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

<table>
<thead>
<tr>
<th>Nets</th>
<th>( N ) ::=</th>
<th>( 0 )</th>
<th>( l :: C )</th>
<th>( N_1 \parallel N_2 )</th>
<th>( (\nu l)N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>( C ) ::=</td>
<td>( {et} )</td>
<td>( P )</td>
<td>( C_1 \parallel C_2 )</td>
<td></td>
</tr>
<tr>
<td>Processes</td>
<td>( P ) ::=</td>
<td>( \text{nil} )</td>
<td>( a.P )</td>
<td>( P_1 \parallel P_2 )</td>
<td>( A )</td>
</tr>
<tr>
<td>Actions</td>
<td>( a ::= )</td>
<td>( \text{out}(t)@l )</td>
<td>( \text{in}(T)@l )</td>
<td>( \text{read}(T)@l )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{eval}(P)@l )</td>
<td>( \text{newloc}(l) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tuples</td>
<td>( t ::= )</td>
<td>( e )</td>
<td>( l )</td>
<td>( t_1, t_2 )</td>
<td></td>
</tr>
<tr>
<td>Evaluated tuples</td>
<td>( et ::= )</td>
<td>( v )</td>
<td>( l )</td>
<td>( et_1, et_2 )</td>
<td></td>
</tr>
<tr>
<td>Templates</td>
<td>( T ::= )</td>
<td>( e )</td>
<td>( l )</td>
<td>( !x )</td>
<td>( !u )</td>
</tr>
</tbody>
</table>
μKlaim semantics

\[
\begin{align*}
[t] &= et \\
\frac{l :: \text{out}(t)@l'.P \parallel l' :: \text{nil} \rightarrow l :: P \parallel l' :: \langle et \rangle}{(Out)} \\
\text{match}([T], et) &= \sigma \\
\frac{l :: \text{in}(T)@l'.P \parallel l' :: \langle et \rangle \rightarrow l :: P\sigma \parallel l' :: \text{nil}}{(In)} \\
\text{match}([T], et) &= \sigma \\
\frac{l :: \text{read}(T)@l'.P \parallel l' :: \langle et \rangle \rightarrow l :: P\sigma \parallel l' :: \langle et \rangle}{(Read)} \\
l :: \text{eval}(Q)@l'.P \parallel l' :: \text{nil} \rightarrow l :: P \parallel l' :: Q \quad (Eval) \\
l :: \text{newloc}(l').P \rightarrow (\nu l')(l :: P \parallel l' :: \text{nil}) \quad (New)
\end{align*}
\]
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 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) \( N \ ::= \ 0 \ | \ l :: C \ | \ l :: \text{empty} \ | \ N_1 || N_2 \ | \ (\nu z)N \)

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

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

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

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

\[ l :: k : \text{out}(t)@l'.P \parallel l' :: \text{empty} \xrightarrow{r} (\nu k', k'') (l :: k' : P \mid [k : \text{out}(t)@l''; k''; k'] \parallel l' :: k''' : \langle \varepsilon t \rangle) \] (Out)

\[ (\nu k'') (l :: k' : P \mid [k : \text{out}(t)@l''; k''; k'] \parallel l' :: k''' : \langle \varepsilon t \rangle) \sim_r l :: k : \text{out}(t)@l'.P \parallel l' :: \text{empty} \] (OutRev)

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

\[ l :: k : \text{in}(T)@l'.P \parallel l' :: h : \langle \varepsilon t \rangle \xrightarrow{r} (\nu k') (l :: k' : P\sigma \mid [k : \text{in}(T)@l'.P; h : \langle \varepsilon t \rangle; k'] \parallel l' :: \text{empty} \] (In)

\[ l :: k' : Q \mid [k : \text{in}(T)@l'.P; h : \langle \varepsilon t \rangle; k'] \parallel l' :: \text{empty} \sim_r l :: k : \text{in}(T)@l'.P \parallel l' :: h : \langle \varepsilon t \rangle \] (InRev)

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

\[ l :: k : \text{read}(T)@l'.P \parallel l' :: h : \langle \varepsilon t \rangle \xrightarrow{r} (\nu k') (l :: k' : P\sigma \mid [k : \text{read}(T)@l'.P; h; k'] \parallel l' :: h : \langle \varepsilon t \rangle \] (Read)

\[ l :: k' : Q \mid [k : \text{read}(T)@l'.P; h; k'] \parallel l' :: h : \langle \varepsilon t \rangle \sim_r l :: k : \text{read}(T)@l'.P \parallel l' :: h : \langle \varepsilon t \rangle \] (ReadRev)
RμKlaim semantics – distribution operators

\[
l :: k : \text{eval}(Q)@l'.P \parallel l' :: \text{empty} \leftrightarrow_r (\nu k', k'') (l :: k' : P | [k : \text{eval}(Q)@l'; k''; k'] \parallel l' :: k'' : Q) \quad (\text{Eval})
\]

\[
l :: k' : P | [k : \text{eval}(Q)@l'; k''; k'] \parallel l' :: k'' : Q \leadsto_r l :: k : \text{eval}(Q)@l'.P \parallel l' :: \text{empty} \quad (\text{EvalRev})
\]

\[
l :: k : \text{newloc}(l').P \leftrightarrow_r (\nu l') ( (\nu k') l :: k' : P | [k : \text{newloc}(l'); k'] \parallel l' :: \text{empty} ) \quad (\text{New})
\]

\[
(\nu l') (l :: k' : P | [k : \text{newloc}(l'); k'] \parallel l' :: \text{empty}) \leadsto_r l :: k : \text{newloc}(l').P \quad (\text{NewRev})
\]
Example

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

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

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

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

- The forward semantics of $R\mu$Klaim follows the semantics of $\mu$Klaim
- The Loop Lemma holds
- $R\mu$Klaim 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 $\lambda$ on memories

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

- **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

| Nets | N ::= 0 | l :: C | l :: empty | N₁ || N₂ | (νz)N |
|------|---------|---------|-------------|--------|--------|
| Components | C ::= k : ⟨et⟩ | k : P | C₁ | C₂ | μ | k₁ ≺ (k₂, k₃) |
| Processes | P ::= nil | a.P | P₁ | P₂ | A | roll(ι) |
| Actions | a ::= outᵣ(τ)@ℓ | inᵣ(T)@ℓ | readᵣ(T)@ℓ |
| | | | evalᵣ(P)@ℓ | newlocᵣ(l) |
| Memories | μ ::= [k : outᵣ(τ)@l.P; k''; k'] | [k : inᵣ(T)@l.P; h : ⟨et⟩; k'] |
| | | | [k : readᵣ(T)@l.P; h; k'] | [k : newlocᵣ(l).P; k'] | [k : evalᵣ(Q)@l.P; k''; k'] |
Example

- From

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

- We get

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

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

\[ l :: k : \text{newloc}_\gamma(l').P \rightarrow_c (vl') \ ( (vk') (l :: k' : P[k/\gamma] \ | \ [k : \text{newloc}_\gamma(l').P; k']) \ | \ l' :: \text{empty} ) \]  (New)

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

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

\[ N_l = 0 \text{ if } k <: M l, \text{ otherwise } N_l = l :: \text{empty} \]  (Roll)

- \( M \) is complete and depends on \( k \)
- \( N_t \): if the undone action is an \text{in} I should release the tuple
- \( N_l \): I should not consume the \text{roll} locality, unless created by the undone computation
- \( N \notin_k \): 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 $\mu$Klaim
- Two main features taken into account
  - Read dependences
  - Localities
Future work

- Part of $\text{HO}\pi$ theory not yet transported to $\mu$Klaim
  - 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?