Concurrent Flexible Reversibility

Ivan Lanese [1] Michael Lienhardt[2] **Claudio Antares Mezzina** [3]
Alan Schmitt[4] Jean-Bernard Stefani [4]

[1]Focus Team, University of Bologna/INRIA, Italy

[2]PPS Laboratory Paris

[3]**Fondazione Bruno Kessler Trento, Italy**

[4]INRIA France

March 21, 2013

ESOP 2013

Roadmap

1. Reversibility
2. A flexible reversible language: crollπ
3. Results

Undo manifestations

Language support for Recovery Oriented Computing (ROC)

- Application undo (e.g. Ctrl+Z)
- System undo (e.g. Windows Undo System Restore)
- Logs and checkpoints
- Transaction rollback
- Distributed rollback-recovery

What if we could undo every action ?

Claim

We want to use reversibility as a unifying framework to build dependable systems

Reversibility: Sequential Vs Concurrent

Reversibility

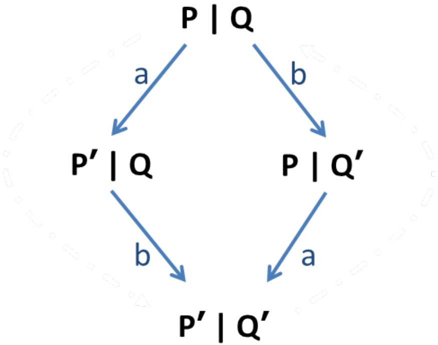
The possibility of executing a computation both in the standard, forward direction, and in the backward direction, going back to a past state

In a **sequential** setting, reversibility is simply undoing (recursively) the last action

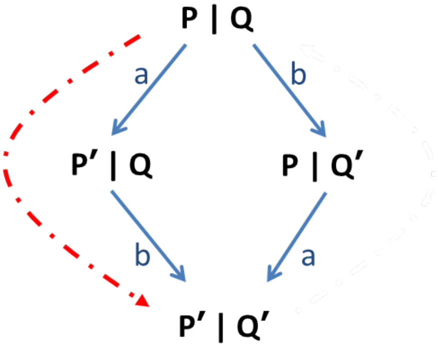
In a **concurrent** setting, one cannot simply undo the **last action**

- Not clear what is the last action
- Independent threads are reversed independently
- Causal dependencies should be respected
    - First undo the consequences and then the causes

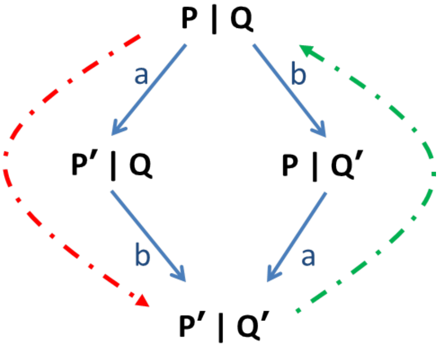
Concurrent Causal Reversibility



Concurrent Causal Reversibility



Concurrent Causal Reversibility



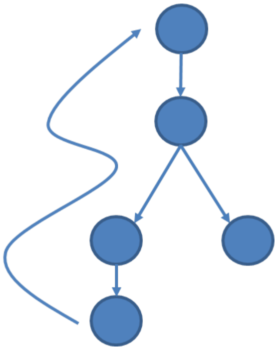
So far

Previous works on reversibility proposed:

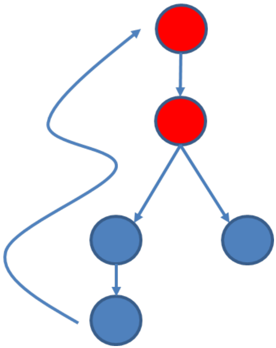
1. **uncontrolled** reversibility
   - show how to go back and forth
   - no hint on when to go forward or backward
   - more useful to understand basics of concurrent reversibility than as programming languages

2. **controlled** reversibility
   - backward computations are enabled in case of an error (when)
   - the computation should undo the events that caused the error (how far to go)
   - the computation that led to an error can be re-executed (may diverge)

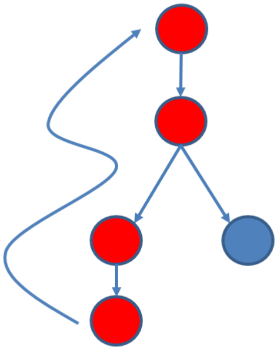
Controlled Reversibility: example



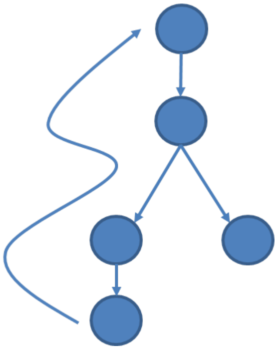
Controlled Reversibility: example



Controlled Reversibility: example



Controlled Reversibility: example



Flexible Reversibility

Problems

- How do I avoid repeating the same erroneous computations?
- How do I specify what to do after a backward computation?

Solution (our idea)

Mixing controlled reversibility and compensations

Flexible Reversibility

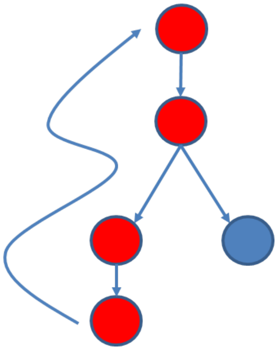
Problems

- How do I avoid repeating the same erroneous computations?
- How do I specify what to do after a backward computation?

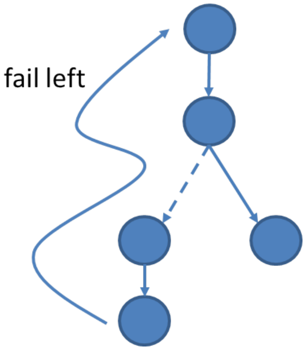
Solution (our idea)

Mixing controlled reversibility and compensations

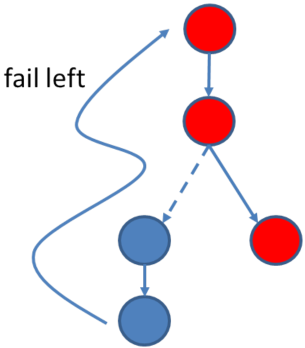
Flexible Reversibility: example



Flexible Reversibility: example



Flexible Reversibility: example



Roadmap

1. Reversibility
2. A flexible reversible language: crollπ
3. Results

HOπ

Syntax

$$
\begin{array}{rll}
P, Q ::= & a\langle P\rangle & \textit{message} \\
\mid & (a(X) \triangleright P) & \textit{trigger} \\
\mid & X & \textit{variable} \\
\mid & (P \mid Q) & \textit{parallel composition} \\
\mid & \nu a.\, P & \textit{new name} \\
\mid & \mathbf{0} & \textit{null process}
\end{array}
$$

$a \in \mathcal{N}$

- Message passing communications
- Higher Order communications (sent values are processes)

Ideas

To introduce reversibility in HOπ:

- Log each action (message receipt)
- Uniquely identify action participants

To control reversibility:

- Use of a specific primitive roll γ where γ is an action identifier

To specify compensations:

- Use messages with alternatives

Syntax: Processes

$$
\begin{array}{rll}
P, Q & \mid \; a\langle P\rangle \div C & \textit{message} \\
& \mid \; (a(X) \triangleright_{\gamma} P) & \textit{trigger} \\
& \mid \; \mathsf{roll}\ \gamma & \text{roll} \\
& \mid \; \mathsf{roll}\ k & \text{active roll} \\
& \mid \; X & \textit{variable} \\
& \mid \; (P \mid Q) & \textit{parallel composition} \\
& \mid \; \nu a.\, P & \textit{new name} \\
& \mid \; \mathbf{0} & \textit{null process} \\
\\
C ::= & \mathbf{0} & \text{empty alternative} \\
& \mid \; a\langle P\rangle \div \mathbf{0} & \text{message alternative}
\end{array}
$$

Syntax: Configurations

$$
\begin{array}{rll}
M, N ::= & & \textit{configurations} \\
& \mid\ k : P & \textit{thread} \\
& \mid\ [\mu; k] & \textit{memory} \\
& \mid\ k \prec (k_1, k_2) & \textit{connector} \\
& \mid\ \nu u.\, M & \textit{restriction} \\
& \mid\ (M \mid N) & \textit{parallel} \\
& \quad \mathbf{0} & \textit{null configuration}
\end{array}
$$

$$\mu ::= \ (k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q)$$

$$u \in \mathcal{I} \quad a \in \mathcal{N} \quad k \in \mathcal{K}$$

Syntax intuitions

- $k : P$ thread of computation (process) P uniquely identified by tag k
- $[(k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q); k]$ action identified by k
- roll k reverts all the effects of a memory (message receipt) k
- $k : a\langle P\rangle \div C$ alternative C to the message $a\langle P\rangle$
- $k \prec (k_1, k_2)$ thread k is divided into two sub-threads k_1 and k_2

Operational Semantics 1/2

$$(\text{Com}) \quad \frac{\mu = (k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q)}{(k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q) \rightarrow \nu k\,.(k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$(\text{TagP})\ k : P \mid Q \rightarrow \nu k_1\, k_2 .\, k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

- Communication side-effects:
  - tags the new instance of Q with the new key k
  - γ is replaced by k in the instance of Q
  - stores the configuration that generated the even k in a memory
- Partial order among process identifiers

Operational Semantics 1/2

$$(\textsc{Com})\quad \frac{\mu = (k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q)}{(k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q) \to \nu k\,.(k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$(\textsc{TagP})\ k : P \mid Q \to \nu k_1\, k_2 .\, k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

- Communication side-effects:
  - tags the new instance of Q with the new key k
  - γ is replaced by k in the instance of Q
  - stores the configuration that generated the even k in a memory
- Partial order among process identifiers

Operational Semantics 1/2

$$(\textsc{Com}) \quad \frac{\mu = (k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q)}{(k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q) \rightarrow \nu k\,.(k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$(\textsc{TagP})\ k : P \mid Q \rightarrow \nu k_1\, k_2 .\, k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

- Communication side-effects:
  - tags the new instance of Q with the new key k
  - γ is replaced by k in the instance of Q
  - stores the configuration that generated the even k in a memory
- Partial order among process identifiers

Operational Semantics 1/2

$$(\textsc{Com}) \quad \frac{\mu = (k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q)}{(k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q) \to \nu k\,.(k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$(\textsc{TagP})\ k : P \mid Q \to \nu k_1\, k_2 .\, k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

- Communication side-effects:
  - tags the new instance of Q with the new key k
  - γ is replaced by k in the instance of Q
  - stores the configuration that generated the even k in a memory
- Partial order among process identifiers

Operational Semantics 1/2

$$(\textsc{Com})\quad \frac{\mu = (k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \rhd_\gamma Q)}{(k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \rhd_\gamma Q) \to \nu k\,.(k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$(\textsc{TagP})\ k : P \mid Q \to \nu k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

- Communication side-effects:
  - tags the new instance of Q with the new key k
  - γ is replaced by k in the instance of Q
  - stores the configuration that generated the even k in a memory
- Partial order among process identifiers

Operational Semantics 2/2

$$(\text{S.ROLL}) \quad \frac{k <: N \qquad \texttt{complete}(N \mid [\mu; k] \mid (k_r : \mathsf{roll}\ k)) \qquad \mu' = \texttt{xtr}(\mu)}{N \mid [\mu; k] \mid (k_r : \mathsf{roll}\ k) \to \mu' \mid N \lightning_k}$$

How the rule works:

1. **collects** all the processes caused by k
2. **deletes** them and frees resources consumed by the computation to be rolled-back
3. **substitutes** the message in μ with its alternative ($\texttt{xtr}$ function)

$$\texttt{xtr}(k_1 : a\langle P_1 \rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q) = k_1 : C \mid (k_2 : a(X) \triangleright_\gamma Q)$$

Operational Semantics 2/2

$$\text{(S.ROLL)}\quad \frac{k <: N \qquad \texttt{complete}(N \mid [\mu; k] \mid (k_r : \mathsf{roll}\ k)) \qquad \mu' = \texttt{xtr}(\mu)}{N \mid [\mu; k] \mid (k_r : \mathsf{roll}\ k) \to \mu' \mid N \not\downarrow_k}$$

How the rule works:

1. **collects** all the processes caused by k
2. **deletes** them and frees resources consumed by the computation to be rolled-back
3. **substitutes** the message in μ with its alternative (xtr function)

$$\texttt{xtr}(k_1 : a\langle P_1\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q) = k_1 : C \mid (k_2 : a(X) \triangleright_\gamma Q)$$

Roadmap

Different Alternative Idioms

Simple messages as alternatives are powerful enough to express:

- Different kind of alternatives:
  - General alternatives: e.g. $a\langle P\rangle \div Q$
  - Triggers with alternatives: $(a(X) \triangleright_\gamma P) \div C$
- More sophisticated patterns:
  - Finite retry: $a\langle P\rangle \div_n C$ (try $a\langle P\rangle$ n times and then C)
  - Infinite retry: $a\langle P\rangle$

Some encodings in crollπ

$$(\!|a\langle P\rangle \div Q|\!)_{aa} = \nu c.\, a\langle (\!|P|\!)_{aa}\rangle \div c\langle (\!|Q|\!)_{aa}\rangle \div \mathbf{0} \mid c(X) \triangleright X$$

$$(\!|a\langle P\rangle|\!)_{er} = \nu t.\, Y \mid a\langle (\!|P|\!)_{er}\rangle \div t\langle Y\rangle \quad Y = t(Z) \triangleright Z \mid a\langle (\!|P|\!)_{er}\rangle \div t\langle Z\rangle$$

Encodings Correctness

$P \approx_c (\!|P|\!)_{aa}$ for any closed process P
$P \approx_c (\!|P|\!)_{er}$ for any closed process P

$\approx_c$ weak barbed congruence

Interacting Transactions

- A transaction is a computation that may:
  - **succeed**, making results permanent
  - **abort**, undoing all its effect
  - have a compensation to be executed upon abort
- Interacting transactions are allowed to interact with the environment during their execution.
  - no **isolation**

Hennessy interacting transactions

TransCCS

$$(\textsc{Com})\ \overline{a} \mid a.P \to P \qquad (\textsc{Co})\ [\![P \mid \mathsf{co}\, k \rhd_k Q]\!] \to P$$

$$(\textsc{Ab})\ [\![P \rhd_k Q]\!] \to Q \qquad (\textsc{Emb})\ [\![P \rhd_k Q]\!] \mid R \to [\![P \mid R \rhd_k Q \mid R]\!]$$

- $\to$ closed under structural congruence (π one) and under the contexts: $\bullet \mid P$, $[\![\bullet \rhd_k Q]\!]$ and $\nu a.\bullet$
- $[\![P \rhd_k Q]\!]$ transaction with name k, body P and compensation Q
- rule Emb allows an arbitrary process to get into the transactional scope
- abort is spontaneous, commit is explicit

Interacting transactions in crollπ

Encoding

$$\llparenthesis \nu a.\, P \rrparenthesis \;=\; \nu a.\, \llparenthesis P \rrparenthesis \qquad \llparenthesis P \mid Q \rrparenthesis \;=\; \llparenthesis P \rrparenthesis \mid \llparenthesis Q \rrparenthesis \qquad \llparenthesis \overline{a} \rrparenthesis \;=\; \overline{a}$$

$$\llparenthesis a.P \rrparenthesis \;=\; a \triangleright P \qquad \llparenthesis \mathsf{co}\; k \rrparenthesis \;=\; l(X) \triangleright \mathbf{0} \qquad \llparenthesis \mathbf{0} \rrparenthesis \;=\; \mathbf{0}$$

$$\llparenthesis [\![P \triangleright_l Q]\!] \rrparenthesis \;=\; [\nu l.\, \llparenthesis P \rrparenthesis \mid l\langle \mathsf{roll}\ \gamma \rangle \mid l(X) \triangleright X \,,\, \llparenthesis Q \rrparenthesis]_\gamma$$

$$[P \,,\, Q]_\gamma \;=\; \nu a, c.\, \overline{a} \div \overline{c} \div \mathbf{0} \mid (a \triangleright_\gamma P) \mid (c \triangleright Q)$$

- abort is modelled as rollback
- a transaction is started by a message with alternative
- compensation Q is started upon rollback

Results

Theorem (Encoding Correctness)

For each TransCCS process P, $P \;{}_{t}{\approx}_{\mathsf{croll}\pi}\; \nu k.\, k : (\!| P |\!)$

- ${}_{t}{\approx}_{\mathsf{croll}\pi}$ ad-hoc weak barbed bisimulation between transCCS processes and crollπ processes
- crollπ causality tracking mechanism is more precise than Hennessy embedding
- we avoid spurious rollbacks they have (thread independence)

Conclusion

- we presented a calculus with explicit rollback and core facilities for alternatives
- simple messages are enough to built more complex alternatives
- rollback and alternatives can encode transactional constructs
- messages with alternatives increase the expressive power of the calculus (not shown here)
- simple Maude interpreter (not shown here)
  - some primitive for errors handling (try/catch , stabilizer)
  - http://proton.inrialpes.fr/~mlienhar/croll-pi/implem/
- context lemma to help proofs (not shown here)

Future works

- try to encode more transactional constructs (e.g. STM)
- study relationships between alternatives and compensations in long-running transactions
- improve the Maude interpreter (e.g. garbage collection of unreachable memories)
- test the interpreter against more complex case studies