Reversible π calculus

Ioana Cristescu

Training period at PPS, Université Paris Diderot

Abstract. We propose a reversible process algebra, that implements backtracking to π calculus. Rπ, the introduced calculus, is based on RCCS, a reversible CCS that attaches memories to processes. There are two main difficult problems that arise when one wants a reversible π calculus. These are due to substitution and scope extrusion. Their forward and backwards rules have to be symmetric and the semantics of the calculus, compositional.

1 Introduction

Reversibility and concurrency

Reversible computations in a concurrent setting facilitates the implementation of fault-recovery methods, aids in debugging, fault diagnosis and analysis. It can also be used in modeling transactions for reliable systems.

By reversibility we understand the possibility of undoing a series of actions and returning back to a previous version of our computation. In a sequential setting, this can be achieved by backtracking on the computation trace. From this trace, one can derive the order in which to undo the actions and the previous states of the computation.

In a distributed environment, we regard an action to be the receiving or sending of a message. The communicating parties are called *processes*. We can consider the computation's state from two points of view: the global view, as seen by the entire system, and a local one, specific to each process. When one process decides to backtrack, the reached state has to be a valid one for the entire system. Unlike the sequential case, fixing an order for the actions and ensuring it is preserved both forward and backwards is too strict. A process can decide to backtrack but it should not be able to affect independent processes. Concurrent processes have interleaved their actions in the forward trace, but as they are independent, one should not force them to mimic the same order going backwards. A backward computation trace is considered correct if it corresponds to a possible forward one. Therefore we have to determine the dependencies within a concurrent system. We say that dependent actions are in a *causal* relation: t is a cause for t' if the occurrence of t is a necessary condition for the occurrence of t'.

CCS and the π calculus

CCS [Mil82] is a mathematical model for distributed computing. The basic entity used is the *name* of a channel. A process can send or receive through a channel, or two processes can interact if they share a name. When a process performs an action, we represent it as a transition $P \xrightarrow{\alpha} P'$, where α is the action and P' is the process obtained. The behavior of a process can then be described using a set of transitions rules, called a labeled transition system (LTS).

We have the following process constructors in CCS: prefixing, parallel composition, sum and restriction. The first adds a prefix to the process, either as an input (a) or as an output ($\overline{a}$). The transition rule for an action is $\alpha.P \xrightarrow{\alpha} P$, where P is called the *continuation*. The parallel composition allows processes to interact. For example, let $\overline{a}.b$ and $a.\overline{c}$, be two processes that can communicate using channel a. Their synchronization, denoted with $\overline{a}.b \mid a.\overline{c} \xrightarrow{\tau} b \mid \overline{c}$, forms the new process $b \mid \overline{c}$, that can do an input on b or an output on c. Each component in a parallel composition is called a *thread*. The sum stands for nondeterministic choice: $a.P + b.Q$ can either perform a and become P or it can perform b and become Q. We can also have private names, denoted by the restriction operator ν. In a process $\nu y P$, y is a name unknown to the environment. All such names are *bound*. If the name is not bound, we say it is *free*. We call νy a binder for y as y can be α-converted in order to ensure it is indeed private.

The π calculus [SW01] is based on CCS except that the channels are no longer used only for synchronization, but can also carry information. Therefore the processes interaction consists in exchanging names.

$$\text{IN}\quad x(z).P + Q \xrightarrow{x(z)} P \qquad \text{OUT}\quad \overline{x}\langle y\rangle.P + Q \xrightarrow{\overline{x}\langle y\rangle} P \qquad \text{COM}\quad \frac{P \xrightarrow{\overline{x}\langle y\rangle} P' \qquad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}}$$

Note that receiving a name implies a substitution in the continuation. For example the process $\overline{a}\langle y\rangle \mid a(x).x(t)$ can do the following transition:

$$\frac{\overline{a}\langle y\rangle \xrightarrow{\overline{a}\langle y\rangle} 0 \qquad a(x).x(t) \xrightarrow{a(x)} x(t)}{\overline{a}\langle y\rangle \mid a(x).x(t) \xrightarrow{\tau} 0 \mid y(t)}$$

As in CCS, we have the set of bound names of a process R, denoted by $\text{bn}(R)$. The set contains the names used with the restriction ($y \in \text{bn}(\nu y P)$) and the names received through an input ($x \in \text{bn}(a(x).P)$). The remaining names are called free and denoted with $\text{fn}(R)$. Another difference between the two calculi consists in the usage of private names.

$$\text{NEW}\quad \frac{P \xrightarrow{\gamma}_\pi P'}{\nu y P \xrightarrow{\gamma}_\pi \nu y P'} \text{ if } y \notin \gamma \qquad \text{OPEN}\quad \frac{P \xrightarrow{\overline{x}\langle y\rangle} P'}{\nu y P \xrightarrow{\overline{x}(\nu y)} P'} \qquad \text{CLOSE}\quad \frac{P \xrightarrow{\overline{x}(\nu y)} P' \qquad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} \nu y(P' \mid Q')} y \notin \text{fn}(Q)/\{z\}$$

In a process $\nu y(\overline{a}\langle y\rangle \mid y(t).P)$, the transition on y is not allowed. Instead, the process has first to *extrude* the bound name, that is send it to the environment $\nu y(\overline{a}\langle y\rangle.P) \xrightarrow{\overline{a}\langle y\rangle} P$. The binder ν disappears from the process structure, hence y is now a free name of P. This signals that the channel y is no longer private and it can be used to communicate with the environment. In case of a synchronization in which a bound name is exchanged (in rule CLOSE), the binder reappears in the structure of the process. In order to support this behavior, we distinguish between the output of a free name and the one of a bound name, by denoting the latter with $\overline{x}(\nu y)$.

The condition on rule CLOSE ensures that bound names are distinct one from another and from any free name. We have to verify this condition also when a communication involving it passes through a context, in particular in the case of a parallel composition. On a transition label we define the sets of free or bound name as follows: on $x(z)$ or $\overline{x}(\nu y)$, x is free and z, y are bound while in $\overline{x}\langle y\rangle$ both names are free.

$$\text{PAR}\quad \frac{P \xrightarrow{\gamma}_\pi P'}{P \mid Q \xrightarrow{\gamma}_\pi P' \mid Q} \text{ if } \text{bn}(\gamma) \cap \text{fn}(Q) = \emptyset$$

A feature of both calculi is their compositional semantics. We can take two processes, so far completely independent and form a new one, that is correct both syntactically and semantically. As it will be shown, compositionality is not easily preserved when we want to implement reversibility and it is a feature we strive to maintain in the new calculus.

Causality

Causality [BS95], [DP95] is also a feature for debugging. When a problem occurs in the computation, knowing on which previous computations the "failing" action depended on makes it easier to track the bug.

Defining a causal relation is necessary in order to implement reversibility. Let us consider some examples. In $a(x).b(t).P$ the transition on $a(x)$ is the cause of the transition on $b(t)$. Similarly, in $a(x).b(t).P \mid \overline{a}\langle c\rangle$, the transition τ has to occur before the one on b. This type of causality results from the structure of the process and is present also in CCS. Consider now the process $\nu y(\overline{a}\langle y\rangle.P \mid y(t).Q)$. Even though the two processes $\overline{a}\langle y\rangle.P$, $y(t).Q$ are in a parallel composition, the actions $\overline{a}\langle y\rangle$ and $y(t)$ are not concurrent. Since the name y is private, for a communication to occur on channel y, the name has first to be send to the environment through channel a.

Reversibility in π calculus

A process in π can perform one of the following actions: it can either send or receive a free name, it can synchronize with another process or it can extrude a bound name. In order to support reversibility for these actions we need to record information available at the forward transition. We store them in a memory that we attach to each process. For instance, we can have the transition $\langle\rangle \rhd \overline{x}\langle y\rangle.P \longrightarrow \langle\star, \overline{x}\langle y\rangle\rangle \rhd P$, where the process has initially an empty memory and after sending y, it stores in the memory the performed action. As each thread has its own memory the computation is still distributed. The memory is also used to infer an identifier for each thread. In a parallel composition we differentiate the left hand side and the right side. This naming scheme is sufficient to yield an unique identifier per thread. For example in the process $\langle\rangle \rhd (P \mid P' \mid P'')$ we can derive the following identifiers: $\langle 1\rangle \rhd (P \mid P') \mid \langle 2\rangle \rhd P''$ and $\langle 1.1\rangle \rhd P \mid \langle 2.1\rangle \rhd P' \mid \langle 2\rangle \rhd P''$.

If the action performed is an input, we have to take care to also store the substitutions. Consider the process $a(x).(x(t) \mid y(z))$, that can receive y on channel x and become $y(t) \mid y(z)$. Backtracking to the process $a(x).(x(t) \mid x(z))$ is incorrect. By storing the substitution and not applying them directly on the process we have $\langle\rangle \rhd a(x).(x(t) \mid y(z)) \longrightarrow \langle\star, a[y/x]\rangle \rhd x(t) \mid y(z)$, from which we can easily recover the original process.

Backtracking a synchronization requires the participation of both parties involved, similar to a forward synchronization. Consequently, when a thread backtracks its synchronizing partner is identified and forced to roll back as well. The processes store not only which actions they performed but also the identifiers of their partner. As an example, consider the transition $\langle 2, a[y/x]\rangle.\langle 1\rangle \rhd P \mid \langle 1, \overline{a}\langle y\rangle\rangle.\langle 2\rangle \rhd 0 \xrightarrow{\tau^-} \langle 1\rangle \rhd a(x).P \mid \langle 2\rangle \rhd \overline{a}\langle y\rangle$. If we allow only one of the threads to backtrack then we reach an incorrect process.

When a process representing a summation does a reduction, then we also have to record the part of the summation that normally disappears, in order to be able to retrieve all the previous possible choices. For example in $\alpha.P + \beta.Q \longrightarrow P$, when P wants to backtrack it has to restore the residual $\beta.Q$. Therefore the transition in this case is $\langle\rangle \rhd \alpha.P + \beta.Q \longrightarrow \langle\star, \alpha, \beta.Q\rangle \rhd P$.

We can see from the examples above that the first type of causality is encoded in the structure of the memory. A process $\langle\star, \overline{a}\langle y\rangle\rangle.\langle\star, \overline{b}\langle y\rangle\rangle \rhd 0$ cannot do the backward transition on b before the one on a. This memory is inferred from the execution of the process $\overline{b}\langle y\rangle.\overline{a}\langle y\rangle$, where the output on b is a cause for the output on a. The second type of causality however cannot be represented in the memory's structure without limiting the distributed aspect of the calculus. The process $\nu y(\overline{a}\langle y\rangle.P \mid y(t).Q)$ can become $\langle\star, \overline{a}\langle y\rangle\rangle \rhd P \mid \langle\star, y(t)\rangle \rhd Q$. If the first thread is allowed to go back then we reach the process $\langle\rangle \rhd \overline{a}\langle y\rangle.P \mid \langle\star, y(t)\rangle \rhd Q$, which is invalid since the extrusion on the channel a is the cause for the further communications on channel y. The causal order $\xrightarrow{\overline{a}\langle y\rangle}\xrightarrow{y(t)}$ should be preserved when going backwards.

Another example is the process $R = \nu y(\overline{a}\langle y\rangle \mid \overline{b}\langle y\rangle \mid y(t))$ which can become $R' = \langle\star, \overline{a}\langle y\rangle\rangle \rhd 0 \mid \langle\star, \overline{b}\langle y\rangle\rangle \rhd 0 \mid \langle\star, y(t)\rangle \rhd 0$. Then, depending on the current execution trace, either the transition on a or the one on b is the cause of the transition on y. The common approach is to fix one of them (either the first or the last one to send y) as the cause, and to ensure that the two appear in the same order throughout the computation. Suppose that we have the trace $R \xrightarrow{\overline{a}\langle y\rangle}\xrightarrow{\overline{b}\langle y\rangle}\xrightarrow{y(t)} R'$. Then if we allow to backtrack on a or on b we reach the processes $\overline{a}\langle y\rangle \mid \langle\star, \overline{b}\langle y\rangle\rangle \rhd 0 \mid \langle\star, y(t)\rangle \rhd 0$ or $\langle\star, \overline{a}\langle y\rangle\rangle \rhd 0 \mid \overline{b}\langle y\rangle \mid \langle\star, y(t)\rangle \rhd 0$, respectively. Both are correct, since an extruder of y has still to backtrack. Therefore our approach is to allow a roll back of a transition that outputs a bound name as long as there still are extruders that have yet to backtrack. Hence, in our example we allow either of the transitions on a or on b to backtrack. We will see later on how backtracking for such processes is implemented. Intuitively, distributed computations cannot be undone without a sort of global memory restricting the behavior of the threads that initially were in the scope of a νy.

The contributions of this training period

We worked on defining a reversible π calculus semantics that is compositional and that handles the causality relation between processes. Our work can be summarized in the following:

– we provide a labeled transition system, similar to the one of π calculus. Our semantics ensures backtracking and is compositional with respect to the extrusion of bound names. Its main features are that it handles name substitution and scope extrusion;
– as a consequence, we also obtained a new causal semantics for π, that is more flexible than previous causal semantics of the calculus;
– we show a correspondence between our reversible π and the standard π calculus.

Structure of this document
In section 2 we present the calculus that served as a basis for reversible π. It is similar to CCS, due to the lack of space we present only RCCS. Then, in section 3 we move to our calculus. First, we make the transition between CCS and π and then show our additions to π. In section 4 the causal semantics is explained and compared to other causal semantics of π calculus. As Rπ is a calculus that allows π calculus processes to backtrack, we show the correspondence between the two calculi in section 5. We conclude with section 6 in which we summarize the report and discuss, as possible future work, the connection with event structures.

2 Reversible CCS

RCCS, introduced in [DK04], [DK05], is a reversible CCS. We start with formally introducing the structure of the processes and the memories. We also define the fork structure, congruence and other notions that are inherited in Rπ.

In order to ease notations, henceforth we omit all trailing 0's either in the process or in the memory. For instance a process of the form $< \phi, \alpha, 0 > \rhd a.0$ will be denoted as $< \phi, \alpha > \rhd a$.

The grammar of processes
We distinguish between two types of processes: the CCS ones, denoted by P, Q, and the monitored processes also called RCCS processes, ranged over R, S. We call them simply processes when it is clear from the context of which we refer to. A CCS process is built using a summation, a parallel composition or the restriction operator.

$$P, Q ::= \alpha.P + Q \mid (P \mid Q) \mid \nu y P \mid 0 \quad \text{(CCS processes)}$$

Both processes in a parallel composition $P \mid Q$ can execute, independently one of another while in a summation $P + Q$ either P or Q executes depending on a choice made by the context. In $\nu y P$ the context cannot use y as it is known only to P.

Only the monitored processes are runnable. They are obtained by attaching a memory to each thread of a CCS process.

$$R, S ::= (m \rhd P \mid R) \mid \nu y R \mid \emptyset \quad \text{(RCCS processes)}$$

The actions a process can do are either an input, an output or a synchronization.

$$\alpha ::= x \mid \overline{x} \mid \tau \quad \text{(actions)}$$

In the case of a synchronization both threads have to backtrack, in order to obtain a consistent global state of the process. When a thread wants to backtrack it has to signal it to its synchronizing partner. To this end, a unique identifier for each thread is needed. It is enough to distinguish each thread within a parallel composition. The left part of a parallel composition is denoted by $\langle 1 \rangle$ and the right part with $\langle 2 \rangle$. We use either the identifier of the synchronizing thread or a $\star$, if it is an unsynchronized communication.

$$\phi ::= \star \mid (1, 2)^* \quad \text{(identifiers)}$$

The memory is used, during the forward execution, to store information needed later to backtrack. For each action α, an entry is added to the memory containing the performed action and whether the action was part of a synchronization or not. For a summation, the residual process is also stored.

$$m ::= \langle\rangle \mid \langle 1 \rangle.m \mid \langle 2 \rangle.m \mid \langle \phi, \alpha, P \rangle.m \quad \text{(memories)}$$

The thread's identifiers are retrieved from their memory using the function $\Phi(m)$.

Definition 1 *The fork structure of a memory, denoted by $\Phi(m)$, is defined inductively on a memory m:*

$$\Phi(\langle\rangle) = \langle\rangle \qquad \Phi(\langle\phi, a, Q\rangle.m) = \Phi(m) \qquad \Phi(\langle i\rangle.m) = \langle i\rangle.\Phi(m)$$

For example, consider the memory $\langle\star, a, Q\rangle.m$. It says that the thread performed an input on a, without synchronizing, and that the initial process is of the form $m \rhd a.P + Q$. In the case of a synchronization, the memories of the two threads can look like $\langle 2, a\rangle.\langle 1\rangle \rhd P \mid \langle 1, \overline{a}\rangle.\langle 2\rangle \rhd Q$ with the initial process $\langle\rangle \rhd (a.P \mid \overline{a}.Q)$.

Structural congruence

Within a process we can rewrite the terms while keeping the resulting process equivalent to the original one. This is formalized using a relation called structural congruence.

Definition 2 *Structural congruence, denoted by $\equiv$, is the smallest equivalence relation that is a congruence, contains α-conversion, and which satisfies the following laws:*

$$m \rhd P_1 \mid P_2 \equiv \langle 1\rangle.m \rhd P_1 \mid \langle 2\rangle.m \rhd P_2 \qquad m \rhd \nu y(P) \equiv \nu y(m \rhd P) \text{ with } y \notin m$$

These laws can be regarded as rewriting rules, enabling the processes to execute. First, we decompose a process in its threads, each thread having a copy of the process's memory. The second rule moves the restriction from a CCS process to a RCCS one. The side condition is necessary as a process $\nu a.(a.\nu a P \mid \overline{a})$ has to do an α-conversion before executing the process P.

The labeled transition system

We denote by $R \xrightarrow{\phi:\alpha} R'$ a transition on processes, where R can evolve to R' after an action α. We add to the label of the transition the thread's identifier. In case of a synchronization, we have the identifiers of both thread in the transition's label. The backward transition is denoted with $\xrightarrow{\phi:\alpha^-}$.

Let $\zeta \in \{\alpha, \alpha^-\}$, and $\eta \in \{(\phi, \star), (\phi_1, \phi_2)\}$. We have that $\overline{\overline{x}} = x$ and $\overline{\tau} = \tau$. We use an update function $R_{(\phi_1,\phi_2)}$ for the memory of R after a synchronization. It is defined inductively on the structure of R:

$$\begin{array}{ll}
(R \mid S)_{(\phi_1,\phi_2)} & = R_{(\phi_1,\phi_2)} \mid S_{(\phi_1,\phi_2)} \\
(\nu y R)_{(\phi_1,\phi_2)} & = \nu y(R_{(\phi_1,\phi_2)}) \\
(\langle\rangle \rhd P)_{(\phi_1,\phi_2)} & = \langle\rangle \rhd P \\
(\langle i\rangle.m \rhd P)_{(\phi_1,\phi_2)} & = \langle i\rangle.m \rhd P \\
(\langle m', \alpha, Q\rangle.m \rhd P)_{(\phi_1,\phi_2)} & = \langle m', \alpha, Q\rangle.m \rhd P \\
(\langle\star, \alpha, Q\rangle.m \rhd P)_{(\phi_1,\phi_2)} & = \langle\phi_1, \alpha, Q\rangle.m \rhd P \quad \text{if } \Phi(m) = \phi_2 \\
& = \langle\phi_2, \alpha, Q\rangle.m \rhd P \quad \text{if } \Phi(m) = \phi_1 \\
& = \langle\star, \alpha, Q\rangle.m \rhd P \quad \text{otherwise}
\end{array}$$

For example, if we apply the function on $(\langle\star, a\rangle.\langle 1\rangle \rhd 0 \mid \langle\star, \overline{a}\rangle.\langle 2\rangle \rhd 0)_{(\langle 1\rangle,\langle 2\rangle)}$ we have $\langle\langle 2\rangle, a\rangle.\langle 1\rangle \rhd 0 \mid \langle\langle 1\rangle, \overline{a}\rangle.\langle 2\rangle \rhd 0$. The behavior of a process is then defined using the following set of transition rules:

$$m \rhd \alpha.P + Q \xrightarrow{\Phi(m),\star:\alpha} \langle\star, \alpha, Q\rangle.m \rhd P \qquad \langle\star, \alpha, Q\rangle.m \rhd P \xrightarrow{\Phi(m),\star:\alpha^-} m \rhd \alpha.P + Q$$

$$\frac{R \xrightarrow{\phi_1,\star:\alpha} R' \qquad S \xrightarrow{\phi_2,\star:\overline{\alpha}} S'}{R \mid S \xrightarrow{\phi_1,\phi_2:\tau} (R' \mid S')_{(\phi_1,\phi_2)}} \qquad \frac{R \xrightarrow{\phi_1,\star:\alpha^-} R' \qquad S \xrightarrow{\phi_2,\star:\overline{\alpha}^-} S'}{(R \mid S)_{(\phi_1,\phi_2)} \xrightarrow{\phi_1,\phi_2:\tau^-} R' \mid S'}$$

$$\frac{R \xrightarrow{\eta:\zeta} R'}{R \mid S \xrightarrow{\eta:\zeta} R' \mid S} \qquad \frac{R \xrightarrow{\eta:\zeta} R'}{\nu y R \xrightarrow{\eta:\zeta} \nu y R'} y \notin \zeta \qquad \frac{R \equiv S \xrightarrow{\eta:\zeta} S' \equiv R'}{R \xrightarrow{\eta:\zeta} R'}$$

The first two rules are axioms for performing either forward or backward. They simply add or remove an entry from the memory. The third and fourth rules represent a forward, respectively backward synchronization that updates the memories of the threads involved. The remaining of the rules are allowing the actions to

pass through the rest of the process structure, possibly with some conditions. In particular, the rule on restriction says that a transition on a restricted name cannot exit the scope of the restriction. We also have in the LTS, the symmetric rules resulting from swapping the two processes involved in a summation or a parallel composition.

Let us look at some examples of a process computation in order to better understand the rules.

- Consider the process $a.P + \overline{b}.Q$ that first chooses the transition on a then backtracks and fires the transition on b:

$$\langle\rangle \rhd a.P + \overline{b}.Q \xrightarrow{a} < \star, a, \overline{b}.Q > \rhd P \xrightarrow{a^-} a.P + \overline{b}.Q \xrightarrow{\overline{b}} < \star, \overline{b}, a.P > \rhd Q$$

- An example of forward synchronization is in $\langle 1\rangle \rhd \overline{a} \mid \langle 2\rangle \rhd a$, where we have the following derivation:

$$\frac{\langle 1\rangle \rhd \overline{a} \xrightarrow{\langle 1\rangle:\overline{a}} \langle \star, \overline{a}\rangle.\langle 1\rangle \rhd 0 \qquad \langle 2\rangle \rhd a \xrightarrow{\langle 2\rangle:a} \langle \star, a\rangle.\langle 2\rangle \rhd 0}{\langle 1\rangle \rhd \overline{a} \mid \langle 2\rangle \rhd a \xrightarrow{\langle 1\rangle,\langle 2\rangle:\tau} \langle\langle 2\rangle, a\rangle.\langle 1\rangle \rhd 0 \mid \langle\langle 1\rangle, \overline{a}\rangle.\langle 2\rangle \rhd 0}$$

  If we want to do a backward transition on a then necessarily both threads have to roll back through a synchronization. We have the derivation:

$$\frac{\langle \star, \overline{a}\rangle.\langle 1\rangle \rhd 0 \xrightarrow{\langle 1\rangle:\overline{a}^-} \langle 1\rangle \rhd \overline{a} \qquad \langle \star, a\rangle.\langle 2\rangle \rhd 0 \xrightarrow{\langle 2\rangle:a^-} \langle 2\rangle \rhd a}{\langle\langle 2\rangle, a\rangle.\langle 1\rangle \rhd 0 \mid \langle\langle 1\rangle, \overline{a}\rangle.\langle 2\rangle \rhd 0 \xrightarrow{\langle 1\rangle,\langle 2\rangle:\tau^-} \langle 1\rangle \rhd \overline{a} \mid \langle 2\rangle \rhd a}$$

- The last rule of the LTS allows process rewriting before triggering a transition. A process of the form $R = m \rhd a.P \mid b.Q$ cannot execute. It is necessary to rewrite it in the equivalent form $R' = \langle 1\rangle.m \rhd a.P \mid \langle 2\rangle.m \rhd b.Q$. If one wants to do a backward transition from R' then it first has to rewrite it back as R. Similarly for the second congruence rule. The rules do not enable two congruent processes to fire the same transition.

Semantic correctness

RCCS supports reversibility only for a subset of monitored processes, the ones that have a valid history. We call these processes *coherent* if their memories are pairwise *coherent*.

Definition 3 *Coherence is the smallest symmetric relation such that*

- $\langle m, \alpha, P\rangle.m' \sim \langle m', \alpha, P\rangle.m$
- $\langle i\rangle.m \sim \langle j\rangle.m$ *with* $i \neq j$

For example the process $\langle 1\rangle.m \rhd P$ (without the part $\langle 2\rangle.m \rhd Q$), cannot do a backward transition. Its execution will eventually block.

3 Reversible π calculus

Reversible π calculus is a process calculus that implements backtracking for π calculus processes, similarly to how RCCS implements it for the CCS processes of section 2. Let us then start by looking at the some differences between π and CCS. We continue the presentation by introducing the grammar of processes and the LTS. The rules in the LTS however are complex and therefore easier to understand with examples and explanations. An important property of the LTS, arising from its symmetry, is that any forward transition can backtrack and reversely. We prove this in the *loop* lemma below. We conclude the section with one main difference between π calculus and Rπ: semantic correctness.

A variant of reversible π calculus was proposed in [LMS10]. The authors use a global memory to store the entire process for every forward computation. Their approach is not compositional, as they are not defining an LTS, and does not include the restriction operator.

The π calculus

Names are transmitted through channels. Therefore the prefixes are now $x(y)$ for an input and $\overline{x}\langle y\rangle$ for an output. When a name is received, a substitution occurs within the continuation process. If a process $x(y).P$ receives a name z then the process becomes $P\{z/y\}$, in which all occurrences of y in P were replaced by z. In CCS νyP cannot communicate with the exterior but only perform internal synchronizations. In π calculus a private name can be communicated to the environment, by sending it on a channel: $\nu y(\overline{a}\langle y\rangle.P) \xrightarrow{\overline{a}(\nu y)} P$. In case of a synchronization $\nu y(\overline{a}\langle y\rangle.P) \mid a(t).Q \xrightarrow{\tau} \nu y(P \mid Q\{y/t\})$, the binder reappears in the process structure, as y is now known by both P and Q but by no one else.

We sometimes keep the notations of CCS, when the name transmitted on the channel is not considered important. For example, $a \mid \overline{a}$ stands for $a(t) \mid \overline{a}\langle y\rangle$ for some t and y.

Preliminary examples for Rπ

We first show some examples in order to gain the intuition of how this calculus works. As in RCCS, memories are used to store the forward computations. There are two problems we encounter when we add the memories in Rπ: one due to name substitution and the second, more important, arises from the extrusion of bound names.

In order to handle substitutions, both the original and the received names are added into the memory but no substitution is done on the process itself. Consider the example we have seen in section 1, $a(x).(x(t) \mid y(z))$ which receives y in the transition:

$$\langle 1\rangle \rhd a(x).(x(t) \mid y(z)) \mid \langle 2\rangle \rhd \overline{a}\langle y\rangle \xrightarrow{\langle 1\rangle,\langle 2\rangle:\tau} \langle 2, a[y/x]\rangle.\langle 1\rangle \rhd x(t) \mid y(z) \mid \langle 1, \overline{a}\langle y\rangle\rangle.\langle 2\rangle \rhd 0$$

Using the memory entry $\langle 2, a[y/x]\rangle$ the process has all the information it needs for its future transitions either forward or backward. Suppose that it wants to do a transition on $x(t)$. Then it looks up in the memory whether the free name x has been subject to a substitution in the past and it applies the substitution $\{y/x\}$ on the transition label. In this manner our policy of applying the substitutions is transparent to the environment. Also we can easily backtrack to the process $a(x).(x(t) \mid y(z))$. The alternative would consist of applying the substitutions on the process and obtain $\langle 2, a[y/x]\rangle.\langle 1\rangle \rhd y(t) \mid y(z)$. Then backtracking would lead to the incorrect process $a(x).(x(t) \mid x(z))$.

A second, more important addition consists in the handling of scope extrusion. Consider the process $P = \nu y(\overline{a}\langle y\rangle \mid \overline{y}\langle t\rangle)$, in which the transition on a is necessarily before the one on y. Then we reach the process $\langle \star, \overline{a}\langle y\rangle\rangle.\langle 1\rangle \rhd 0 \mid \langle \star, \overline{y}\langle t\rangle\rangle.\langle 2\rangle \rhd 0$. Now for backtracking we cannot allow $\overline{a}\langle y\rangle$ to go first, as this does not correspond to a possible forward execution. We forbid this transition until the one on $\overline{y}\langle t\rangle$ has fired. However, this information is local to the second thread. We need a sort of global "memory", that informs the first thread of the execution of others. This limits the distributed aspect of the calculus, as now parallel threads have to be aware of one another. More precisely, a process containing a restriction νyR maintains throughout the execution the scope of the restriction. With this purpose the operator δ is introduced, to act as the "shadow" of ν. When νyR does an extrusion of y the process becomes δyR. In such a process we call y *freed*. The transitions on P are, then as follows:

$$\nu y(\langle 1\rangle \rhd \overline{a}\langle y\rangle \mid \langle 2\rangle \rhd \overline{y}\langle t\rangle) \xrightarrow{\langle 1\rangle:\overline{a}(\nu y)} \delta y(\langle \star, \overline{a}\langle y\rangle\rangle.\langle 1\rangle \rhd 0 \mid \langle 2\rangle \rhd \overline{y}\langle t\rangle) \xrightarrow{\langle 2\rangle:\overline{y}\langle t\rangle} \delta y(\langle \star, \overline{a}\langle y\rangle\rangle.\langle 1\rangle \rhd 0 \mid \langle \star, \overline{y}\langle t\rangle\rangle.\langle 2\rangle \rhd 0)$$

δyP can go back to νyP only when we cannot backtrack any external communications involving y. If the above process wants to do a backward transition on $\overline{a}\langle y\rangle$ it looks in the memories of all its threads belonging to the scope of δy and deduces that there is another transition that has to fire first. It can do the transition on $\overline{a}\langle y\rangle$ only after $\overline{y}\langle t\rangle$ was restored:

$$\delta y(\langle \star, \overline{a}\langle y\rangle\rangle.\langle 1\rangle \rhd 0 \mid \langle 2\rangle \rhd \overline{y}\langle t\rangle) \xrightarrow{\langle 1\rangle:\overline{a}(\nu y)^-} \nu y(\langle 1\rangle \rhd \overline{a}\langle y\rangle \mid \langle 2\rangle \rhd \overline{y}\langle t\rangle)$$

Consider now the process $\nu y(m_1 \rhd \overline{a}\langle y\rangle \mid m_2 \rhd \overline{b}\langle y\rangle \mid m_3 \rhd y(t))$ that can reach $R' = \delta y(\langle\star, \overline{a}\langle y\rangle\rangle.m_1 \rhd 0 \mid \langle\star, \overline{b}\langle y\rangle\rangle.m_2 \rhd 0 \mid \langle\star, y(t)\rangle.m_3 \rhd 0)$. Then any of the following transitions is allowed:

$$R' \xrightarrow{m_1:\overline{a}\langle y\rangle^-} \delta y(m_1 \rhd \overline{a}\langle y\rangle \mid \langle\star, \overline{b}\langle y\rangle\rangle.m_2 \rhd 0 \mid \langle\star, y(t)\rangle.m_3 \rhd 0)$$

$$R' \xrightarrow{m_2:\overline{b}\langle y\rangle^-} \delta y(\langle\star, \overline{a}\langle y\rangle\rangle.m_1 \rhd 0 \mid m_2 \rhd \overline{b}\langle y\rangle \mid \langle\star, y(t)\rangle.m_3 \rhd 0)$$

As an extruder of y still has to backtrack, the δ constructor is kept.

Henceforth, in $\{x(y), \overline{x}\langle y\rangle\}$ we call x the subject and y the object. By $y \notin S$ we understand that y is not present neither in the structure of the process S nor in its memory. We also introduce the *context*, denoted by $C[]$, which is a process with a "hole", serving as a placeholder for another process and defined as follows:

$$C[], D[] = (C[] \mid D[]) \mid []$$

Grammar of processes

The structure of the memories and the thread's identifiers are the same as in RCCS. Similarly, π calculus processes are build using prefixing, summation, parallel composition and restriction.

$$\begin{array}{ll} \phi ::= & \star \mid (1,2)^* \quad \text{(memory's identifier)} \\ m ::= & \langle\rangle \mid \langle 1\rangle.m \mid \langle 2\rangle.m \mid \langle \phi, \alpha, P\rangle.m \quad \text{(memories)} \\ P, Q ::= & x(y).P + Q \mid \overline{x}\langle y\rangle.P + Q \mid (P \mid Q) \mid \nu y P \mid \emptyset \quad (\pi \text{ processes}) \end{array}$$

We differentiate between the prefixes in the structure of a π process and the ones stored in the memory, as the latter have to include also the substitutions in case of an input. However, the name received, used for the substitution, is known only in case of a synchronization.

$$\alpha ::= x[y/z] \mid x[\star/z]) \mid \overline{x}\langle y\rangle \quad \text{(actions)}$$

We can define the bound names in a memory as follows: in $x[y/z]$ or $x[\star/z]$, z is bound. The rest of the names are free. Henceforth the sets $\text{fn}(R)$ and $\text{bn}(R)$ of a process R include also the free and bound names of its memory.

The transition labels are no longer just the prefix as it was the case in RCCS. We also specify, in case of an output, whether the object is private or freed. In the label $\overline{x}\langle\delta y\rangle$ both x and y belong to the set of free names of the transition.

$$\gamma ::= x(z) \mid \overline{x}\langle y\rangle \mid \overline{x}(\nu y) \mid \overline{x}\langle\delta y\rangle \mid \tau \quad \text{(transition's labels)}$$

The Rπ processes, ranged over by R, S are built using parallel composition, restriction or the operator δ:

$$R, S ::= (m \rhd P \mid R) \mid \nu y R \mid \delta x R \mid \emptyset \quad (R\pi \text{ processes})$$

Note that all processes defined have only finite computations. The calculus also supports an infinite behavior using recursive definitions of processes. However, we do not expect that the recursive definitions will introduce any difficulties from the point of view of reversibility and hence are not included in the current presentation.

The congruence rules are the ones defined in RCCS, without any modification.

3.1 The labeled transition system

Let $\zeta \in \{\gamma, \gamma^-\}$, where γ is the label of a transition, and $\eta \in \{(\phi, \star), (\phi_1, \phi_2)\}$.

Definition 4 *Let m be a function on the labels of a transition for a thread, $m \rhd P$, defined inductively on the memory m.*

$$\begin{array}{ll} (\langle \phi, x[z/t], P\rangle.m)(y) = z, \text{ if } t = y & \\ \qquad\qquad\qquad\qquad\quad = m(y), \text{ otherwise} & \\ (\langle \phi, x[\star/t], p\rangle.m)(y) & = (\langle i\rangle.m)(y) = (\langle K[a]\rangle.m)(y) = m(y) \\ (\langle\rangle)(y) & = y \end{array}$$

The function applies the necessary substitutions on the names used in the transition label. For example if the process $\langle \phi, a[y/x]\rangle.\langle 1\rangle \rhd x(t)$ wants to perform an input, it uses the channel $(\langle \phi, a[y/x]\rangle.\langle 1\rangle)(x) = y$. We apply this function on all the free names of a transition.

IN+
$$m \rhd x(z).P + Q \xrightarrow{\Phi(m),\star:m(x)(z)} \langle \star, x[*/z], Q\rangle.m \rhd P$$
OUT+
$$m \rhd \overline{x}\langle y\rangle.P + Q \xrightarrow{\Phi(m),\star:\overline{m(x)}m(y)} \langle \star, \overline{x}\langle y\rangle, Q\rangle.m \rhd P$$
IN-
$$\langle \star, x[*/z], Q\rangle.m \rhd P \xrightarrow{\Phi(m),\star:m(x)(z)^-} m \rhd x(z).P + Q$$
OUT-
$$\langle \star, \overline{x}\langle y\rangle, Q\rangle.m \rhd P \xrightarrow{\Phi(m),\star:\overline{m(x)}(m(y))^-} m \rhd \overline{x}\langle y\rangle.P + Q$$

Apart from substitution, the axioms are very similar to the ones of RCCS. When we perform an input, we do not know the name received. It is only during a synchronization that the substitution occurs. As it was the case for RCCS, we also need a function that updates the memories after a synchronization. This function is now used also to add the missing name in the entry $x[\star/z]$.

COM+
$$\frac{R \xrightarrow{\phi_1,\star:\overline{x}\langle y\rangle} R' \qquad S \xrightarrow{\phi_2,\star:x(z)} S'}{R \mid S \xrightarrow{\phi_1,\phi_2:\tau} (R' \mid S')_{@(\phi_1,\phi_2,y)}}$$
COM-
$$\frac{R \xrightarrow{\phi_1,\star:\overline{x}\langle y\rangle^-} R' \qquad S \xrightarrow{\phi_2,\star:x(z)^-} S'}{(R \mid S)_{@(\phi_1,\phi_2,y)} \xrightarrow{\phi_1,\phi_2:\tau^-} R' \mid S'}$$

We have the rules for passing through a process structure, the equivalent for the ones of RCCS (for parallel composition, for a ν or for congruence). The only difference is the forward parallel composition rule, which has a condition added, in order to ensure that all bound names are different from the free ones. The condition is not necessary backwards, as we explain later on in proposition 10.

CONGR+-
$$\frac{R \equiv S \xrightarrow{\eta:\zeta} S' \equiv R'}{R \xrightarrow{\eta:\zeta} R'}$$
NEW+-
$$\frac{R \xrightarrow{\eta:\zeta} R'}{\nu y R \xrightarrow{\eta:\zeta} \nu y R'} y \notin \zeta$$
PAR+
$$\frac{R \xrightarrow{\eta:\gamma} R'}{R \mid S \xrightarrow{\eta:\gamma} R' \mid S} \mathrm{bn}(\gamma) \cap \mathrm{fn}(S) = \emptyset$$
PAR-
$$\frac{R \xrightarrow{\eta:\gamma^-} R'}{R \mid S \xrightarrow{\eta:\gamma^-} R' \mid S}$$

Let us consider the rules for ν and δ. Whenever a process $\nu y R$ makes an extrusion of y, ν is replaced with δ, to denote that y was freed but also to remember the scope of the binder. Once the process $\delta y R$ is obtained, no forward transition can restore ν or remove δ. This is ensured by the rules OPEN+ and DELTA+ below, that cover all possible forward transitions. Rule DELTA- also says that the only backward transition that can restore ν is the output a freed name.

OPEN+
$$\frac{R \xrightarrow{\eta:\overline{x}\langle y\rangle} R'}{\nu y R \xrightarrow{\eta:\overline{x}(\nu y)} \delta y R'}$$
OPEN DELTA+
$$\frac{R \xrightarrow{\eta:\overline{x}\langle y\rangle} R'}{\delta y R \xrightarrow{\eta:\overline{x}(\delta y)} \delta y R'}$$
DELTA+-
$$\frac{R \xrightarrow{\eta:\zeta} R'}{\delta y R \xrightarrow{\eta:\zeta} \delta y R'} \zeta \neq \overline{x}\langle y\rangle, \overline{x}\langle y\rangle^-, \overline{x}(\delta z)^-$$

We have two rules for a backward transitions on $\overline{x}\langle y\rangle$ from a process $\delta y P$, one to restore ν and one to keep δ. For the former, we ensure that no other external communications involving y have yet to backtrack. More specifically, we do not want to have in the memory of a thread that is within the scope of δy, transitions that have y either as subject or object.

Definition 5 *We define the set of free names over an action α, stored in a memory m, as follows:*

$$\mathrm{fn_m}(x[y/z]) = \{m(x), y\} \qquad \mathrm{fn_m}(x(z)) = \{m(x)\} \qquad \mathrm{fn_m}(\overline{x}\langle z\rangle) = \{m(x), m(z)\}$$

$$y_\star \in R \text{ iff } R \equiv (m_0.\langle \star, \alpha, Q\rangle.m_1 \rhd P \mid S) \text{ with } y \in \mathrm{fn}_{\mathrm{m}_1}(\alpha)$$

Then we can restore ν using the rule:

OPEN-
$$\frac{R \xrightarrow{\eta:\overline{x}\langle y\rangle^-} R'}{\delta y R \xrightarrow{\eta:\overline{x}(\nu y)^-} \nu y R'} y_\star \notin R'$$

We keep δ in the structure of a process when there is an extruder of y in the memory of one of its threads. As we have seen, the transition on a is not possible in $\delta y(\langle\star, \overline{a}\langle y\rangle\rangle \rhd 0 \mid \langle\star, \overline{y}\langle b\rangle\rangle \rhd 0)$ but it is in $\delta y(\langle\star, \overline{a}\langle y\rangle\rangle.m_1 \rhd 0 \mid \langle\star, \overline{b}\langle y\rangle\rangle.m_2 \rhd 0 \mid \langle\star, y(t)\rangle.m_3 \rhd 0)$. Remember that by extrusion we understand the sending of a bound name to the environment. In the process $\nu y(\overline{a}\langle y\rangle \mid a(t))$ the synchronization on a does not extrude y as we obtain $\nu y(\langle 2, \overline{a}\langle y\rangle\rangle.\langle 1\rangle \rhd 0 \mid \langle 1, a[y/t]\rangle.\langle 2\rangle \rhd 0)$.

This condition however is not enough. Consider the process $\nu y(\overline{a}\langle y\rangle \mid y(t).\overline{b}\langle y\rangle)$. The only possible forward computation for this process is:

$$\nu y(\langle 1\rangle \rhd \overline{a}\langle y\rangle \mid \langle 2\rangle \rhd y(t).\overline{b}\langle y\rangle) \xrightarrow{\langle 1\rangle:\overline{a}(\nu y)} \xrightarrow{\langle 2\rangle:y(t)} \xrightarrow{\langle 2\rangle:\overline{b}(\delta y)} \delta y(\langle\star, \overline{a}\langle y\rangle\rangle.\langle 1\rangle \rhd 0 | \langle\star, \overline{b}\langle y\rangle\rangle.\langle\star, y(t)\rangle.\langle 2\rangle \rhd 0)$$

It would seem that the backward transition on a is allowed since the extruder on b has yet to backtrack. However, b became an extruder only after some previous computations on y. We say then that a proper extruder of a bound name is one belonging to a thread that has no external transitions involving y in its past. Now consider the process $\nu y(\overline{a}\langle y\rangle \mid y(t).c(t) \mid \overline{c}\langle t\rangle.\overline{b}\langle y\rangle)$. As above, b is not a true extruder, however, the communication on y is no longer in its past, but in the past of one of its synchronizing partners. Another example is the process $\nu z\nu y(\overline{a}\langle y\rangle \mid z(t).c(t) \mid \overline{c}\langle t\rangle.\overline{b}\langle y\rangle \mid y(t).\overline{d}\langle z\rangle)$. The only possible forward trace is $\xrightarrow{\overline{a}(\nu y)} \xrightarrow{y(t)} \xrightarrow{\overline{d}(\nu z)} \xrightarrow{z(t)} \xrightarrow{\tau} \xrightarrow{\overline{b}(\delta y)}$. We see that b is not an extruder but because there is a y in the past of $\overline{d}\langle z\rangle$ which is an extruder of the freed name z.

We want to cover all these cases in our definition of an extruder, denoted with $\uparrow y_\star$. Therefore the following definition is quite technical. However, the condition could be reformulated using event structures.

Definition 6 *We denote with $\uparrow y_\star$ an extruder of y that has yet to backtrack. The congruence relation used here includes the commutativity and transitivity of the parallel composition. The set freed(R) contains all the freed names in R.*

$$\uparrow y_\star \in R \text{ iff } R \equiv (m_0.\langle\star, \overline{x}\langle z\rangle, Q\rangle.m_1 \rhd P \mid S) \text{ for some } x \text{ with } m_1(z) = y \text{ and}$$

$$\text{let } \mathbb{M} = \{m_1\} \cup synch(m_1, S) \text{ in } accessible_y(\mathbb{M}, m_1 \rhd P \mid S)$$

where synch is a function that gathers in a set the memories of all synchronizing partners:

$$\begin{aligned} synch(\langle\phi, \alpha\rangle.m, R) &= \{m_1\} \cup synch(m_1, R) \cup synch(m, R) \text{ if } R \equiv (m_0.m_1 \rhd P \mid S) \text{ such that } \Phi(m_1) = \phi \\ &= synch(m, R) \text{ otherwise} \\ synch(\langle\star, \alpha\rangle.m, R) &= synch(\langle i\rangle.m, R) = synch(m, R) \\ synch(\langle\rangle, R) &= \emptyset \end{aligned}$$

and $accessible_y(M, R)$ verifies that for each freed name in M there is an extruder in R:

$$\begin{aligned} &accessible_y(M, R) = true \text{ if} \\ &\quad \forall m \in \mathbb{M}, y \notin \mathrm{fn}(m) \text{ and } \forall z \in \mathrm{fn}(m) \cap freed(R), R \equiv m_0.\langle\star, \overline{x}\langle t\rangle, Q\rangle.m_1 \rhd P \mid S \text{ with} \\ &\qquad \text{let } \mathbb{M} = \{m_1\} \cup synch(m_1, S) \text{ in } accessible_y(M, m_1 \rhd P \mid S) \\ &\qquad\quad = false \text{ otherwise} \\ &accessible_y(M, 0) = true \end{aligned}$$

Note that each time we encounter a δ in the process structure the set of freed names of a process increases. Therefore we have the two rules for a backward transition on $\overline{x}(\delta y)$.

$$\text{OPEN DELTA-}\quad \frac{R \xrightarrow{\eta:\overline{x}\langle y\rangle^-} R'}{\delta y R \xrightarrow{\eta:\overline{x}(\delta y)^-} \delta y R'} \uparrow y_\star \in R' \qquad \text{DELTA FREED-}\quad \frac{R \xrightarrow{\eta:\overline{x}(\delta y)^-} R'}{\delta z R \xrightarrow{\eta:\overline{x}(\delta y)^-} \delta z R'} \uparrow y_\star \in \delta z R'$$

Let us see how we can write the forward and backward versions of the rule CLOSE from the π calculus. The former one is a straightforward adaptation of the initial rule.

$$\text{CLOSE+}\quad \frac{R \xrightarrow{\phi_1,\star:\overline{x}(\nu y)} R' \qquad S \xrightarrow{\phi_2,\star:x(z)} S'}{R \mid S \xrightarrow{\phi_1,\phi_2:\tau} \nu y(R' \mid S')_{@(\phi_1,\phi_2,y)}} y \notin \mathrm{fn}(S)/\{z\}$$

The update function however is changed. Consider the following derivation tree:

$$\dfrac{\dfrac{\langle 1\rangle \rhd \overline{a}\langle y\rangle \xrightarrow{\langle 1\rangle,\star:\overline{a}\langle y\rangle} \langle \star, \overline{a}\langle y\rangle\rangle.\langle 1\rangle \rhd 0}{\nu y(\langle 1\rangle \rhd \overline{a}\langle y\rangle) \xrightarrow{\langle 1\rangle,\star:\overline{a}(\nu y)} \delta y(\langle \star, \overline{a}\langle y\rangle\rangle.\langle 1\rangle \rhd 0)} \qquad \langle 2\rangle \rhd a(t) \xrightarrow{\langle 2\rangle,\star:a(t)} \langle \star, a[\star/t]\rangle.\langle 2\rangle \rhd 0}{\nu y(\langle 1\rangle \rhd \overline{a}\langle y\rangle) \mid \langle 2\rangle \rhd a(t) \xrightarrow{\langle 1\rangle,\langle 2\rangle:\tau} \nu y(\delta y(\langle \star, \overline{a}\langle y\rangle\rangle.\langle 1\rangle \rhd 0) \mid \langle \star, a[\star/t]\rangle.\langle 2\rangle \rhd 0)_{@(\phi_1,\phi_2,y)}}$$

where we have to erase δ from the resulting process. To understand why it is necessary let us discuss, as an example, the process $\nu y(R \mid S_1 \mid S_2)$, with $R = n \rhd \overline{a}\langle y\rangle.\overline{a}\langle y\rangle$, $S_1 = m_1 \rhd a(t).P_1$ and $S_2 = m_2 \rhd a(t).P_2$. The possible forward computations for this process are: either R does a synchronization with S_1 and then one with S_2 or the reverse. The processes reached in both scenarios should be congruent, but if we let ν and δ to pile up in a process structure, they are not:

$$\nu y(R \mid S_1 \mid S_2) \longrightarrow^\star \nu y(\delta y(\delta y R' \mid S_1') \mid S_2') \qquad \nu y(R \mid S_1 \mid S_2) \longrightarrow^\star \nu y(\delta y(\delta y R' \mid S_2') \mid S_1')$$

Definition 7 *The update function for the memory of a process after a synchronization, denoted by $R_{@(m_1,m_2,y)}$ is defined inductively on the structure of R:*

$$\begin{array}{ll}
(R \mid S)_{@(\phi_1,\phi_2,y)} & = (R_{@(\phi_1,\phi_2,y)}) \mid S_{@(\phi_1,\phi_2,y)} \\
(\delta t.R)_{@(\phi_1,\phi_2,y)} & = R_{@(\phi_1,\phi_2,y)},\ \textit{if } t = y \\
 & = \delta t(R_{@(\phi_1,\phi_2,y)}),\ \textit{otherwise} \\
(\nu y R)_{@(\phi_1,\phi_2,y)} & = \nu y(R_{@(\phi_1,\phi_2,y)}) \\
(\langle \star, x[\star/z], Q\rangle.m' \rhd P)_{@(\phi_1,\phi_2,y)} & = \langle \phi_1, x[y/z], Q\rangle.m' \rhd P,\ \textit{if } \Phi(m') = \phi_2 \\
(\langle \star, \overline{x}\langle y\rangle, Q\rangle.m' \rhd P)_{@(\phi_1,\phi_2,y)} & = \langle \phi_2, \overline{x}\langle y\rangle, Q\rangle.m' \rhd P,\ \textit{if } \Phi(m') = \phi_1
\end{array}$$

The backward rule for CLOSE has the process $\nu y(R \mid S)$ doing a backward synchronization on y. In order not to lose the scope of νy we add a side condition to the rule:

$$\text{CLOSE-}\quad \dfrac{R \xrightarrow{\phi_1,\star:\overline{x}(\nu y)^-} R' \qquad S \xrightarrow{\phi_2,\star:x(z)^-} S'}{\nu y(R \mid S)_{@(\phi_1,\phi_2,y)} \xrightarrow{\phi_1,\phi_2:\tau^-} R' \mid S'} y \notin \mathrm{fn}(S')/\{z\}$$

We have so far covered the cases in which a synchronization occurs on a free or a bound name. We still have to cover the synchronization on a freed name. To show why such rules are necessary consider the process $P = \nu y(\overline{a}\langle y\rangle \mid \overline{b}\langle y\rangle) \mid a(t).\overline{c}\langle t\rangle$ which has the following correct computations:

$$\langle\rangle \rhd P \xrightarrow{\phi_1,\phi_3:\tau} \xrightarrow{\phi_2:\overline{b}\langle y\rangle} \xrightarrow{\phi_3:\overline{c}\langle y\rangle} \delta y(\langle \phi_3, \overline{a}\langle y\rangle\rangle.m_1 \rhd 0 \mid \langle \star, \overline{b}\langle y\rangle\rangle.m_2 \rhd 0 \mid \langle \star, \overline{c}\langle t\rangle\rangle.\langle \phi_1, a[y/t]\rangle.m_3 \rhd 0)$$

$$\langle\rangle \rhd P \xrightarrow{\phi_2:\overline{b}\langle y\rangle} \xrightarrow{\phi_1,\phi_3:\tau} \xrightarrow{\phi_3:\overline{c}\langle y\rangle} \delta y(\langle \phi_3, \overline{a}\langle y\rangle\rangle.m_1 \rhd 0 \mid \langle \star, \overline{b}\langle y\rangle\rangle.m_2 \rhd 0) \mid \langle \star, \overline{c}\langle t\rangle\rangle.\langle \phi_1, a[y/t]\rangle.m_3 \rhd 0$$

where the obtained processes are not congruent. This example motivates then the need of the two following rules in the LTS and why we have to distinguish freed names in the label of an output.

$$\text{CLOSE DELTA+}\quad \dfrac{R \xrightarrow{\phi_1,\star:\overline{x}(\delta y)} R' \qquad S \xrightarrow{\phi_2,\star:x(z)} S'}{R \mid S \xrightarrow{\phi_1,\phi_2:\tau} \delta y(R' \mid S')_{@(\phi_1,\phi_2,y)}}$$

$$\text{CLOSE DELTA-}\quad \dfrac{R \xrightarrow{\phi_1,\star:\overline{x}(\delta y)^-} R' \qquad S \xrightarrow{\phi_2,\star:x(z)^-} S'}{\delta y(R \mid S)_{@(\phi_1,\phi_2,y)} \xrightarrow{\phi_1,\phi_2:\tau^-} R' \mid S'} y \notin \mathrm{fn}(S')/\{z\}$$

We provide the totality of the LTS in the Appendix. Note that, as $\mid$ and $+$ are not commutative, most of the rules above have also their symmetric counterpart included in the LTS. In the following, we make two remarks on some alternatives we had when designing the LTS.

Late semantics. The semantics of π calculus we used is called the *late* semantics, which means that the name substitution is done as late as possible within the derivation tree of a transition. The intuition is that, when we perform an input, we do not know yet the name received. Alternatively, one has also an early semantics for π calculus. In the latter the received name is known when an input is performed and the name substitution occurs in the derivation as early as possible. The input transition is then $a(t).P \xrightarrow{a(y)} P\{y/t\}$.

In such a semantics the following situation could appear:

$$\nu y(\overline{a}\langle y\rangle) \mid b(z).\overline{c}\langle z\rangle \xrightarrow{\overline{a}\langle y\rangle} \xrightarrow{b(y)} \delta y(\langle\star, \overline{a}\langle y\rangle\rangle \rhd 0) \mid \langle\star, b[y/z]\rangle \rhd \overline{c}\langle z\rangle$$

in which y is extruded to the environment and then received back on the channel b. Consequently both threads should be now included into the scope of δ. Such situations lead to rules in the LTS that are unnatural.

Congruence and the LTS. There are versions of π calculus, in which the congruence relation includes the abelian monoid laws for $\mid$, given below.

$$P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 \qquad P \mid 0 \equiv P$$

However, in Rπ, the last law does not hold as $\langle 1\rangle \rhd P \mid \langle 2\rangle \rhd 0 \equiv \langle 1\rangle \rhd P$, leads to processes that can, at some point in the execution, block.

If we employ these laws on the π processes (and use the results in Rπ), we have that $\langle 1\rangle \rhd P_1 \mid \langle 2\rangle \rhd P_2 \equiv \langle 1\rangle \rhd P_2 \mid \langle 2\rangle \rhd P_1$, which is not correct. Instead the correct congruence is $\langle 1\rangle \rhd P_1 \mid \langle 2\rangle \rhd P_2 \equiv \langle 2\rangle \rhd P_2 \mid \langle 1\rangle \rhd P_1$. The commutativity and associativity of $\mid$ hold in Rπ, but as the law on the identity element does not, we chose to exclude these laws and have instead symmetric transitions rules in the LTS.

In π calculus, one can also consider the following law $\nu yR \mid S \equiv \nu y(R \mid S)$ with $y \notin S$. Such a law would ease the notations in the rules CLOSE+- and CLOSE DELTA+- in our LTS. However, it can also make the following congruent processes $\nu yR \mid S$ and $\nu y(R \mid S)$ do a synchronization (one using rule CLOSE+ and one using rule COM+) and become $\nu y(R' \mid S')$. Hence the law is not added to the congruence relation.

3.2 Properties of the calculus

For proving the following properties, it is useful to define *reachable* processes, which are processes obtained from a process with an empty memory, that is $\langle\rangle \rhd P \longrightarrow^\star R$, where P is a valid π calculus process. Recall that $\delta x \notin P$.

Proposition 8 *In a reachable process νyR, $y_\star \notin R$.*

Proposition 9 *In a reachable process δyR, $\uparrow y_\star \in R$.*

Proposition 10 *In $\delta yR \mid S$ reachable, if $y \in S$ then there is a context $C[]$ such that $S \equiv C[\nu yS]$.*

The rules of a transition forward or backward are not symmetric, due to the side conditions on some of the rules in the LTS (for example, the rules OPEN+ and OPEN-). We can add these side conditions in both directions, but as the above properties show, it is not necessary. For instance, in rule OPEN+ the side condition would be $y_\star \notin R$, which is guaranteed by proposition 8 for all reachable processes. Similarly, from proposition 10, α-conversion is not necessary when going backwards.

We have to ensure that the rules OPEN- and OPEN DELTA- are mutually exclusive. Otherwise the process δyR could do a backward transition using $\overline{x}\langle y\rangle$ in two ways. In particular, the side conditions are mutually exclusive. In the first direction, that is if $\uparrow y_\star \in R$ then $y_\star \in R$, is trivial. The other direction is shown below.

Proposition 11 (Mutual exclusiveness of rules OPEN- and OPEN DELTA-)
If $\langle\rangle \rhd \nu yP \longrightarrow^\star R$ then $\uparrow y_\star \notin R$ implies $y_\star \notin R$.

Note that we cannot impose reverse side conditions on these two rules, even though they are mutually exclusive. Consider the process $R = \delta y(\langle\star, \overline{a}\langle y\rangle\rangle.\langle 1\rangle \rhd 0 | \langle\star, \overline{y}\langle t\rangle\rangle.\langle 2\rangle \rhd 0)$ which wants to backtrack on a. If the side condition of rule OPEN- is $\uparrow y_\star \notin (\langle 1\rangle \rhd \overline{a}\langle y\rangle | \langle\star, \overline{y}\langle t\rangle\rangle.\langle 2\rangle \rhd 0)$ then the transition would be allowed, and we obtain the process $\nu y(\langle 1\rangle \rhd \overline{a}\langle y\rangle | \langle\star, \overline{y}\langle t\rangle\rangle.\langle 2\rangle \rhd 0)$. Similarly, the condition $y_\star \in R$ is not enough for rule OPEN DELTA-. In the process above, such a condition would lead to the process $\delta y(\langle 1\rangle \rhd \overline{a}\langle y\rangle | \langle\star, \overline{y}\langle t\rangle\rangle.\langle 2\rangle \rhd 0)$. Therefore we do not allow the transition on a to backtrack in R.

In the following we show that reachable processes do not gather several δ for the same name.

Proposition 12 *In δyR reachable, then there is no context $C[]$ such that $R \equiv C[\delta yS]$.*

Proposition 13 *In νyR reachable, then there is no context $C[]$ such that $R \equiv C[\delta yS]$.*

Let us consider some examples that explain why certain interleavings of ν and δ are allowed.

- From proposition 10 the process $\delta yR \mid \delta yS$ is not allowed, but $\delta yR \mid \nu yS$ and $\nu yR \mid \nu yS$ are. In $\delta yR \mid \nu yS$ whenever S does a transition on $\overline{x}(\nu y)$ then the side conditions of rule PAR+ forces α-conversion and we obtain the correct process $\delta yR \mid \delta zS\{z/y\}$. On the contrary, for $\delta yR \mid \delta yS$ no α-conversion is possible.
- The process $\delta y(\delta yR \mid S)$ is not reachable. Then $\nu y(\delta yR \mid S)$, is not reachable either as it can reach $\delta y(\delta yR \mid S)$, in the transition: $\nu y(\delta yR \mid S) \xrightarrow{\eta:\overline{x}(\nu y)} \delta y(\delta yR \mid S')$. These cases are covered in propositions 12 and 13.
- The processes $\nu y(\nu yR \mid S)$ and $\delta y(\nu yR \mid S)$ are correct. From the congruence rule $\nu y(m \rhd P) \equiv m \rhd \nu yP$ if $y \notin m$ and proposition 10 we have that when νyR wants to perform an extrusion α-conversion is necessary and we obtain the correct process $\delta y(\delta zR\{z/y\} \mid S)$.

The following lemma shows that, given any forward trace s from a process with an empty memory one can obtain a backward trace, denoted by $s_\star$ by reversing each transition in the trace.

Lemma 14 (Loop lemma) *For R reachable and for every forward transition $t : R \xrightarrow{\eta:\gamma} R'$ there exists a backward transition $t_- : R' \xrightarrow{\eta:\gamma^-} R$, and conversely.*

Proof (sketch).

1. *We have $t : R \xrightarrow{\eta:\gamma} R'$ and want to prove that $R' \xrightarrow{\eta:\gamma^-} R$. We reason by induction on $R \xrightarrow{\eta:\gamma} R'$. We consider only one case, as example.*
   - *Rule* OPEN+*: $\nu yR \xrightarrow{\eta:\overline{x}(\nu y)} \delta yR'$. We want to do a transition backward on $\overline{x}(\nu y)$, for which rule* OPEN- *applies. From the hypothesis of rule* OPEN+*, we have $R \xrightarrow{\eta:\overline{x}\langle y\rangle} R'$ and it follows by induction that $R' \xrightarrow{\eta:\overline{x}\langle y\rangle^-} R$. We obtain $\delta yR' \xrightarrow{\eta:\overline{x}(\nu y)^-} \nu yR$. The condition $y_\star \notin R$ of rule* OPEN- *is verified by proposition 8.*
2. *We want to show that if $t_- : R' \xrightarrow{\eta:\gamma^-} R$ then $R \xrightarrow{\eta:\gamma} R'$.*
   - *Rule* CLOSE DELTA-*: $\delta y(R \mid S)_{@(\phi_1,\phi_2,y)} \xrightarrow{\phi_1,\phi_2:\tau^-} R' \mid S'$ with $y \notin S'$. By induction we have*

$$R \xrightarrow{\phi_1,\star:\overline{x}(\delta y)^-} R' \xrightarrow{\phi_1,\star:\overline{x}(\delta y)} R \qquad\qquad S \xrightarrow{\phi_2,\star:x(z)^-} S' \xrightarrow{\phi_2,\star:x(z)} S$$

   We can apply rule CLOSE DELTA *and we obtain: $R' \mid S' \xrightarrow{\phi_1,\phi_2:\tau} \delta y(R \mid S)_{@(\phi_1,\phi_2,y)}$.*

3.3 Semantic correctness

In π calculus, syntactically correct processes are also semantically correct. This is not the case in Rπ, as one can for instance write processes like $\delta yR \mid \delta yS$. This problem appears also in RCCS, where semantically correct processes need to have *coherent* memories. However, we can consider a set of processes (the reachable ones) for which syntactic correctness implies semantic correctness.

Let us first give a syntactic definition to the set of sound processes.

Definition 15 *A Rπ process R is sound if the following constraints hold:*

- *its memories are pairwise coherent;*
- *in $\nu y R$, $y_\star \notin R$;*
- *in $\delta y R$ there is at least one extruder of $y_\star$ in R;*
- *in $\delta y R \mid S$ if $y \in S$ then there is a context $C[]$ such that $S \equiv C[\nu y S]$;*
- *in $\delta y R$ there is no context $C[]$ such that $R \equiv C[\delta y S]$;*
- *in $\nu y R$ there is no context $C[]$ such that $R \equiv C[\delta y S]$.*

Using the above proposition on the LTS, we have that any reachable process is sound.

Lemma 16 *If R is reachable then R is sound.*

Corollary 17 *If R is sound, then for any R' such that $R \longrightarrow R'$, R' is sound.*

The completeness result that we want is the following: if R unsound then R not reachable. However it is not true with our LTS as there are unsound processes that can eventually become sound. Consider the following traces:

$$\delta y(\overline{x}\langle y\rangle) \xrightarrow{\eta:\overline{x}(\delta y)} \delta y\langle\star, \overline{x}\langle y\rangle\rangle \rhd 0 \xrightarrow{\eta:\overline{x}(\nu y)^-} \nu y(\overline{x}\langle y\rangle) \qquad \nu y(\langle\star, y\rangle \rhd 0 \mid \overline{x}\langle y\rangle) \xrightarrow{\eta:\overline{x}(\nu y)} \delta y(\langle\star, y\rangle \rhd 0 \mid \langle\star, \overline{x}\langle y\rangle\rangle \rhd 0)$$

$$\delta y(\delta y\langle\star, \overline{a}\langle y\rangle\rangle \rhd 0 \mid \langle\star, \overline{b}\langle y\rangle\rangle \rhd 0) \xrightarrow{\eta:\overline{a}(\nu y)^-} \delta y(\nu y\overline{a}\langle y\rangle \mid \langle\star, \overline{b}\langle y\rangle\rangle \rhd 0)$$

We can ensure that reduction preserves unsoundness if we add some side conditions on the rules OPEN+, OPEN DELTA+ and PAR-:

$$\textsc{open}+ \quad \frac{R \xrightarrow{\eta:\overline{x}\langle y\rangle} R'}{\nu y R \xrightarrow{\eta:\overline{x}(\nu y)} \delta y R'} y_\star \notin R \qquad \textsc{open delta}+ \quad \frac{R \xrightarrow{\eta:\overline{x}\langle y\rangle} R'}{\delta y R \xrightarrow{\eta:\overline{x}(\delta y)} \delta y R'} \uparrow y_\star \in R \qquad \textsc{par}- \quad \frac{R \xrightarrow{\eta:\gamma^-} R'}{R \mid S \xrightarrow{\eta:\gamma^-} R' \mid S} \mathrm{bn}(\gamma) \cap \mathrm{fn}(S) = \emptyset$$

4 Causality

When going backwards, the transitions do not have to preserve the exact order of the forward computation, but only the causal one. Let us first note that we are interested in the causality between two transitions that have the same sign (either both forward or both backward). Moreover, all processes considered are reachable.

There are two types of causality in π. The first one, called *subject* causality exists also in CCS and appears from prefixing. For instance in $a(x).b(t).P$ the transition on a is a cause for the transition on b. We can then use the memory of a Rπ process to retrieve this relation between transitions. In a thread $m \rhd P$ all the transitions stored in m are causes for the future transition of P.

Definition 18 (Subject causality) *Let $t_1 : R \xrightarrow{\mu_1:\zeta_1} S$ and $t_2 : S \xrightarrow{\mu_2:\zeta_2} T$ be two transition. Then t_1 is a cause of t_2 if there exists $\phi_1 \in \mu_1$ and $\phi_2 \in \mu_2$ such that $\phi_1 \subset \phi_2$.*

The other type of causality, called *object* causality, is specific to π and arises from its binding mechanism. In $\nu y(\overline{a}\langle y\rangle \mid y(t))$ the transition on a is cause for the one on y, while in $\nu y(\overline{a}\langle y\rangle \mid \overline{b}\langle y\rangle \mid y(t))$, either the transition on a or the one on b is a cause, depending on the current execution. However, such a causality is not seen in the local memory of each thread, instead can be derived from the transition's labels.

Definition 19 (Object causality) *Two transitions $t_1 : R \xrightarrow{\mu_1:\zeta_1} S$ and $t_2 : S \xrightarrow{\mu_2:\zeta_2} T$ are in a object causality relation if one of the following holds:*

- *if t_1 forward and $\nu y \in obj(\zeta_1)$ then $y \in subj(\zeta_2)$*

– *if t_2 backward and $\nu y \in obj(\zeta_2)$ then $y \in subj(\zeta_1)$*

For example, in $\nu y(\overline{a}\langle y\rangle \mid y(t))$ the transition on $\overline{a}(\nu y)$ is a cause of the one on $y(t)$. Likewise, in $\delta y(\langle\star, \overline{a}\langle y\rangle\rangle \rhd 0 \mid \langle\star, y[\star/t]\rangle \rhd 0)$ the backward transitions on $\overline{a}(\nu y)$ and $y(t)$ are in a causal relation.

Definition 20 (Weakly causality) *The weakly causality relation is defined as the transitive closure of the union of subject and object causality relations. If two transitions are not weakly causal then we say that they are concurrent.*

We call this causality weakly to emphasize that this is not the causality relation we look for. We return to this point in section 6 where we discuss event structure, as part of the future work we envisage. For the current presentation however, the relation as it is defined above, is enough.

For example, the process $R = \nu y(\overline{a}\langle y\rangle \mid \overline{b}\langle y\rangle \mid y)$ the transitions on a and b are concurrent. The transitions on the first extruder and y are also concurrent:

$$R \xrightarrow{\eta_1:\overline{b}(\nu y)} \xrightarrow{\eta_2:\overline{a}\langle\delta y\rangle} \xrightarrow{\eta_3:y} \delta y(\langle\star, \overline{a}\langle y\rangle\rangle|\langle\star, \overline{b}\langle y\rangle\rangle|\langle\star, y\rangle)$$
$$\xrightarrow{\eta_2:\overline{a}(\nu y)} \xrightarrow{\eta_1:\overline{b}\langle\delta y\rangle} \xrightarrow{\eta_3:y}$$
$$\xrightarrow{\eta_2:\overline{a}(\nu y)} \xrightarrow{\eta_3:y} \xrightarrow{\eta_1:\overline{b}\langle\delta y\rangle}$$

However, in the process $S = \nu y(\overline{a}\langle y\rangle \mid y.c \mid \overline{c}.\overline{b}\langle y\rangle)$ the transitions $\overline{a}(\nu y)$ and $\overline{b}(\delta y)$ are not concurrent.

For the process R above, the transitions $\overline{a}\langle\delta y\rangle$ and $\overline{a}(\nu y)$ are not equal. We say that the labels of two transitions ζ_1 and ζ_2 are *equivalent* if, after applying the substitutions, $\zeta_1 = \overline{x}(\nu y)$ and $\zeta_2 = \overline{x}\langle\delta y\rangle$.

Lemma 21 (Square) *Let $t_1 : R \xrightarrow{\mu_1:\zeta_1} S_1$ and $t_2 : S_1 \xrightarrow{\mu_2:\zeta_2} T$ be two concurrent transitions. Then there exists $t_2' : R \xrightarrow{\mu_2:\zeta_2'} S_2$ and $t_1' : S_2 \xrightarrow{\mu_1:\zeta_1'} T$, where ζ_i and ζ_i' are equivalent.*

Proof (sketch). *Because of the condition $\mu_1 \cap \mu_2 \neq \emptyset$ on concurrent transitions, the transitions occur on two different threads that do not synchronize. As the binders have no effect on the memory, R is of the form $R_1 \mid R_2$, $\nu y(R_1 \mid R_2)$ or $\delta y(R_1 \mid R_2)$, with $R_1 \xrightarrow{\mu_1:\zeta_1} R_1'$ and $R_2 \xrightarrow{\mu_2:\zeta_2} R_2'$. We proceed by cases on R, t_1 and t_2 and exhibit for each case the transitions t_1' and t_2'.*

We say that two transitions are *composable*, denoted by $t; t'$, if the target process of t is the source process for t'. A trace is *initial* if its source is a sound process with an empty memory. Then two traces are *coinitial* if they are initial and start from the same process and *cofinal* if they end in the same process.

Using the square lemma we can derive a *causal equivalence* between transitions, denoted by $\sim$, which is the smallest equivalence relation such that:

$$t_1; t_2 \sim t_2'; t_1' \qquad\qquad t; t_- \sim \varepsilon \qquad\qquad t_-; t \sim \varepsilon.$$

The causal equivalence between traces is defined as closing the equivalence between traces under composition.

Lemma 22 *Let s be an initial trace, there exists a trace r only forward such that $s \sim r$.*

For example, the process $P = \nu y(\overline{a}\langle y\rangle| \overline{b}\langle y\rangle \mid y)$ can have the trace $\xrightarrow{\overline{a}(\nu y)} \xrightarrow{y} \xrightarrow{\overline{b}(\delta y)} \xrightarrow{\overline{a}(\delta y)^-}$. We can rewrite it in several steps: $\xrightarrow{\overline{a}(\nu y)} \xrightarrow{y} \xrightarrow{\overline{b}(\delta y)} \xrightarrow{\overline{a}(\delta y)^-} \sim \xrightarrow{\overline{b}(\nu y)} \xrightarrow{\overline{a}(\delta y)} \xrightarrow{y} \xrightarrow{\overline{a}(\delta y)^-} \sim \xrightarrow{\overline{b}(\nu y)} \xrightarrow{y} \xrightarrow{\overline{a}(\delta y)} \xrightarrow{\overline{a}(\delta y)^-} \sim \xrightarrow{\overline{b}(\nu y)} \xrightarrow{y}$.

Theorem 23 *Two traces s_1, s_2 are coinitial and cofinal if and only if $s_1 \sim s_2$.*

Proof (sketch). *If $s_1 \sim s_2$ then we can derive that s_1, s_2 are coinitial and cofinal from the definition of $\sim$. Suppose then that s_1, s_2 are coinitial and cofinal. Using lemma 22 we have that there exists s_1', s_2' both forward such that $s_1 \sim s_1'$ and $s_2 \sim s_2'$. We reason by induction on the depth of the earliest disagreement between them, denoted by the pair t_1, t_2. If t_1, t_2 are concurrent, then there exists $t_2' \in s_1'$ and we can bubble it up and have a later earliest divergence in the two traces. If t_1, t_2 are not concurrent, then there is a subject or a object dependence between them. If they are subject dependent then, it follows that t_1 and t_2 necessarily have to add the same thing into the memory, hence $t_1 = t_2$. If there exists a object dependency between them there exists an earlier disagreement pair.*

Causal semantics for π calculus. By defining a causal relation on the transitions, we have also introduced a causal semantics for π calculus. Previous causal semantics have been studied in [DP95] and [BS95]. These works introduced the notions of *object* and *subject* dependencies. The first type of causality in Rπ, is encoded using the ordering on the memories and corresponds exactly to the one defined in the cited works. However, for the subject dependency we have a finer version, as both previous approaches differentiate the first extruder from the subsequent ones. Consider the transitions $t_1 : R \xrightarrow{\overline{x}(\nu y)} R'$ and $t_2 : R' \xrightarrow{\overline{z}\langle y\rangle} R''$, for which t_1 is considered a cause of t_2. This is not the case for the corresponding transitions in Rπ: $t_1 : R' \xrightarrow{\eta:\overline{x}(\nu y)} R'$ and $t_2 : R' \xrightarrow{\eta:\overline{z}\langle \delta y\rangle} R''$.

5 Correspondence with the π calculus

As we have seen Rπ is a calculus that adds memories to π processes in order to support reversibility. Let us now prove that indeed Rπ traces correspond to the ones in π. We show this in two steps. First we establish a strong bisimulation between forward Rπ and π. Secondly, using the results from section 4, we show that an Rπ computation has an equivalent, only forward computation in π.

We want to define a function that translates a Rπ process into one of π calculus. It is clear that such a function needs to erase all the memories and δ operators from a process. Additionally, it also has to apply all the substitutions stored in the memories. We introduce an intermediate calculus, similar to Rπ, but that attaches to a process a list of substitutions (instead of a memory), called an *environment*. As in Rπ, an input does not modify the process itself but adds the substitutions to the list. Only when a name is used in a transition's label that the substitutions are retrieved from the list and applied. Other transitions do not modify the environment. We call the calculus π_l and say that this type of substitution is a *late* one.

5.1 The intermediate calculus

We denote by T, U processes of π_l and preserve the rest of the notations from the previous sections.

$$\begin{array}{ll} \xi ::= & [y/z] :: \xi \mid [*/z] :: \xi \mid \varepsilon \quad \text{(environments)} \\ T, U ::= & (\xi \cdot P \mid T) \mid \nu y T \mid \emptyset \quad (\pi_l \text{ processes}) \end{array}$$

As in Rπ, we have the following congruence rules:

$$\xi \cdot (P \mid Q) \equiv_l \xi \cdot P \mid \xi \cdot Q \qquad\qquad \xi \cdot \nu y P \equiv_l \nu y(\xi \cdot P) \text{ if } y \notin \xi$$

There is no rule that allows a rearrangement (up to congruence) of the environment. In this manner the order in which the substitutions occurred in the process is preserved.

We also need to define the functions that apply the substitution on the transition labels and that update the environment after a synchronization.

Definition 24

– *Let ξ be a function on the labels of a transition for a thread, defined inductively on the environment:*

$$\begin{array}{ll} ([z/t] :: \xi)(y) & = z, \text{ if } t = y \\ & = \xi(y), \text{ otherwise} \\ ([\star/t] :: \xi)(y) & = \xi(y) \qquad (\varepsilon)(y) = y \end{array}$$

– *For the rules involving a synchronization we need a function $T_{[y/z]}$ which updates an entry $[*/t]$ in the environment of T:*

$$\begin{array}{ll} (T \mid U)_{[y/z]} & = T_{[y/z]} \mid U_{[y/z]} \\ (\nu x T)_{[y/z]} & = \nu x (T_{[y/z]}) \\ ([\star/t] :: \xi \cdot P)_{[y/z]} & = [y/t] :: \xi \cdot P, \text{ if } z = t \\ & = [\star/t] :: \xi \cdot P, \text{ otherwise} \end{array}$$

The transition rules are very similar to the ones of π calculus, except that the environment is updated.

$$\text{IN}_l \quad \xi \cdot x(z).P + Q \xrightarrow{\xi(x)(z)} [*/z] :: \xi \cdot P \qquad \text{OUT}_l \quad \xi \cdot \overline{x}\langle y\rangle.P + Q \xrightarrow{\overline{\xi}(x)\langle\xi(y)\rangle} \xi \cdot P$$

$$\text{COM}_l \quad \frac{T \xrightarrow{\overline{x}\langle y\rangle} T' \qquad U \xrightarrow{x(z)} U'}{T \mid U \xrightarrow{\tau} T' \mid U'_{[y/z]}} \qquad \text{PAR}_l \quad \frac{T \xrightarrow{\gamma} T'}{T \mid U \xrightarrow{\gamma} T' \mid U} \text{ if } \mathrm{bn}(\gamma) \cap \mathrm{fn}(U) = \emptyset \qquad \text{CONG}_l \quad \frac{T \equiv_l U \xrightarrow{\gamma} U' \equiv_l T'}{T \xrightarrow{\gamma} T'}$$

$$\text{RES}_l \quad \frac{T \xrightarrow{\gamma} T'}{\nu y T \xrightarrow{\gamma} \nu y T'} \text{ if } y \notin \gamma \qquad \text{OPEN}_l \quad \frac{T \xrightarrow{\overline{x}\langle y\rangle} T'}{\nu y T \xrightarrow{\overline{x}(\nu y)} T'} \qquad \text{CLOSE}_l \quad \frac{T \xrightarrow{\overline{x}(\nu y)} T' \qquad U \xrightarrow{x(z)} U'}{T \mid U \xrightarrow{\tau} T' \mid U'_{[y/z]}} \text{ if } y \notin \mathrm{fn}(U)/\{z\}$$

Definition 25 *Let ϕ be a function that translates π_l processes into π, by applying all the substitutions:*

$$\phi(T \mid U) = \phi(T) \mid \phi(U) \qquad \phi(\nu y T) = \nu y \phi(T) \qquad \phi([y/z] :: \xi \cdot P) = \phi(\xi \cdot P\{y/z\})$$

$$\phi([*/y] :: \xi \cdot P) = \phi(\xi \cdot P) \qquad \phi(\varepsilon \cdot P) = P$$

Lemma 26 (Strong bisimulation between $\xi\pi$ and its π-image)

1. *If $T \xrightarrow{\gamma} U$ and $\phi(T) = P$ then there exists Q such that $P \xrightarrow{\gamma} Q$ and $\phi(U) = Q$.*
2. *If $P \xrightarrow{\gamma}_\pi Q$ then for all T such that $\phi(T) = P$ there exists U with $T \xrightarrow{\gamma} U$ and $\phi(U) = Q$.*

As a remark, such a calculus was not studied before, mostly because apart from reversibility, it has little interest. However it bears a close resemblance to a π calculus with *explicit substitution*. The latter is a calculus in which the substitution are handled explicitly, similarly to the λ calculus with explicit substitution. Examples of such calculi are given in[Hir], [GW00] and also in [FMQ94], from which we borrowed the idea of an environment. However a difference in our approach is that we never need to apply the substitutions, as deriving the transition labels is still a meta-syntactic operation. This is a consequence of the fact that we are interested in a framework for reversibility. In this sense, the aforementioned calculi are closer to the explicit substitution of λ calculus than the late substitution we employ in π_l.

5.2 Correspondence between π_l and Rπ

Definition 27 *Let ϕ be a function that translates $R\pi$ processes into π_l, by recording all the substitutions into the environment and erasing all δ from the process.*

$$\phi(R \mid S) = \phi(R) \mid \phi(S) \qquad \phi(\nu y R) = \nu y \phi(R) \qquad \phi(\delta y R) = \phi(R)$$

$$\phi(\langle m', x[y/z], Q\rangle.m \rhd P) = [y/z] :: \phi(m \rhd P) \qquad \phi(\langle m', x[*/z], Q\rangle.m \rhd P) = [*/z] :: \phi(m \rhd P)$$

$$\phi(\_.m \rhd P) = \phi(m \rhd P) \qquad \phi(\langle\rangle \rhd P) = \varepsilon \cdot P.$$

Lemma 28 (Stong bisimulation between forward Rπ and its π_l-image)

1. *If $R \xrightarrow{\gamma} S$ then $\phi(R) \xrightarrow{\gamma} \phi(S)$.*
2. *If $T \xrightarrow{\gamma} U$ and $\forall R$ such that $\phi(R) = T$, $\exists S$ with $R \xrightarrow{\gamma} S$ and $\phi(S) = U$.*

We are now ready to show the correspondence between Rπ and π. The *forgetful map* that translates Rπ into π is given by the composition of the two translations above, and denoted with ϕ. The two lemmas extend to traces, $\longrightarrow^\star$.

Lemma 29 *If $P \longrightarrow^\star Q$ and $\phi(R) = P$ then there exists S such that $R \longrightarrow^\star S$ and $\phi(S) = Q$.*

Proof. Using the second part of the lemmas 26 and 28 we have that for each transition of a π process there exists one for its Rπ correspondent. The result follows from induction on the length of the trace.

Lemma 30 *If $\langle\rangle \rhd P \longrightarrow^\star R$ then $P \longrightarrow^\star \phi(R)$.*

6 Conclusions and Future Work

Reversible π calculus provides the mechanisms necessary for π calculus processes to backtrack to a previous state of the computation. It has a semantic that is symmetric for the forward and backward transitions and compositional with respect to the extrusion of the bound names. Our work relied on RCCS, a calculus that implements backtracking by adding memories to CCS processes. We aimed to define a similar calculus for π. The main difficulties arised from the differences between CCS and π calculus, in particular the name substitutions after an input and the extrusion of a bound names using the rules OPEN and CLOSE. For a distributed computation, the trace is not fixed, as the interleavings of transitions between concurrent processes change from a run to another. Therefore our calculus allows backtracking to follow any trace that is a valid one for the forward execution. The only transitions that have to respect an order when backtracking are the ones that are in a causal relation. Lastly, we also proved that our calculus is indeed a richer form, in the sense that we have reversibility, of the π calculus.

Our work is not yet complete. As we have already noted the causal relation we have is not satisfactory. Moreover, the operator δ is not compositional. On our Rπ calculus, we also want to define an equivalence relation between processes and prove a correspondence with event structures. These problems and future works are discussed below.

An equivalence relation. A weak bisimulation is an equivalence relation between processes that compares their behavior, as observed by the environment. In particular internal synchronizations are not observable. In a reversible calculus, as noted by [LMS10], a process can be bisimilar to the one of its derivatives. For instance $\langle 1\rangle \rhd \overline{a}.b \mid \langle 2\rangle \rhd a.\overline{b}$ is weakly bisimilar to $\langle 2, \overline{a}\rangle.\langle 1\rangle \rhd b \mid \langle 1, a\rangle.\langle 2\rangle \rhd \overline{b}$. However, we want an equivalence that differentiate such processes, and is interesting from the point of view of reversibility.

Our intuition is that *causal bisimulation* plays an important role in such an equivalence relation. It is employed in [BS95] and [DP95] to establish an equivalence between processes whom actions have the same causes. Consider the example of [Kie94], where the processes $P_1 = a \mid b$ and $P_2 = a.b \mid b.a$ are not causal bisimilar. Both processes can do either $\xrightarrow{a}\xrightarrow{b}$ or $\xrightarrow{b}\xrightarrow{a}$, but in P_1 the transitions are independent while for P_2 they are not. We can define a causal bisimulation for Rπ for both types of causality, by using the causal relation of section 4. Moreover, the processes are also distinguished if the transitions exhibit different types of causality as in $\nu y(\overline{a}\langle y\rangle \mid y(t))$ and $\nu y(\overline{a}\langle y\rangle.y(t))$.

Moving δ into the memory. Our LTS is useful when one wants to define a bisimulation, as part of the work on an equivalence relation between reversible processes. However, the LTS as it is, is not truly compositional. Consider a process δyR that does a transition on y. If we want to make the transition part of a synchronization, then another process S needs to know y. In our semantics such a process $(\delta yR \mid \delta yS)$ is not allowed. This in particular shows us that δ is not an operator suited for compositionality. We envisage, as a solution, to distribute the set of freed name of a process into the memory of its threads.

Event structures. An event structure [Win86] represents a process as a set of *events* on which two relations are defined: one for events that are in *conflict* and one for events that are causally dependent. The latter, denoted by $\leq$, defines a partial order on events. An event stands for the occurrence of an action. Then a *labeling function* associates to each event an action of the represented process. Events that are neither in conflict nor causally dependent, are *concurrent*. A *configuration* represents a possible run of the process.

A causal semantics for the π calculus in terms of event structures was given in [VY06] and [CVY12]. The authors translate a π process in a tuple (E, X), where E is an event structure and X is a set of bound names. The transition $E \xrightarrow{\beta} E/e$ reduces the event structure E to one in which the minimal event e and all events that are in conflict with it, have been removed. In order to deal with object causality, the configurations generated by the obtained event structure E have to verify some conditions. Otherwise, the transition is not allowed.

As future work, we can first move from an LTS on event structures to one on configurations, on which we think, is easier to implement reversibility. A backward transition consists then of adding one of the minimal

events back to a configuration. We can also add to each transition the labels $\overline{x}(\nu y)$ or $\overline{x}\langle\delta y\rangle$ based on the set X. Then we prove that there is an isomorphism between the LTS on configurations and the one of Rπ.

As part of showing a correspondence between event structures and Rπ, the weakly causal relation of definition 20 will be reconsidered. We want a causality that is more true to the one of event structures. For example, in a process $\nu y(\overline{a}\langle y\rangle \mid \overline{b}\langle y\rangle \mid y)$, the transition on y is depended on at least one of the extruders. If we reach the process $\delta y(\langle\star, \overline{a}\langle y\rangle\rangle \rhd 0 \mid \overline{b}\langle y\rangle \mid \langle\star, \overline{y}\rangle \rhd 0)$, then we can say that, for certain, b is not a cause of the transition on y. However, in our causal relation we ignore this information once the forward transition on b occurs.

Acknowledegements. Silvia Crafa and Ivan Lanese have provided insightful comments and suggestions on this work.

During this internship, I worked in the PPS research team part of the computer science laboratories of Université Paris Diderot, under the supervision of Jean Krivine and Daniele Varacca. We worked on reversibility for concurrent models, focusing on the π calculus. This is part of the research undergoing in the ANR project REVER. A more technical description of our results, which includes all the proofs, is available in [Cri12].

References

[BS95] Michele Boreale and Davide Sangiorgi. A fully abstract semantics for causality in the pi-calculus, 1995.

[Cri12] Ioana Cristescu. Reversible π-calculus. http://perso.ens-lyon.fr/ioana.domnina.cristescu, 2012.

[CVY12] Silvia Crafa, Daniele Varacca, and Nobuko Yoshida. Event structure semantics of parallel extrusion in the pi-calculus. In *FoSSaCS*, pages 225–239, 2012.

[DD93] Philippe Darondeau and Pierpaolo Degano. Refinement of actions in event structures and causal trees. *Theor. Comput. Sci.*, 118(1):21–48, 1993.

[DK04] Vincent Danos and Jean Krivine. Reversible communicating systems. In *CONCUR*, pages 292–307, 2004.

[DK05] Vincent Danos and Jean Krivine. Transactions in rccs. In *In Proc. of CONCUR, LNCS 3653*, pages 398–412. Springer, 2005.

[DP95] Pierpaolo Degano and Corrado Priami. Non-interleaving semantics for mobile processes. *Theoretical Computer Science*, 216:237–270, 1995.

[FMQ94] Gian Luigi Ferrari, Ugo Montanari, and Paola Quaglia. A pi-calculus with explicit substitutions: the late semantics. In *Proceedings of the 19th International Symposium on Mathematical Foundations of Computer Science 1994*, MFCS '94, pages 342–351, London, UK, 1994. Springer-Verlag.

[GW00] Philippa Gardner and Lucian Wischik. Explicit fusions. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science*, MFCS '00, pages 373–382, London, UK, 2000. Springer-Verlag.

[Hir] Daniel Hirschkoff. Handling substitutions explicitely in the pi-calculus.

[Kie94] Astrid Kiehn. Comparing locality and causality based equivalences. *Acta Inf.*, 31(8):697–718, 1994.

[LMS10] Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversing higher-order pi. In *CONCUR*, pages 478–493, 2010.

[Mil82] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[SW01] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes.* Cambridge University Press, 2001.

[VY06] Daniele Varacca and Nobuko Yoshida. Typed event structures and the π-calculus. *Electron. Notes Theor. Comput. Sci.*, 158:373–397, May 2006.

[Win86] Glynn Winskel. Event structures. In *Advances in Petri Nets*, pages 325–392, 1986.

Appendix

The LTS of Rπ

IN+
$$m \rhd x(z).P + Q \xrightarrow{\Phi(m),\star:m(x)(z)} \langle \star, x[*/z], Q\rangle.m \rhd P$$
IN-
$$\langle \star, x[*/z], Q\rangle.m \rhd P \xrightarrow{\Phi(m),\star:m(x)(z)^-} m \rhd x(z).P + Q$$

OUT+
$$m \rhd \overline{x}\langle y\rangle.P + Q \xrightarrow{\Phi(m),\star:\overline{m(x)}m(y)} \langle \star, \overline{x}\langle y\rangle, Q\rangle.m \rhd P$$
OUT-
$$\langle \star, \overline{x}\langle y\rangle, Q\rangle.m \rhd P \xrightarrow{\Phi(m),\star:\overline{m(x)}(m(y))^-} m \rhd \overline{x}\langle y\rangle.P + Q$$

COM+
$$\frac{R \xrightarrow{\phi_1,\star:\overline{x}\langle y\rangle} R' \qquad S \xrightarrow{\phi_2,\star:x(z)} S'}{R \mid S \xrightarrow{\phi_1,\phi_2:\tau} (R' \mid S')_{@(\phi_1,\phi_2,y)}}$$

COM-
$$\frac{R \xrightarrow{\phi_1,\star:\overline{x}\langle y\rangle^-} R' \qquad S \xrightarrow{\phi_2,\star:x(z)^-} S'}{(R \mid S)_{@(\phi_1,\phi_2,y)} \xrightarrow{\phi_1,\phi_2:\tau^-} R' \mid S'}$$

CONGR+-
$$\frac{R \equiv S \xrightarrow{\eta:\zeta} S' \equiv R'}{R \xrightarrow{\eta:\zeta} R'}$$

NEW+-
$$\frac{R \xrightarrow{\eta:\zeta} R'}{\nu y R \xrightarrow{\eta:\zeta} \nu y R'} y \notin \zeta$$

DELTA+-
$$\frac{R \xrightarrow{\eta:\zeta} R'}{\delta y R \xrightarrow{\eta:\zeta} \delta y R'} \zeta \neq \overline{x}\langle y\rangle, \overline{x}\langle y\rangle^-, \overline{x}(\delta z)$$

PAR+
$$\frac{R \xrightarrow{\eta:\gamma} R'}{R \mid S \xrightarrow{\eta:\gamma} R' \mid S} \mathrm{bn}(\gamma) \cap \mathrm{fn}(S) = \emptyset$$

PAR-
$$\frac{R \xrightarrow{\eta:\gamma^-} R'}{R \mid S \xrightarrow{\eta:\gamma^-} R' \mid S}$$

OPEN+
$$\frac{R \xrightarrow{\eta:\overline{x}\langle y\rangle} R'}{\nu y R \xrightarrow{\eta:\overline{x}(\nu y)} \delta y R'}$$

OPEN-
$$\frac{R \xrightarrow{\eta:\overline{x}\langle y\rangle^-} R'}{\delta y R \xrightarrow{\eta:\overline{x}(\nu y)^-} \nu y R'} y_\star \notin R'$$

OPEN DELTA+
$$\frac{R \xrightarrow{\eta:\overline{x}\langle y\rangle} R'}{\delta y R \xrightarrow{\eta:\overline{x}(\delta y)} \delta y R'}$$

OPEN DELTA-
$$\frac{R \xrightarrow{\eta:\overline{x}\langle y\rangle^-} R'}{\delta y R \xrightarrow{\eta:\overline{x}(\delta y)^-} \delta y R'} \uparrow y_\star \in R'$$

DELTA FREED-
$$\frac{R \xrightarrow{\eta:\overline{x}(\delta y)^-} R'}{\delta z R \xrightarrow{\eta:\overline{x}(\delta y)^-} \delta z R'} \uparrow y_\star \in \delta z R'$$

CLOSE+
$$\frac{R \xrightarrow{\phi_1,\star:\overline{x}(\nu y)} R' \qquad S \xrightarrow{\phi_2,\star:x(z)} S'}{R \mid S \xrightarrow{\phi_1,\phi_2:\tau} \nu y(R' \mid S')_{@(\phi_1,\phi_2,y)}} y \notin \mathrm{fn}(S)/\{z\}$$

CLOSE-
$$\frac{R \xrightarrow{\phi_1,\star:\overline{x}(\nu y)^-} R' \qquad S \xrightarrow{\phi_2,\star:x(z)^-} S'}{\nu y(R \mid S)_{@(\phi_1,\phi_2,y)} \xrightarrow{\phi_1,\phi_2:\tau^-} R' \mid S'} y \notin S'$$

CLOSE DELTA+
$$\frac{R \xrightarrow{\phi_1,\star:\overline{x}(\delta y)} R' \qquad S \xrightarrow{\phi_2,\star:x(z)} S'}{R \mid S \xrightarrow{\phi_1,\phi_2:\tau} \delta y(R' \mid S')_{@(\phi_1,\phi_2,y)}}$$

CLOSE DELTA-
$$\frac{R \xrightarrow{\phi_1,\star:\overline{x}(\delta y)^-} R' \qquad S \xrightarrow{\phi_2,\star:x(z)^-} S'}{\delta y(R \mid S)_{@(\phi_1,\phi_2,y)} \xrightarrow{\phi_1,\phi_2:\tau^-} R' \mid S'} y \notin S'$$

and the symmetric rules arising from the commutativity of $\mid$.