Causal-Consistent Reversibility in a Tuple-Based Distributed Language

Ivan Lanese
Computer Science Department
University of Bologna/INRIA
Italy

Joint work with Elena Giachino, Claudio Mezzina and Francesco Tiezzi

Reversing different languages

- In principle we would like to take any language and automatically build its causal-consistent reversible extension
  - Not trivial
  - Try from Phillips et al. [FoSSaCS 2006] very limited
- Less ambitious: understand the interplay between causal-consistent reversibility and various constructs and language features
- We put our techniques at work on a language featuring some constructs not considered in past works

Map of the talk

- Klaim
- Uncontrolled reversibility
- The **roll** operator
- Conclusions

Map of the talk

- Klaim
- Uncontrolled reversibility
- The **roll** operator
- Conclusions

Klaim

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



μKlaim syntax

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

μKlaim semantics

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

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

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

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

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

Map of the talk

- Klaim
- Uncontrolled reversibility
- The roll operator
- Conclusions

Making μKlaim reversible

- We want to apply the technique used for HOπ to μKlaim
- We call RμKlaim the resulting language
- 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

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

RμKlaim semantics – tuple operators

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

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

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

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

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

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

RμKlaim semantics – distribution operators

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

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

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

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

Example

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

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

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

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

Properties

- The forward semantics of RμKlaim follows the semantics of μKlaim
- The Loop Lemma holds
- RμKlaim is causally consistent

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

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

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

Map of the talk

- Klaim
- Uncontrolled reversibility
- The **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

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

Example

- From

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

- We get

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

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

CRμKlaim semantics

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

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

- M is complete and depends on k
- N_t: if the undone action is an **in** I should release the tuple
- N_l: I 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

- Klaim
- Uncontrolled reversibility
- The 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
  - Behavioral theory
  - Alternatives
  - Encoding of the reversible language in the basic one
    - Would allow to exploit Klaim implementations
  - Low-level controlled semantics
- The killer application may be in the field of STM
- Relation between
  - Definition of concurrent transitions in uncontrolled reversibility
  - The causality relation used in controlled reversibility

End of talk

Thanks!

Questions?