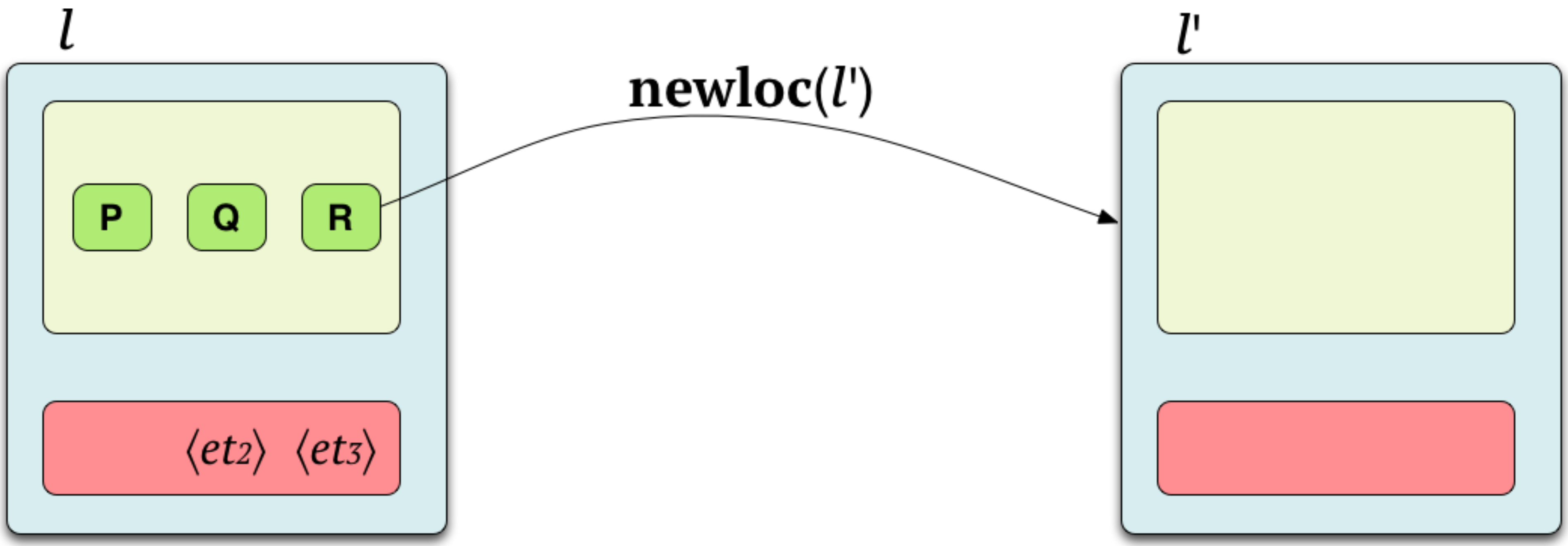
μKlaim syntax

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



μKlaim syntax

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



μKlaim semantics

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

μKlaim semantics

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

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

μKlaim semantics

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

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

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

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

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

μKlaim semantics

Evaluation-closed relation

A relation is evaluation closed if it is closed under active contexts

$$N1 \longmapsto N1' \text{ implies } N1 \parallel N2 \longmapsto N1' \parallel N2 \text{ and } (\nu l)\, N1 \longmapsto (\nu l) N1'$$

and under structural congruence

$$N \equiv M \longmapsto M' \equiv N' \text{ implies } N \longmapsto N'$$

μKlaim semantics

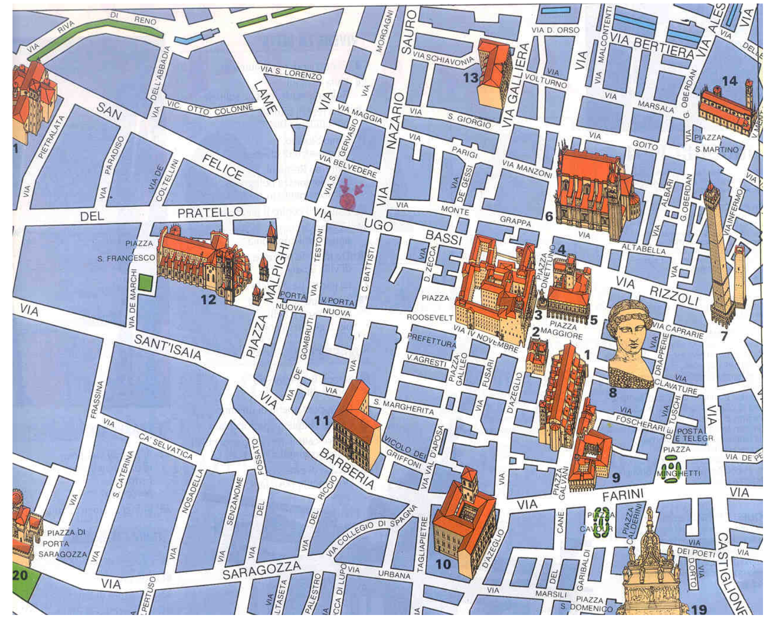
The μKlaim reduction relation $\longmapsto$ is the smallest evaluation-closed relation satisfying the rules in previous slide

Example

$$\begin{tikzcd}[column sep=small]
& l_1 :: \langle \mathit{foo} \rangle \parallel l_2 :: \mathbf{read}(\mathit{foo})@l_1.P \parallel l_3 :: \mathbf{read}(\mathit{foo})@l_1.P' \arrow[dl] \arrow[dr] & \\
l_1 :: \langle \mathit{foo} \rangle \parallel l_2 :: \mathbf{read}(\mathit{foo})@l_1.P \parallel l_3 :: P' \arrow[dr] & & l_1 :: \langle \mathit{foo} \rangle \parallel l_2 :: P \parallel l_3 :: \mathbf{read}(\mathit{foo})@l_1.P' \arrow[dl] \\
& l_1 :: \langle \mathit{foo} \rangle \parallel l_2 :: P \parallel l_3 :: P' &
\end{tikzcd}$$

Map of the talk

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



Making μKlaim reversible

- We define RμKlaim, an extension of μKlaim allowing:
  - *forward* actions, modeling μKlaim actions
  - *backward* actions, undoing them
- Some new problems arise
- Read dependencies
  - Two **read**s on the same tuple should not create dependences
  - If the **out** creating the tuple is undone then **read**s 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

$$
\begin{array}{rcl}
\text{(Nets)}\ N & ::= & \mathbf{0} \quad | \quad l :: C \quad | \quad l :: \mathbf{empty} \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu z)N \\
\text{(Components)}\ C & ::= & k : \langle et \rangle \quad | \quad k : P \quad | \quad C_1 \mid C_2 \quad | \quad \mu \quad | \quad k_1 \prec (k_2, k_3) \\
\text{(Processes)}\ P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \\
\text{(Actions)}\ a & ::= & \mathbf{out}(t)@\ell \quad | \quad \mathbf{eval}(P)@\ell \\
 & | & \mathbf{in}(T)@\ell \quad | \quad \mathbf{read}(T)@\ell \quad | \quad \mathbf{newloc}(l) \\
\text{(Memories)}\ \mu & ::= & [k : \mathbf{out}(t)@l; k''; k'] \quad | \quad [k : \mathbf{in}(T)@l.P; h : \langle et \rangle; k'] \\
 & | & [k : \mathbf{read}(T)@l.P; h; k'] \quad | \quad [k : \mathbf{newloc}(l); k'] \\
 & | & [k : \mathbf{eval}(Q)@l; k''; k']
\end{array}
$$

RμKlaim semantics – tuple operators

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

RμKlaim semantics – tuple operators

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

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

RμKlaim semantics – tuple operators

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

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

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

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

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

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

RμKlaim semantics – distribution operators

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

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

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

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

Example

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

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

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

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

Example

$$\begin{aligned} l_1 \ :: \ k_1 : \langle \mathit{foo} \rangle \ \| \ & l_2 \ :: \ k_2 : \mathbf{in}(\mathit{foo})@l_1.\mathbf{out}(\mathit{foo})@l_1.P \\ \| \ & l_3 \ :: \ k_3 : \mathbf{in}(\mathit{foo})@l_1.\mathbf{out}(\mathit{foo})@l_1.P' \end{aligned}$$

$$\Big\downarrow \text{execute } \mathbf{in} \text{ in } l_2$$

$$\Big\downarrow \text{execute } \mathbf{out} \text{ in } l_2$$

$$\Big\downarrow \text{execute } \mathbf{in} \text{ in } l_3$$

$$\Big\downarrow \text{execute } \mathbf{out} \text{ in } l_3$$

$$\begin{aligned} & (\nu k_2', k_2'', k_2''', k_3', k_3'', k_3''')(l_1 \ :: \ k_3''' : \langle \mathit{foo} \rangle \\ & \ \| \ l_2 \ :: \ k_2'' : P \ | \ [k_2 : \mathbf{in}(\mathit{foo})@l_1.\mathbf{out}(\mathit{foo})@l_1.P; k_1 : \langle \mathit{foo} \rangle; k_2'] \\ & \qquad\qquad\qquad | \ [k_2' : \mathbf{out}(\mathit{foo})@l_1; k_2'''; k_2''] \\ & \ \| \ l_3 \ :: \ k_3'' : P' \ | \ [k_3 : \mathbf{in}(\mathit{foo})@l_1.\mathbf{out}(\mathit{foo})@l_1.P'; k_2''' : \langle \mathit{foo} \rangle; k_3'] \\ & \qquad\qquad\qquad | \ [k_3' : \mathbf{out}(\mathit{foo})@l_1; k_3'''; k_3''] \ ) \end{aligned}$$

Example

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

execute **in** in l_2

execute **out** in l_2

execute **in** in l_3

execute **out** in l_3

it needs $k_2''' : \langle foo \rangle$ in l_1 to perform a backward step

$$(\nu k_2', k_2'', k_2''', k_3', k_3'', k_3''')(l_1 :: k_3''' : \langle foo \rangle$$
$$\parallel l_2 :: k_2'' : P \mid [k_2 : \mathbf{in}(foo)@l_1.\mathbf{out}(foo)@l_1.P; k_1 : \langle foo \rangle; k_2']$$
$$\mid [k_2' : \mathbf{out}(foo)@l_1; k_2'''; k_2'']$$
$$\parallel l_3 :: k_3'' : P' \mid [k_3 : \mathbf{in}(foo)@l_1.\mathbf{out}(foo)@l_1.P'; k_2''' : \langle foo \rangle; k_3']$$
$$\mid [k_3' : \mathbf{out}(foo)@l_1; k_3'''; k_3''])$$

Properties

- The forward semantics of RμKlaim follows the semantics of μKlaim
- The Loop Lemma holds
  - i.e., each reduction in RμKlaim has an inverse
- RμKlaim 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 λ on memories

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

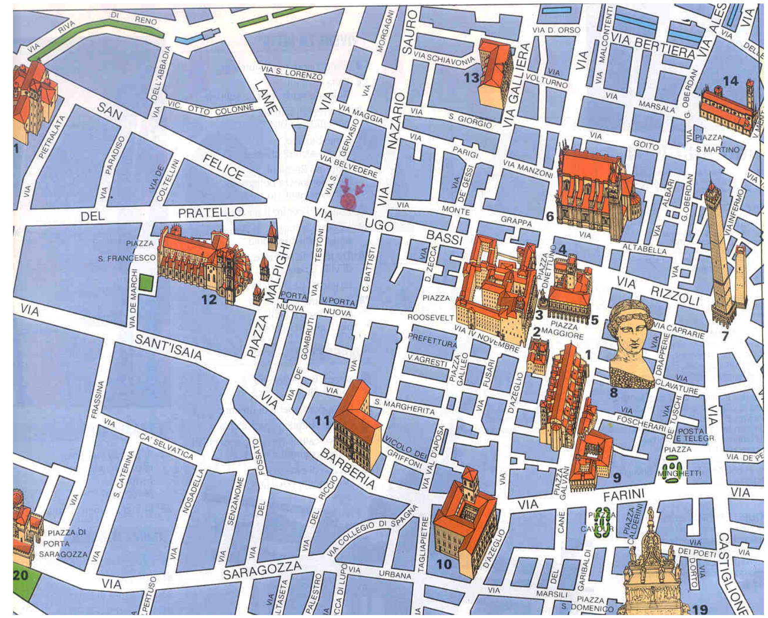
- **Read** uses the tuple in read-only modality
- All primitives but **newloc** use the locality name in read-only modality

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 CRμKlaim the extension of μKlaim with **roll**

CRμKlaim syntax

$$
\begin{array}{rlcl}
\text{(Nets)} & N & ::= & \mathbf{0} \quad | \quad l :: C \quad | \quad l :: \mathbf{empty} \quad | \quad N_1 \parallel N_2 \quad | \quad (\nu z)N \\
\text{(Components)} & C & ::= & k : \langle et \rangle \quad | \quad k : P \quad | \quad C_1 \mid C_2 \quad | \quad \mu \quad | \quad k_1 \prec (k_2, k_3) \\
\text{(Processes)} & P & ::= & \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad A \quad | \quad \mathbf{roll}(\iota) \\
\text{(Actions)} & a & ::= & \mathbf{out}_\gamma(t)@\ell \quad | \quad \mathbf{eval}_\gamma(P)@\ell \quad | \quad \mathbf{in}_\gamma(T)@\ell \quad | \\
 & & & \mathbf{read}_\gamma(T)@\ell \quad | \quad \mathbf{newloc}_\gamma(l) \\
\text{(Memories)} & \mu & ::= & [k : \mathbf{out}_\gamma(t)@l.P; k''; k'] \quad | \quad [k : \mathbf{in}_\gamma(T)@l.P; h : \langle t \rangle; k'] \\
 & & | & [k : \mathbf{read}_\gamma(T)@l.P; h; k'] \quad | \quad [k : \mathbf{newloc}_\gamma(l).P; k'] \\
 & & | & [k : \mathbf{eval}_\gamma(Q)@l.P; k''; k']
\end{array}
$$

Example

- From

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

- We get

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

- When we undo the **out** we need to restore the **in**

CRμKlaim semantics

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

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

- M is complete and depends on k
- N_t: if the undone action is an **in**, we should release the tuple
- N_l: we should not consume the **roll** locality, unless created by the undone computation
- $N \lightning_k$: resources consumed by the computation should be released

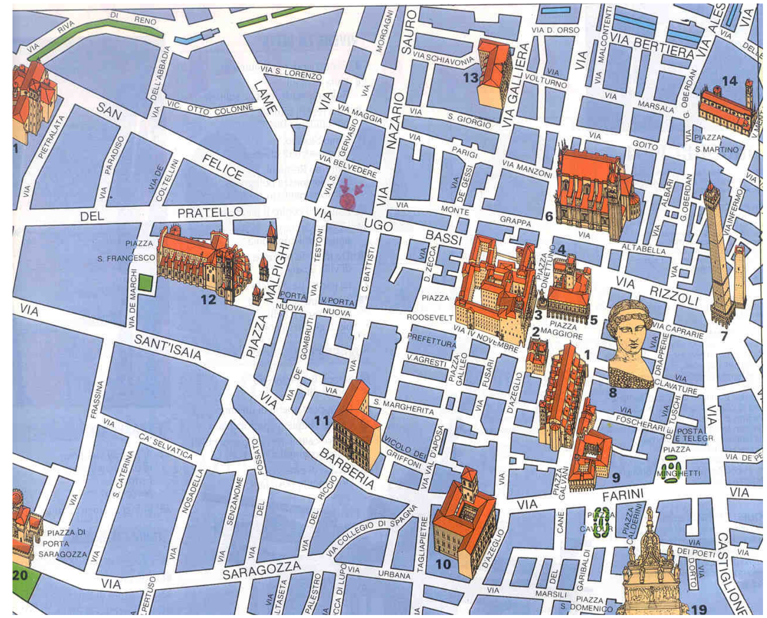
Results

- 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 μKlaim

- 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