Controlling Reversibility in Higher-Order Pi*

Ivan Lanese[1], Claudio Antares Mezzina[2],
Alan Schmitt[2], and Jean-Bernard Stefani[2]

[1] University of Bologna & INRIA, Italy
[2] INRIA Grenoble-Rhône-Alpes, France

Abstract. We present in this paper a fine-grained rollback primitive for the higher-order π-calculus (HOπ), that builds on the reversibility apparatus of reversible HOπ [9]. The definition of a proper semantics for such a primitive is a surprisingly delicate matter because of the potential interferences between concurrent rollbacks. We define in this paper a high-level operational semantics which we prove sound and complete with respect to reversible HOπ backward reduction. We also define a lower-level distributed semantics, which is closer to an actual implementation of the rollback primitive, and we prove it to be fully abstract with respect to the high-level semantics.

1 Introduction

Motivation and contributions. Reversible computing, or related notions, can be found in many areas, including hardware design, program debugging, discrete-event simulation, biological modeling, and quantum computing (see [2] and the introduction of [10] for early surveys on reversible computing). Of particular interest is the application of reversibility to the study of programming abstractions for fault-tolerant systems. In particular, most fault tolerance schemes based on *system recovery* techniques [1], including rollback/recovery schemes and transaction abstractions, imply some form of undo. The ability to undo any single action in a reversible computation model provides an ideal setting to study, revisit, or imagine alternatives to these different schemes. This is in part the motivation behind the recent development of the reversible process calculi RCCS [4] and $\rho\pi$ [9], with [5] showing how a general notion of interactive transaction emerges from the introduction of irreversible (commit) actions in RCCS. However, these calculi provide very little in the way of controlling reversibility. The notion of irreversible action in RCCS only prevents a computation from rolling back past a certain point. Exploiting the low-level reversibility machinery available in these models of computation for fault-recovery purposes would require more extensive control on the reversal of actions, including *when* they can take place and *how far back* (along a past computation) they apply.

We present in this paper the study of a fine-grained rollback control primitive, where potentially every single step in a concurrent execution can be undone.

* Partly funded by the EU project FP7-231620 HATS, the ANR-2010-SEGI-013 project AEOLUS, and the ANR-2010-BLAN-0305-01 project PiCoq.

Specifically, we introduce a rollback construct for an asynchronous higher-order π-calculus (HOπ [11]), building on the machinery of $\rho\pi$, the reversible higher-order π-calculus presented in [9]. We chose HOπ as our substrate because we find it a convenient starting point for studying distributed programming models with inherently higher-order features such as dynamic code update, which we aim to combine with abstractions for system recovery and fault tolerance. Surprisingly, finding a suitable definition for a fine-grained rollback construct in HOπ is more difficult than one may think, even with the help of the reversible machinery from [9]. There are two main difficulties. The first one is in actually pinning down the intended effect of a rollback operation, especially in presence of concurrent rollbacks. The second one is in finding a suitably distributed semantics for rollback, dealing only with local information and not relying on complex atomic transitions involving a potentially unbounded number of distinct processes.

We show in this paper how to deal with these difficulties by making the following contributions: (i) we define a high-level operational semantics for a rollback construct in an asynchronous higher-order π-calculus, which we prove *maximally permissive*, in the sense that it makes reachable all past states in a given computation; (ii) we present a low-level semantics for the proposed rollback construct which can be understood as a fully distributed variant of our high-level semantics, and we prove it to be *fully abstract* with respect to the high-level one.

Paper Outline. In Section 2, we informally present our rollback calculus, which we call roll-π, and illustrate the difficulties that may arise in defining a fine-grained rollback primitive. In Section 3, we formalize roll-π and its high-level operational semantics. In Section 4, we present a distributed operational semantics for roll-π, and we prove that it is fully abstract with respect to the high-level one. Section 5 discusses related work and concludes the paper. The interested reader can find proofs of the main results in [8].

2 Informal presentation

To define roll-π and its rollback construct, we rely on the same support for reversibility as in $\rho\pi$ [9]. Let us review briefly its basic mechanisms.

Reversibility in $\rho\pi$. We attach to each process P a unique tag κ (either simple, written as k, or composite, denoted as $\langle h_i, \tilde{h} \rangle \cdot k$). The uniqueness of tags for processes is achieved thanks to the following structural congruence rule that defines how tags and parallel composition commute.

$$k : \prod_{i=1}^{n} \tau_i \equiv \nu \tilde{h}.\ \prod_{i=1}^{n} (\langle h_i, \tilde{h} \rangle \cdot k : \tau_i) \quad \text{with } \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2 \qquad (1)$$

In equation (1), $\prod_{i=1}^{n}$ is n-ary parallel composition and ν is the restriction operator, both standard from the π-calculus. Each *thread* τ_i is either a message, of the form

$a\langle P\rangle$ (where a is a channel name), or a receiver process (also called a *trigger*), of the form $a(X) \triangleright P$. A *forward* computation step (or forward reduction step, noted with arrow $\twoheadrightarrow$) consists of the reception of a message by a receiver process, and takes the following form (note that $\rho\pi$ is an asynchronous calculus).

$$(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q) \twoheadrightarrow \nu k.\, k : Q\{^{P}/_{X}\} \mid [M; k] \tag{2}$$

In this forward step, κ_1 identifies a thread consisting of message $a\langle P\rangle$ on channel a, and κ_2 identifies a thread consisting of a trigger process $a(X) \triangleright Q$ that expects a message on channel a. The result of the message input yields, as usual, an instance $Q\{^{P}/_{X}\}$ of the body of the trigger Q with the formal parameter X instantiated by the received value, i.e., the process P ($\rho\pi$ is higher-order). Message input also has two side effects: (i) the tagging of the newly created process $Q\{^{P}/_{X}\}$ by a fresh tag k, and (ii) the creation of a memory $[M; k]$, which records the original two threads, $M = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q)$, together with tag k.

In $\rho\pi$, a forward reduction step such as (2) above is systematically associated with a backward reduction step (noted with arrow $\rightsquigarrow$) of the form:

$$(k : Q) \mid [M; k] \rightsquigarrow M \tag{3}$$

which undoes the communication between threads κ_1 and κ_2. When necessary to avoid confusion, we will add a $\rho\pi$ subscript to arrows representing $\rho\pi$ reductions.

Given a configuration M, the set of memories present in M provides us with an ordering $:>$ between tags in M that reflects their causal dependency: if memory $[\kappa_1 : P_1 \mid \kappa_2 : P_2; k]$ occurs in M, then $\kappa_i > k$. Also, $k > \langle h_i, \tilde{h}\rangle \cdot k$, and we define the relation $:>$ as the reflexive and transitive closure of the $>$ relation. We say that tag κ has κ' as a causal antecedent if $\kappa' :> \kappa$.

Reversibility in roll-π. The notion of memory introduced in $\rho\pi$ is in some way a checkpoint, uniquely identified by its tag. In roll-π, we exploit this intuition to introduce an explicit form of backward reduction. Specifically, backward reduction is not allowed by default as in $\rho\pi$, but has to be triggered by an instruction of the form roll k, whose intent is that the current computation be rolled back to a state just prior to the creation of the memory bearing the tag k. To be able to form an instruction of the form roll k, one needs a way to pass the knowledge of a memory tag to a process. This is achieved in roll-π by adding a bound variable to each trigger process, which now takes the form $a(X) \triangleright_\gamma P$, where γ is the tag variable bound by the trigger construct and whose scope is P. A forward reduction step in roll-π therefore is:

$$(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow \nu k.\, k : Q\{^{P,k}/_{X,\gamma}\} \mid [M; k] \tag{4}$$

where the only difference with (2) lies in the fact that the newly created tag k is passed as an argument to the trigger body Q. We write $a(X) \triangleright P$ in place of $a(X) \triangleright_\gamma P$ if the tag variable γ does not appear free in P.

Now, given the above intent for the rollback primitive roll, how does one define its operational semantics? As hinted at in the introduction, this is actually

a subtler affair than one may expect. A big difference with $\rho\pi$, where communication steps are undone one by one, is that the k in $\mathsf{roll}\ k$ may refer to a communication step far in the past. So the idea behind a $\mathsf{roll}\ k$ is to restore the content of a memory $[M;k]$ and to delete all its forward history. Consider the following attempt at a rule for roll:

$$(\textsc{Naive})\quad \frac{N \blacktriangleright k \qquad \mathtt{complete}(N \mid [M;k] \mid (\kappa : \mathsf{roll}\ k))}{N \mid [M;k] \mid (\kappa : \mathsf{roll}\ k) \rightsquigarrow M \mid N \lightning_k}$$

The different predicates and the $\lightning$ operator used in the rule are defined formally in the next section, but an informal explanation should be enough to understand how the rule works. Briefly, the assertion $N \blacktriangleright k$ states that all the active threads and memories in N bear tags κ that have k as causal antecedent, i.e., $k :> \kappa$ (N does not contain unrelated processes). The assertion $\mathtt{complete}(M_c)$ states that configuration M_c gathers all the threads (inside or outside memories) whose tags have as a causal antecedent the tag of a memory in M_c itself, i.e., if a memory in M_c is of the form $[M';k']$ (the communication M' created a process tagged with k'), then a process or a memory containing a process tagged with k' has to be in M_c (M_c contains every related process). The premises of rule Naive thus asserts that the configuration $M_c = N \mid [M;k] \mid \kappa : \mathsf{roll}\ k$, on the left hand side of the reduction in the conclusion of the rule, gathers all (and only) the threads and memories which have originated from the process tagged by k, itself created by the interaction of the message and trigger recorded in M. Being complete, M_c is thus ready to be rolled back and replaced by the configuration M which is at its origin. Rolling back M_c has another effect, noted as $N \lightning_k$ in the right hand side of the conclusion, which is to release from memories those messages or triggers which do not have k as a causal antecedent, but which participated in communications with causal descendants of k.

For instance, the configuration $M_0 = M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y)$, where $M_1 = (\kappa_0 : a\langle P\rangle) \mid (\kappa_1 : a(X) \triangleright_\gamma c\langle \mathsf{roll}\ \gamma\rangle)$, has the following forward reductions (where $M_2 = (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y)$):

$$\begin{aligned} M_0 &\twoheadrightarrow \nu k.\, [M_1;k] \mid (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y) \\ &\twoheadrightarrow \nu k, l.\, [M_1;k] \mid [M_2;l] \mid (l : \mathsf{roll}\ k) = M_3 \end{aligned}$$

Applying rule Naive (and structural congruence, defined later) on M_3 we get:

$$M_3 \rightsquigarrow M_1 \mid [M_2;l]\lightning_k = M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y) = M_0$$

where $(\kappa_2 : c(Y) \triangleright_\delta Y)$ is released from memory $[M_2;l]$ because it does not have k as a causal antecedent.

Rule Naive looks reasonable enough, but difficulties arise when concurrent rollbacks are taken into account. Consider the following configuration:

$$M = (k_1 : \tau_1) \mid (k_2 : a\langle \mathbf{0}\rangle) \mid (k_3 : \tau_3) \mid (k_4 : b\langle \mathbf{0}\rangle)$$

where[1] $\tau_1 = a(X) \triangleright_\gamma d\langle \mathbf{0}\rangle \mid (c(Y) \triangleright \mathsf{roll}\ \gamma)$ and $\tau_3 = b(Z) \triangleright_\delta c\langle \mathbf{0}\rangle \mid (d(U) \triangleright \mathsf{roll}\ \delta)$.

[1] We assume parallel composition has precedence over trigger.

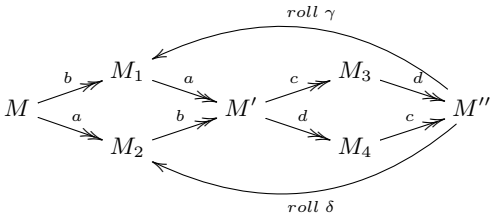

Fig. 1. Concurrent rollback anomaly

The most interesting reductions of M are depicted in Figure 1. Forward reductions are labelled by the name of the channel used for communication, while backward reductions are labelled by the executed roll instruction. The main processes and short-cuts are detailed below:

$$M_1 = \nu l_2, h_3, h_4.\, \sigma_1 \mid [\sigma_2; l_2] \mid (\kappa_3 : c\langle \mathbf{0} \rangle) \mid (\kappa_4 : \tau_4)$$
$$M_2 = \nu l_1, h_1, h_2.\, [\sigma_1; l_1] \mid (\kappa_1 : d\langle \mathbf{0} \rangle) \mid (\kappa_2 : \tau_2) \mid \sigma_2$$
$$M'' = \nu l_1 \dots l_4, h_1 \dots h_4.\, [\sigma_1; l_1] \mid [\sigma_2; l_2] \mid [\sigma_3; l_3] \mid [\sigma_4; l_4] \mid (l_3 : \mathsf{roll}\ l_1) \mid (l_4 : \mathsf{roll}\ l_2)$$

$$\sigma_1 = (k_1 : \tau_1) \mid (k_2 : a\langle \mathbf{0} \rangle) \quad \sigma_2 = (k_3 : \tau_3) \mid (k_4 : b\langle \mathbf{0} \rangle) \quad \tau_2 = c(Y) \triangleright \mathsf{roll}\ l_1$$
$$\sigma_3 = (\kappa_2 : \tau_2) \mid (\kappa_3 : c\langle \mathbf{0} \rangle) \quad \sigma_4 = (\kappa_1 : d\langle \mathbf{0} \rangle) \mid (\kappa_4 : \tau_4) \quad \tau_4 = d(U) \triangleright \mathsf{roll}\ l_2$$

The anomaly here is that there is no way from M_1 or M_2 to get back to the original configuration M, despite the fact that M'' has two roll instructions which would seem sufficient to undo all the reductions which lead from M to M''. Note that M_1 and M_2 are configurations which could both have been reached from M. Thus rule Naive is not unsound, but incomplete or insufficiently permissive, at least with respect to what is possible in $\rho\pi$: if we were to undo actions in M'' step by step, using $\rho\pi$'s backward reductions, we could definitely reach all of M, M_1, and M_2. Note that the higher-order aspects do not matter here.

The main motivation to have a complete rule comes from the fact that, in an abstract semantics, one wants to be as liberal as possible, and not unduly restrict implementations. If we were to pick the Naive rule as our semantics for rollback, then a correct implementation would have to enforce the same restrictions with respect to states reachable from backward reductions, restrictions which, in the case of rule Naive, are both complex to characterize (in terms of conflicting rollbacks) and quite artificial since they do not correspond to any clear execution policy. In the next section, we present a *maximally permissive* semantics for rollback, using $\rho\pi$ as our benchmark for completeness.

3 The roll-π calculus and its high-level semantics

3.1 Syntax

Names, keys, and variables. We assume the existence of the following denumerable infinite mutually disjoint sets: the set $\mathcal{N}$ of *names*, the set $\mathcal{K}$ of *keys*, the

$$
\begin{aligned}
P, Q &::= \mathbf{0} \mid X \mid \nu a.\, P \mid (P \mid Q) \mid a\langle P \rangle \mid a(X) \triangleright_\gamma P \mid \mathsf{roll}\ k \mid \mathsf{roll}\ \gamma \\
M, N &::= \mathbf{0} \mid \nu u.\, M \mid (M \mid N) \mid \kappa : P \mid [\mu; k] \mid [\mu; k]^\bullet \\
\kappa &::= k \mid \langle h, \tilde{h} \rangle \cdot k \\
\mu &::= ((\kappa_1 : a\langle P \rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)) \\
& a \in \mathcal{N} \quad X \in \mathcal{V}_\mathcal{P} \quad \gamma \in \mathcal{V}_\mathcal{K} \quad u \in \mathcal{I} \quad h, k \in \mathcal{K}
\end{aligned}
$$

Fig. 2. Syntax of roll-π

set $\mathcal{V}_\mathcal{K}$ of *tag variables*, and the set $\mathcal{V}_\mathcal{P}$ of process variables. The set $\mathcal{I} = \mathcal{N} \cup \mathcal{K}$ is called the set of *identifiers*. We note $\mathbb{N}$ the set of natural integers. We let (together with their decorated variants): a, b, c range over $\mathcal{N}$; h, k, l range over $\mathcal{K}$; u, v, w range over $\mathcal{I}$; δ, γ range over $\mathcal{V}_\mathcal{K}$; X, Y, Z range over $\mathcal{V}_\mathcal{P}$. We note $\tilde{u}$ a finite set of identifiers $\{u_1, \ldots, u_n\}$.

Syntax. The syntax of the roll-π calculus is given in Figure 2 (we often add balanced parenthesis around roll-π terms to disambiguate them). *Processes*, given by the P, Q productions in Figure 2, are the standard processes of the asynchronous higher-order π-calculus, except for the presence of the roll primitive and the extra bound tag variable in triggers. A trigger in roll-π takes the form $a(X) \triangleright_\gamma P$, which allows the receipt of a message of the form $a\langle Q \rangle$ on channel a, and the capture of the tag of the receipt event with tag variable γ.

Processes in roll-π cannot directly execute, only *configurations* can. *Configurations* in roll-π are given by the M, N productions in Figure 2. A configuration is built up from *tagged processes* and *memories*.

In a tagged process $\kappa : P$ the tag κ is either a single key k or a pair of the form $\langle h, \tilde{h} \rangle \cdot k$, where $\tilde{h}$ is a set of keys with $h \in \tilde{h}$. A tag serves as an identifier for a process. As in $\rho\pi$ [9], tags and memories help capture the flow of causality in a computation.

A *memory* is a configuration of the form $[\mu; k]$, which keeps track of the fact that a configuration μ was reached during execution, that triggered the launch of a process tagged with the fresh tag k. In a memory $[\mu; k]$, we call μ the *configuration part* of the memory, and k the *tag* of the memory. The configuration part $\mu = (\kappa_1 : a\langle P \rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)$ of a memory records the message $a\langle P \rangle$ and the trigger $a(X) \triangleright_\gamma Q$ involved in the message receipt, together with their respective thread tags κ_1, κ_2. A *marked memory* is a configuration of the form $[\mu; k]^\bullet$, which just serves to indicate that a rollback operation targeting this memory has been initiated.

We note $\mathcal{P}$ the set of roll-π processes, and $\mathcal{C}$ the set of roll-π configurations. We call *agent* an element of the set $\mathcal{A} = \mathcal{P} \cup \mathcal{C}$. We let (together with their decorated variants) P, Q, R range over $\mathcal{P}$; L, M, N range over $\mathcal{C}$; and A, B, C range over $\mathcal{A}$. We call *thread*, a process that is either a message $a\langle P \rangle$, a trigger $a(X) \triangleright_\gamma P$, or a rollback instruction roll k. We let τ and its decorated variants range over threads.

Free identifiers and free variables. Notions of free identifiers and free variables in roll-π are usual. Constructs with binders are of the following forms: $\nu a.\, P$ binds the name a with scope P; $\nu u.\, M$ binds the identifier u with scope M; and $a(X) \triangleright_\gamma P$ binds the process variable X and the tag variable γ with scope P. We note $\mathtt{fn}(P)$, $\mathtt{fn}(M)$, and $\mathtt{fn}(\kappa)$ the set of free names, free identifiers, and free keys, respectively, of process P, of configuration M, and of tag κ. Note in particular that $\mathtt{fn}(\kappa : P) = \mathtt{fn}(\kappa) \cup \mathtt{fn}(P)$, $\mathtt{fn}(\mathsf{roll}\ k) = \{k\}$, $\mathtt{fn}(k) = \{k\}$ and $\mathtt{fn}(\langle h, \tilde{h}\rangle \cdot k) = \tilde{h} \cup \{k\}$. We say that a process P or a configuration M is *closed* if it has no free (process or tag) variable. We note $\mathcal{P}^{cl}$, $\mathcal{C}^{cl}$ and $\mathcal{A}^{cl}$ the sets of closed processes, configurations, and agents, respectively.

Initial and consistent configurations. Not all configurations allowed by the syntax in Figure 2 are meaningful. For instance, in a memory $[\mu; k]$, tags occurring in the configuration part μ must be different from the key k; if a tagged process $\kappa_1 : \mathsf{roll}\ k$ occurs in a configuration M, we expect a memory $[\mu; k]$ to occur in M as well. In the rest of the paper, we only will be considering well-formed, or *consistent*, closed configurations. A configuration is consistent if it can be derived using the rules of the calculus from an *initial* configuration. A configuration is initial if it does not contain memories, all the tags are distinct and simple (i.e., of the form k), and the argument of each roll is bound by a trigger.

We do not give here a syntactic characterization of consistent configurations as it is not essential to understand the developments in this paper (the interested reader may find some more details in [9], where a syntactic characterization of $\rho\pi$ consistent configurations is provided).

Remark 1. We have no construct for replicated processes or guarded choice in roll-π: as in HOπ, these can easily be encoded.

Remark 2. In the remainder of the paper, we adopt *Barendregt's Variable Convention*: if terms $t_1, \ldots, t_n$ occur in a certain context (e.g., definition, proof), then in these terms all bound identifiers and variables are chosen to be different from the free ones.

3.2 Operational semantics

The operational semantics of the roll-π calculus is defined via a reduction relation $\rightarrow$, which is a binary relation over closed configurations ($\rightarrow \subset \mathcal{C}^{cl} \times \mathcal{C}^{cl}$), and a structural congruence relation $\equiv$, which is a binary relation over processes and configurations ($\equiv \subset \mathcal{P}^2 \cup \mathcal{C}^2$). We define *evaluation contexts* as "configurations with a hole $\cdot$", given by the following grammar:

$$\mathbb{E} ::= \cdot \mid (M \mid \mathbb{E}) \mid \nu u.\, \mathbb{E}$$

General contexts $\mathbb{C}$ are just processes or configurations with a hole $\cdot$. A *congruence* on processes or configurations is an equivalence relation $\mathcal{R}$ that is closed for general contexts: $P\,\mathcal{R}\,Q \implies \mathbb{C}[P]\,\mathcal{R}\,\mathbb{C}[Q]$ or $M\,\mathcal{R}\,N \implies \mathbb{C}[M]\,\mathcal{R}\,\mathbb{C}[N]$.

The relation $\equiv$ is defined as the smallest congruence on processes and configurations that satisfies the rules in Figure 3. We note $t =_\alpha t'$ when terms

$$\text{(E.ParC)}\ A \mid B \equiv B \mid A \qquad \text{(E.ParA)}\ A \mid (B \mid C) \equiv (A \mid B) \mid C$$

$$\text{(E.ParN)}\ A \mid \mathbf{0} \equiv A \qquad \text{(E.NewN)}\ \nu u.\, \mathbf{0} \equiv \mathbf{0} \qquad \text{(E.NewC)}\ \nu u.\, \nu v.\, A \equiv \nu v.\, \nu u.\, A$$

$$\text{(E.NewP)}\ (\nu u.\, A) \mid B \equiv \nu u.\, (A \mid B) \qquad \text{(E.}\alpha\text{)}\ A =_\alpha B \implies A \equiv B$$

$$\text{(E.TagN)}\ \kappa : \nu a.\, P \equiv \nu a.\, \kappa : P$$

$$\text{(E.TagP)}\ k : \prod_{i=1}^{n} \tau_i \equiv \nu \tilde{h}.\, \prod_{i=1}^{n} (\langle h_i, \tilde{h} \rangle \cdot k : \tau_i) \quad \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2$$

Fig. 3. Structural congruence for roll-π

t, t' are equal modulo α-conversion. If $\tilde{u} = \{u_1, \ldots, u_n\}$, then $\nu\tilde{u}.\, A$ stands for $\nu u_1.\, \ldots \nu u_n.\, A$. We note $\prod_{i=1}^{n} A_i$ for $A_1 \mid \ldots \mid A_n$ (there is no need to indicate how the latter expression is parenthesized because the parallel operator is associative by rule E.ParA). In rule E.TagP, processes τ_i are threads. Recall the use of the variable convention in these rules: for instance, in the rule $(\nu u.\, A) \mid B \equiv \nu u.\, (A \mid B)$ the variable convention makes implicit the condition $u \notin \mathtt{fn}(B)$. The structural congruence rules are the usual rules for the π-calculus (E.ParC to E.α) without the rule dealing with replication, and with the addition of two new rules dealing with tags: E.TagN and E.TagP. Rule E.TagN is a scope extrusion rule to push restrictions to the top level. Rule E.TagP allows to generate unique tags for each thread in a configuration. An easy induction on the structure of terms provides us with a kind of normal form for configurations (by convention $\prod_{i \in I} A_i = \mathbf{0}$ if $I = \emptyset$, and $[\mu; k]^\circ$ stands for $[\mu; k]$ or $[\mu; k]^\bullet$):

Lemma 1 (Thread normal form). *For any configuration M, we have*

$$M \equiv \nu\tilde{u}.\, \prod_{i \in I} (\kappa_i : \rho_i) \mid \prod_{j \in J} [\mu_j; k_j]^\circ$$

with $\rho_i = \mathbf{0}$, $\rho_i = \mathsf{roll}\ k_i$, $\rho_i = a_i\langle P_i \rangle$, or $\rho_i = a_i(X_i) \triangleright_{\gamma_i} P_i$.

We say that a binary relation $\mathcal{R}$ on closed configurations is *evaluation-closed* if it satisfies the inference rules:

$$\text{(R.Ctx)}\ \frac{M\ \mathcal{R}\ N}{\mathbb{E}[M]\ \mathcal{R}\ \mathbb{E}[N]} \qquad \text{(R.Eqv)}\ \frac{M \equiv M' \qquad M'\ \mathcal{R}\ N' \qquad N' \equiv N}{M\ \mathcal{R}\ N}$$

The reduction relation $\to$ is defined as the union of two relations, the *forward* reduction relation $\twoheadrightarrow$ and the *backward* reduction relation $\rightsquigarrow$: $\to\ =\ \twoheadrightarrow \cup \rightsquigarrow$. Relations $\twoheadrightarrow$ and $\rightsquigarrow$ are defined to be the smallest evaluation-closed binary relations on closed configurations satisfying the rules in Figure 4 (note again the use of the variable convention: in rule H.Com the key k is fresh).

The rule for forward reduction H.Com is the standard communication rule of the higher-order π-calculus with three side effects: (i) the creation of a new

$$\text{(H.Com)}\quad \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow \nu k.\, (k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$\text{(H.Start)}\quad (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k] \rightsquigarrow (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k]^\bullet$$

$$\text{(H.Roll)}\quad \frac{N \blacktriangleright k \qquad \mathtt{complete}(N \mid [\mu; k])}{N \mid [\mu; k]^\bullet \rightsquigarrow \mu \mid N \not\downarrow_k}$$

Fig. 4. Reduction rules for roll-π

memory to record the configuration that gave rise to it; (ii) the tagging of the continuation of the message receipt with the fresh key k; (iii) the passing of the newly created tag k as a parameter to the newly launched instance of the trigger's body Q.

Backward reduction is subject to the rules H.Roll and H.Start. Rule H.Roll is similar to rule Naive defined in the previous section, except that it relies on the presence of a marked memory instead of on the presence of the process $\kappa : \mathsf{roll}\ k$ to roll back a given configuration. Rule H.Start just marks a memory to enable rollback.

The definition of rule H.Roll exploits several predicates and relations which we define below.

Definition 1 (Causal dependence). *Let M be a configuration and let T_M be the set of tags occurring in M. The binary relation $>_M$ on T_M is defined as the smallest relation satisfying the following clauses:*

- $k >_M \langle h_i, \tilde{h}\rangle \cdot k$;
- $\kappa' >_M k$ *if* κ' *occurs in* μ *for some memory* $[\mu; k]^\circ$ *that occurs in* M.

The causal dependence relation $:>_M$ is the reflexive and transitive closure of $>_M$.

Relation $\kappa :>_M \kappa'$ reads "κ is a causal antecedent of κ' according to M". When configuration M is clear from the context, we write $\kappa :> \kappa'$ for $\kappa :>_M \kappa'$.

Definition 2 (κ dependence). *Let $M \equiv \nu\tilde{u}.\, \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J}[\mu_j; \kappa_j]^\circ$. Configuration M is κ-dependent, written $M \blacktriangleright \kappa$, if $\forall i \in I \cup J,\ \kappa :>_M \kappa_i$.*

We now define the *projection* operation on configurations $M \not\downarrow_\kappa$, that captures the parallel composition of all tagged processes that do not depend on κ occurring in memories in M.

Definition 3 (Projection). *Let $M \equiv \nu\tilde{u}.\, \prod_{i\in I}(\kappa_i : \rho_i) \mid \prod_{j\in J}[\mu_j; \kappa_j]^\circ$, with $\mu_j = \kappa'_j : R_j \mid \kappa''_j : T_j$. Then:*

$$M \not\downarrow_\kappa = \nu\tilde{u}.\, (\prod_{j'\in J'} \kappa'_{j'} : R_{j'}) \mid (\prod_{j''\in J''} \kappa''_{j''} : T_{j''})$$

where $J' = \{j \in J \mid \kappa :\not> \kappa'_j\}$ and $J'' = \{j \in J \mid \kappa :\not> \kappa''_j\}$.

Finally we define the notion of *complete* configuration, used in the premise of rule H.ROLL.

Definition 4 (Complete configuration). *A configuration M* contains *a tagged process $\kappa : P$, written $\kappa : P \in M$, if $M \equiv \nu\tilde{u}.\,(\kappa : P) \mid N$ or $M \equiv \nu\tilde{u}.\,[\kappa : P \mid \kappa_1 : Q; k]^\circ \mid N$.*

A configuration M is complete*, noted* $\texttt{complete}(M)$*, if for each memory $[\mu; k]^\circ$ that occurs in M, one of the following holds:*

1. *There exists a process P such that $k : P \in M$.*
2. *There is $\tilde{h}$ such that for each $h_i \in \tilde{h}$ there exists a process P_i such that $\langle h_i, \tilde{h}\rangle \cdot k : P_i \in M$.*

Barbed bisimulation. The operational semantics of the roll-π calculus is completed classically by the definition of a contextual equivalence between configurations, which takes the form of a barbed congruence. We first define observables in configurations. We say that name a is *observable in configuration M*, noted $M \downarrow_a$, if $M \equiv \nu\tilde{u}.\,(\kappa : a\langle P\rangle) \mid N$, with $a \notin \tilde{u}$. Keys are not observable: this is because they are just an internal device used to support reversibility. We note $\Rightarrow$, $\twoheadrightarrow^*$, $\rightsquigarrow^*$ the reflexive and transitive closures of $\rightarrow$, $\twoheadrightarrow$, and $\rightsquigarrow$, respectively.

One of the aims of this paper is to define a low-level semantics for roll-π, and show that it is equivalent to the high-level one. We want to use weak barbed congruence for this purpose. Thus we need a definition of barbed congruence able to relate roll-π configurations executed under different semantics. These semantics will also rely on different runtime syntaxes. Thus, we define a family of relations, each labeled by the semantics to be used on the left and right components of its elements. We also label sets of configurations with the corresponding semantics, thus highlighting that the corresponding runtime syntax has to be included. However, contexts do not include runtime syntax, since we never add contexts at runtime.

Definition 5 (Barbed bisimulation and congruence). *A relation ${}_{s1}\mathcal{R}_{s2} \subseteq \mathcal{C}^{cl}_{s1} \times \mathcal{C}^{cl}_{s2}$ on closed consistent configurations is a* strong *(resp.* weak*)* barbed simulation *if whenever $M\ {}_{s1}\mathcal{R}_{s2}\ N$*

- *$M \downarrow_a$ implies $N \downarrow_a$ (resp. $N \Rightarrow_{s2}\downarrow_a$)*
- *$M \rightarrow_{s1} M'$ implies $N \rightarrow_{s2} N'$, with $M'\,{}_{s1}\mathcal{R}_{s2} N'$ (resp. $N \Rightarrow_{s2} N'$ with $M'\,{}_{s1}\mathcal{R}_{s2} N'$)*

A relation ${}_{s1}\mathcal{R}_{s2} \subseteq \mathcal{C}^{cl}_{s1} \times \mathcal{C}^{cl}_{s2}$ is a strong *(resp.* weak*)* barbed bisimulation *if ${}_{s1}\mathcal{R}_{s2}$ and $({}_{s1}\mathcal{R}_{s2})^{-1}$ are strong (resp. weak) barbed simulations. We call* strong *(resp.* weak*)* barbed bisimilarity *and note ${}_{s1}\sim_{s2}$ (resp. ${}_{s1}\approx_{s2}$) the largest strong (resp. weak) barbed bisimulation with respect to semantics $s1$ and $s2$.*

We say that two configurations M and N are strong *(resp.* weak*)* barbed congruent*, written ${}_{s1}\sim^c_{s2}$ (resp. ${}_{s1}\approx^c_{s2}$), if for each roll-π context $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are consistent, then $\mathbb{C}[M]\,{}_{s1}\sim_{s2}\,\mathbb{C}[N]$ (resp. $\mathbb{C}[M]\,{}_{s1}\approx_{s2}\,\mathbb{C}[N]$).*

3.3 Soundness and completeness of backward reduction in roll-π

We present in this section a Loop Theorem, that establishes the soundness of backward reduction in roll-π, and we prove the completeness (or maximal permissiveness) of backward reduction in roll-π.

Theorem 1 (Loop Theorem - Soundness of backward reduction). *For any (consistent) configurations M and M' with no marked memories, if $M \rightsquigarrow^* M'$, then $M' \twoheadrightarrow^* M$.*

To state the completeness result for backward reduction in roll-π, we define a family of functions $\phi_e : \mathcal{C}_{\text{roll-}\pi} \to \mathcal{C}_{\rho\pi}$, where $e \in \mathcal{N}$, mapping a roll-π configuration to a $\rho\pi$ configuration. Function ϕ_e is defined by induction as follows:

$$\begin{array}{ccc}
\phi_e(\nu u.\, A) = \nu u.\, \phi_e(A) & \phi_e(A \mid B) = \phi_e(A) \mid \phi_e(B) & \phi_e(\kappa : P) = \kappa : \phi_e(P) \\
\phi_e([\mu; k]^\circ) = [\phi_e(\mu); k] & \phi_e(\mathbf{0}) = \mathbf{0} & \phi_e(X) = X \\
\phi_e(\mathsf{roll}\ k) = e\langle \mathbf{0} \rangle & \phi_e(\mathsf{roll}\ \gamma) = e\langle \mathbf{0} \rangle & \phi_e(a\langle P \rangle) = a\langle \phi_e(P) \rangle \\
 & \phi_e(a(X) \rhd_\gamma P) = a(X) \rhd \phi_e(P) &
\end{array}$$

Note that roll instructions are transformed not into $\mathbf{0}$ but into a thread $e\langle \mathbf{0} \rangle$: this is to ensure a consistent roll-π configuration is transformed into a consistent $\rho\pi$ configuration (recall that $\mathbf{0}$ is not a thread, thus it may be collected by structural congruence and there would be no thread corresponding to the roll k process).

We now state that roll-π is maximally permissive: any subset of roll primitives in evaluation context may successfully be executed, unlike in the naive example of Section 2. Let $M = \nu \tilde{u}.\, [\mu; k] \mid (k : P) \mid N$ be a $\rho\pi$ configuration and $S = \{k_1, \ldots, k_n\}$ a set of keys. We note $M \rightsquigarrow_S M'$ if $M \rightsquigarrow_{\rho\pi} M'$, $M' = \nu \tilde{u}.\, \mu \mid N$, and $k_i :> k$ for some $k_i \in S$ (here k is the key of the memory $[\mu; k]$ consumed by the reduction). If $M' \not\rightsquigarrow_S$, we say that M' is *final with respect to* S. We note $\rightsquigarrow_S^*$ the reflexive and transitive closure of $\rightsquigarrow_S$. We assume here that reductions are name-preserving, i.e., existing keys are not α-converted (cf. [9] for a discussion on the topic).

Theorem 2 (Completeness of backward reduction). *Let M be a (consistent) roll-π configuration such that $M \equiv \nu \tilde{u}.\, \prod_{i=1}^{n} \kappa_i : \mathsf{roll}\ k_i \mid M_1$, let $S = \{k_1, \ldots, k_n\}$, and let $e \in \mathcal{N} \setminus \mathtt{fn}(M)$. Then for all $T \subseteq S$, if $\phi_e(M) \rightsquigarrow_T^* N$ and N is final with respect to T, there exists M' such that $N = \phi_e(M')$, and $M \rightsquigarrow_{\text{roll-}\pi}^* M'$.*

4 A distributed semantics for roll-π

The semantics defined in the previous section captures the behavior of rollback, but its H.Roll rule specifies an atomic action involving a configuration with an unbounded number of processes and relies on global checks on this configuration, for verifying that it is complete and κ-dependent. This makes it arduous to implement, especially in a distributed setting.

$$\text{(L.Com)}\quad \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow_{LL} \nu k.\, (k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$\text{(L.Start)}\quad (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k] \rightsquigarrow_{LL} (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k]^\bullet \mid \mathsf{rl}\ k$$

$$\text{(L.Span)}\quad \mathsf{rl}\ \kappa_1 \mid [\kappa_1 : P \mid M; k]^\circ \rightsquigarrow_{LL} [\lfloor \kappa_1 : P \rfloor \mid M; k]^\circ \mid \mathsf{rl}\ k$$

$$\text{(L.Branch)}\quad \frac{\langle h_i, \tilde{h}\rangle \cdot k \text{ occurs in } M}{\mathsf{rl}\ k \mid M \rightsquigarrow_{LL} \prod_{h_i \in \tilde{h}} \mathsf{rl}\ \langle h_i, \tilde{h}\rangle \cdot k \mid M}$$

$$\text{(L.Up)}\quad \mathsf{rl}\ \kappa_1 \mid (\kappa_1 : P) \rightsquigarrow_{LL} \lfloor \kappa_1 : P \rfloor \qquad \text{(L.Stop)}\quad [\mu; k]^\circ \mid \lfloor k : P \rfloor \rightsquigarrow_{LL} \mu$$

Fig. 5. Reduction rules for LL

$$\text{(E.Gb1)}\quad \nu k.\, \mathsf{rl}\ k \equiv_{LL} 0 \qquad \text{(E.Gb2)}\quad \nu k.\, \prod_{h_i \in \tilde{h}} \mathsf{rl}\ \langle h_i, \tilde{h}\rangle \cdot k \equiv_{LL} 0$$

$$\text{(E.TagPFr)}\ \lfloor k : \prod_{i=1}^{n} \tau_i \rfloor \equiv_{LL} \nu \tilde{h}.\, \prod_{i=1}^{n} \lfloor (\langle h_i, \tilde{h}\rangle \cdot k : \tau_i) \rfloor \quad \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2$$

Fig. 6. Additional structural laws for LL

We thus present in this section a low-level (written LL) semantics, where the conditions above are verified incrementally by relying on the exchange of rl notifications. We show that the LL semantics captures the same intuition as the one introduced in Section 3 by proving that given a (consistent) configuration, its behaviors under the two semantics are weak barbed congruent according to Definition 5.

To avoid confusion between the two semantics, we use a subscript LL to identify all the elements (reductions, structural congruence, ...) referred to the low-level semantics presented here, and HL (for high-level) for the semantics described in Section 3.

The LL semantics $\rightarrow_{LL}$ of roll-π is defined as for the HL one (cf. Section 3.2), as $\rightarrow_{LL} = \twoheadrightarrow_{LL} \cup \rightsquigarrow_{LL}$, where relations $\twoheadrightarrow_{LL}$ and $\rightsquigarrow_{LL}$ are defined to be the smallest evaluation-closed binary relations on closed LL configurations satisfying the rules in Figure 5. The notion of structural congruence used in the definition of evaluation-closed is here the smallest congruence on LL processes and configurations that satisfies the rules in Figure 3 and in Figure 6.

LL configurations differ from HL configurations in two aspects. First, tagged processes (inside or outside memories) can be frozen, denoted $\lfloor \kappa : P \rfloor$, to indicate that they are participating to a rollback (rollback is no longer atomic). Second,

LL configurations include notifications of the form $\mathsf{rl}\ \kappa$, used to notify a tagged process with key κ to enter a rollback.

Let us describe the LL rules. Communication rule L.Com is as before. The main idea for rollback is that when a memory pointed by a roll is marked (rule L.Start), a notification $\mathsf{rl}\ k$ is generated. This notification is propagated by rules L.Span and L.Branch. Rule L.Span also freezes threads inside memories, specifying that they will be eventually removed by the rollback. Rule L.Branch (where the predicate "κ occurs in M" means that either $M = \kappa : P$ or $M = [\mu; k']^\circ$ with $\kappa : P \in M$) is used when the target configuration has been split into multiple threads: a notification has to be sent to each of them. Rule L.Up is similar to L.Span, but it applies to tagged processes outside memories. It also stops the propagation of the rl notification. The main idea is that by using rules L.Span, L.Branch, and L.Up one is able to tag all the causal descendants of a marked memory. Finally, rule L.Stop rolls back a single computation step by removing a frozen process and freeing the content of the memory created with it. In the LL semantics a rollback request is thus executed incrementally, while it was atomic in the HL semantics (rule H.Roll). The LL semantics also exploits an extended structural congruence, adding axioms E.Gb1 and E.Gb2 to garbage collect rl notifications when they are no more needed, and extending axiom E.TagP to deal with frozen threads (axiom E.TagPFr).

We now show an example to clarify the semantics (each reduction is labeled by the name of the axiom used to derive it). Let $M_0 = M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y)$, where $M_1 = (\kappa_0 : a\langle P\rangle) \mid (\kappa_1 : a(X) \triangleright_\gamma c\langle \mathsf{roll}\ \gamma\rangle)$. We have:

$$
\begin{aligned}
M_0 &\twoheadrightarrow \nu k.\, [M_1; k] \mid (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y) \\
\text{(L.Com)} \quad &\twoheadrightarrow \nu k, l.\, [M_1; k] \mid [M_2; l] \mid (l : \mathsf{roll}\ k) \\
\text{(L.Start)} \quad &\rightsquigarrow \nu k, l.\, [M_1; k]^\bullet \mid [M_2; l] \mid (l : \mathsf{roll}\ k) \mid \mathsf{rl}\ k \\
\text{(L.Span)} \quad &\rightsquigarrow \nu k, l.\, [M_1; k]^\bullet \mid [M_2'; l] \mid (l : \mathsf{roll}\ k) \mid \mathsf{rl}\ l \\
\text{(L.Up)} \quad &\rightsquigarrow \nu k, l.\, [M_1; k]^\bullet \mid [M_2'; l] \mid \lfloor (l : \mathsf{roll}\ k) \rfloor \\
\text{(L.Stop)} \quad &\rightsquigarrow \nu k.\, [M_1; k]^\bullet \mid M_2' \\
\text{(L.Stop)} \quad &\rightsquigarrow M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y)
\end{aligned}
$$

where:

$$
M_2 = (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y) \quad M_2' = \lfloor (k : c\langle \mathsf{roll}\ k\rangle) \rfloor \mid (\kappa_2 : c(Y) \triangleright_\delta Y)
$$

One can see that the rollback operation starts with the application of the rule L.Start, whose effects are (i) to mark the memory aimed by a roll process, and (ii) to generate a notification $\mathsf{rl}\ k$ to freeze its continuation. Since the continuation of the memory $[M_1; k]$ is contained in the memory $[M_2; l]$ then the rule L.Span is applied. So, the part of the memory containing the tag k gets frozen and a freeze notification $\mathsf{rl}\ l$ is generated. The notification eventually reaches the process $l : \mathsf{roll}\ k$ and freezes it (rule L.Up). Now, since there exists a memory whose continuation is a frozen process, we can apply the rule L.Stop, and free the configuration part of the memory (M_2'). Again, we have that the continuation

of $[M_1; k]$ is a frozen process and by applying the rule L.Stop we can free the configuration M_1, obtaining the initial configuration. In general, a rollback of a step whose memory is tagged by k is performed by executing a top-down visit of its causal descendants, freezing them, followed by a bottom-up visit undoing the steps one at the time.

We can now state the correspondence result between the two semantics.

Theorem 3 (Correspondence between HL and LL). *For each roll-π HL consistent configuration M, $M \,{}_{HL}{\approx}^{c}_{LL}\, M$.*

Proof. The proof is quite long and technical, and relies on a several additional semantics used as intermediate steps from HL to LL. It can be found in [8]. □

This result can be easily formulated as full abstraction. In fact, the encoding j from HL configurations to LL configurations defined by the injection (HL configurations are a subset of LL configurations) is fully abstract.

Corollary 1 (Full abstraction). *Let j be the injection from HL (consistent) configurations to LL configurations and let M, N be two HL configurations. Then we have $j(M) \,{}_{LL}{\approx}^{c}_{LL}\, j(N)$ iff $M \,{}_{HL}{\approx}^{c}_{HL}\, N$.*

Proof. From Theorem 3 we have $M \,{}_{HL}{\approx}^{c}_{LL}\, j(M)$ and $N \,{}_{HL}{\approx}^{c}_{LL}\, j(N)$. The thesis follows by transitivity. □

The results above ensure that the loss of atomicity in rollback preserves the reachability of configurations yet does not make undesired configurations reachable.

5 Related work and conclusion

We have introduced in this paper a fine-grained undo capability for the asynchronous higher-order π-calculus, in the form of a rollback primitive. We present a simple but non-trivial high-level semantics for rollback, and we prove it both sound (rolling back brings a concurrent program back to a state that is a proper antecedent of the current one) and complete (rolling back can reach all antecedent states of the current one). We also present a lower-level distributed semantics for rollback, which we prove to be fully abstract with respect to the high-level one. The reversibility apparatus we exploit to support our rollback primitive is directly taken from our reversible HOπ calculus [9].

Undo or rollback capabilities in programming languages have been the subject of numerous previous works and we do not have the space to review them here; see [10] for an early survey in the sequential setting. Among the recent works that have considered undo or rollback capabilities for concurrent program execution, we can single out [3] where logging primitives are coupled with a notion of process group to serve as a basis for defining transaction abstractions, [12] which introduces a checkpoint abstraction for functional programs, and [7] which extends the actor model with constructs to create globally-consistent

checkpoints. Compared to these works, our rollback primitive brings immediate benefits: it provides a general semantics for undo operations which is not provided in [3]; thanks to the fine-grained causality tracking implied by our reversible substrate, our roll-π calculus does not suffer from uncontrolled cascading rollbacks (domino effect) which may arise with [12], and, in contrast to [7], provides a built-in guarantee that, in failure-free computations, rollback is always possible and reaches a consistent state (soundness of backward reduction).

Our low-level semantics for rollback, being a first refinement towards an implementation, is certainly related to distributed checkpoint and rollback schemes, in particular to the causal logging schemes discussed in the survey [6]. A thorough analysis of this relationship must be left for further study, however, as it requires a proper modeling of site and communication failures, as well as an explicit model for persistent data.

References

1. A. Avizienis, J.C. Laprie, B. Randell, and C.E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1), 2004.
2. C.H. Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 32(1), 1988.
3. T. Chothia and D. Duggan. Abstractions for fault-tolerant global computing. *Theor. Comput. Sci.*, 322(3), 2004.
4. V. Danos and J. Krivine. Reversible communicating systems. In *Proc. of CONCUR'04*, volume 3170 of *LNCS*. Springer, 2004.
5. V. Danos and J. Krivine. Transactions in RCCS. In *Proc. of CONCUR'05*, volume 3653 of *LNCS*. Springer, 2005.
6. E.N. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3), 2002.
7. J. Field and C.A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *Proc. of POPL'05*. ACM, 2005.
8. I. Lanese, C.A. Mezzina, A. Schmitt, and J.B. Stefani. Controlling reversibility in higher-order pi (TR). `http://www.cs.unibo.it/~lanese/publications/fulltext/TR-rollpi.pdf.gz`.
9. I. Lanese, C.A. Mezzina, and J.B. Stefani. Reversing higher-order pi. In *Proc. of CONCUR 2010*, volume 6269 of *LNCS*. Springer, 2010.
10. G.B. Leeman. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8(1), 1986.
11. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, University of Edinburgh, 1992.
12. L. Ziarek and S. Jagannathan. Lightweight checkpointing for concurrent ML. *J. Funct. Program.*, 20(2), 2010.