Reversible Communicating Concurrent Systems

Vincent Danos[1]* and Jean Krivine[2]

[1] Université Paris 7 & CNRS
[2] INRIA Rocquencourt

Abstract. One obtains in this paper a process algebra RCCS, in the style of CCS, where processes can backtrack. Backtrack, just as plain forward computation, is seen as a synchronization and incurs no additional cost on the communication structure. It is shown that, given a past, a computation step can be taken back if and only if it leads to a causally equivalent past.

1 Introduction

Backtracking means rewinding one's computation trace. In a distributed setting, actions are taken by different threads of computation, and no currently running thread will retain a complete description of the others past. Therefore, there is no guarantee that when a given thread goes back in its own local computation history, this will correspond to going back a step in the global computation trace. Of course, one could ask a thread willing to go back a step, to first verify that it was the last to take an action. But then all concurrent behaviour would be lost, not speaking about the additional communication machinery this choice would incur. On the other hand, letting any thread freely backtrack would result in losing the initial computation structure and reaching computation states which were formerly inaccessible. So, one has to strike a compromise here.

This is what we propose in this paper. A notion of distributed backtracking built on top of Milner's CCS [1] is provided. At any time each thread may either fork or synchronize with another thread. In both cases whatever action was taken is recorded in a memory. When the thread wants to rewind a computation step, it has to synchronize with either its siblings, in the case of a fork, or with its partner in the case of a former synchronization. Thus backtrack is considered also as a synchronization mechanism.

This mechanism can be construed as a distributed monitoring system and it meshes well with the specifics of the host calculus CCS. Backtrack doesn't involve any additional communication structure and we could obtain a syntax, termed RCCS, for *reversible* CCS, that stays really close to ordinary CCS.

There is another aspect in which the syntax seems to do well. The compromise it corresponds to has a clearcut theoretical characterization. Given a process and a past, one can show that the calculus allows backtrack along any

* *Corresponding author*: Équipe PPS, Université Paris 7 Denis Diderot, Case 7014, 2 Place Jussieu 75251 PARIS Cedex 05, Vincent.Danos@pps.jussieu.fr

causally equivalent past. Computation traces originating from a process are said to be causally equivalent when one can reshape one in the other by commuting successive concurrent actions or cancelling successive inverse actions.

A similar notion of computation trace equivalence exists in λ-calculus which Lévy could characterize by a suitable labelling system [2, 3]. Thus, a pretty good summary of the theoretical status of this backtracking mechanism, is to say that RCCS is a Lévy labelling for CCS. Two reduction paths will be equivalent if and only if they lead to the same process in RCCS. This is what we prove and it seems to be the best one can expect on the theoretical side.

To summarize the contribution, the present study proposes a syntax for reversible communicating concurrent systems, together with a charaterization, in terms of causally equivalent traces, of the exact amount of flexibility one allows in backtracking. One also explains how irreversible, or unbactrackable actions, can be included in the picture and a procedure of memory cleansing is introduced and proved to be sound.

Following Regev [4, 5], process algebras have been investigated recently for modeling biological systems. Since reversibility is the rule in biological interaction, the second author was thus prompted to look for a theoretical setup for distributed and reversible computations. Biological modeling in a former version of RCCS was explored. By that time soundness was proved directly, and the key connection to causal equivalence went unnoticed [6]. Future work, and in particular, applications to the synthesis of sound transactional mechanisms is discussed in the conclusions.

1.1 Related Work

Process algebras with backtracking were seen early to be valuable computational objects and studied independently by Prasad [7] and later by Bergstra et al. [8]. However, both had an exception model in mind, which while providing interesting programming constructs would not have any specific theoretical structure. Another well developed line of research, partly inspired by Lévy's work on causal equivalence in λ-calculus, and partly by the need for refined non-interleaving semantics, is that of the causal analysis of distributed systems [9–16]. However, the only concern here is forward computation. Causal analysis is thought of as a static analysis method or a theoretical measure of how concurrent a system is, and not as inducing some policy that threads should obey in order to backtrack soundly. In some sense, we present here a synthesis of these two lines of research which, to the best of our knowledge, were never brought together to interact.

2 RCCS

The plan to implement backtrack is to assign to each currently running thread an individual memory stack keeping track of past communications. This memory will also serve as a naming scheme and yield a unique identifier for the thread.

Upon doing a forward transition, the information needed for a potential roll-back will be pushed on the memory stack.

As said briefly in the introduction, two constraints are shaping the actual syntactic solution explained below. First the notion of past built in the memories has to have some degree of flexibility. Even if one could somehow record the complete succession of events during a distributed computation and only allow backward moves in whatever precise order was taken, this would induce fake causal dependencies on backward sequences of actions. Actions which could have been taken in any order would have to be undone in the precise incidental order in which they happened. So one should not be too adamant on the exact order in which things done have to be undone.

On the other hand the notion of past should not be too flexible. Because if it is, then one might be in danger of losing *soundness*, in that some reversing computations could give access to formerly unreachable states. Clearly, if actions are undone before whatever action they caused is, the result is not going to be consistent.

It turns out that the solution proposed here is at the same time consistent and maximally flexible. The final take on this will be a theorem proving that any two computation traces starting in a same state and reaching a same end state are causally equivalent, or in other words that one can be rearranged so as to obtain the other by commuting concurrent actions. Consistency will follow.

But, first of all we need a syntax to describe our processes and this is the matter to which we turn in the next subsection.

2.1 A Syntax for Backtrak

Simple processes. Simple processes are taken from CCS [1]:

Actions:	$\alpha ::= a \mid \bar{a} \mid \ldots$	Action on a channel
	$\mid \tau$	Silent action
Processes:	$P ::= 0$	End of process
	$\mid \sum \alpha_i.P_i$	Choice
	$\mid (P \mid P)$	Fork
	$\mid (a)P$	Restriction

Let us briefly remind that interaction consists only of binary synchronized communication. In a CCS system something happens when two processes are performing complementary actions at the same time, very much as a handshake. Recursive definitions can be dealt with, but they are not central to the point being made in this paper and we will do without them.

As the notation for choice suggests, the order in which choices add up is irrelevant. Simple processes will therefore be considered only up to *additive structural congruence* that is to say the equivalence relation generated by the following

usual identities:

$$
\begin{aligned}
P_1 + P_2 &\equiv P_2 + P_1 \\
(P_1 + P_2) + P_3 &\equiv P_1 + (P_2 + P_3) \\
P + 0 &\equiv P
\end{aligned}
$$

Monitored processes. In RCCS, simple processes are not runnable as such, only monitored processes are. This second kind of process is defined as follows:

$$
\begin{array}{llr}
\text{Memories:} & m ::= \langle\rangle & \text{Empty memory} \\
& \quad \mid \langle 1 \rangle \cdot m & \text{Left-Fork} \\
& \quad \mid \langle 2 \rangle \cdot m & \text{Right-Fork} \\
& \quad \mid \langle *, \alpha, P \rangle \cdot m & \text{Semi-Synch} \\
& \quad \mid \langle m, \alpha, P \rangle \cdot m & \text{Synch} \\
\\
\text{Monitored Processes:} & R ::= m \triangleright P & \text{Threads} \\
& \quad \mid (R \mid R) & \text{Product} \\
& \quad \mid (a)R & \text{Restriction}
\end{array}
$$

To sort visually our two kinds of processes, the simple ones will be ranged over by P, Q, ... while the other ones will be ranged over by R, S, ...

Sometimes, when it is clear from the context which kind of process is being talked about, we will say simply process in place of monitored process.

As one may readily see from the syntax definition, a monitored process can be uniquely constructed from a set of terms of the form $m \triangleright P$, which we will call its *threads*. In a thread $m \triangleright P$, m represents a memory carrying the information that this process will need in case it wants to backtrack. That memory is organized as a stack with the last action taken by the thread sitting on the top together with additional information that we will comment on later. There is an evident *prefix ordering* between memories which will be denoted simply $\leq$.

As an example we can consider the following monitored process:

$$R = \langle 1 \rangle \langle *, a, 0 \rangle \triangleright b.\bar{c}.0 \mid \langle 2 \rangle \langle *, a, 0 \rangle \triangleright c.0$$

It consists of two threads, namely $S_1 = \langle 1 \rangle \langle *, a, 0 \rangle \triangleright b.\bar{c}.0$ and $S_2 = \langle 2 \rangle \langle *, a, 0 \rangle \triangleright c.0$. Taking a closer look at S_1, we see a fork action $\langle 1 \rangle$ sitting on top of its memory stack, indicating that the last interaction the thread took part in was a fork. Below one finds $\langle *, a, 0 \rangle$ indicating that the penultimate action was an a action exchanged with an unidentified partner $*$. That part of the past of S_1 is shared by S_2 as well. Actually, they both can be obtained from a same process $\langle\rangle \triangleright a.(b.\bar{c}.0 \mid c.0)$ as will become evident when we have a precise notion of computation.

Coherent processes. Not all monitored processes are going to be of interest. Since memories are also serving as a naming scheme for threads, they should better be unique. Actually we can ask for a little more than memory uniqueness and this is what we call *coherence* and proceed now to define.

Definition 1 *Coherence is the smallest symmetric relation such that:*

– $\forall i, j : i \neq j \Rightarrow \langle i \rangle \cdot m \frown \langle j \rangle \cdot m$;
– $\forall m_1, m_2 : m \frown m' \Rightarrow m_1 \cdot m \frown m_2 \cdot m'$.

Memories are coherent if they branch on a fork.

Definition 2 *A* monitored *process is said to be* coherent *if its memories are pairwise coherent.*

Coherence implies in particular that memories are *unique* in a coherent term, since coherence is irreflexive. But, as said, it is a bit stronger than that. For instance $\langle *, b \rangle \langle 1 \rangle \triangleright P \mid \langle *, a \rangle \langle 1 \rangle \triangleright Q$ is not coherent, even if its two memories are distinct.

Define inductively the *fork structure* $\lambda(m)$ of a memory m:

$$\begin{aligned} \lambda(\langle\rangle) &= \langle\rangle \\ \lambda(\langle *, a, Q \rangle \cdot m) &= \lambda(\langle m', a, Q \rangle \cdot m) = \lambda(m) \\ \lambda(\langle i \rangle \cdot m) &= \langle i \rangle\, \lambda(m) \end{aligned}$$

Clearly $m \frown m'$ if and only if $\lambda(m) \frown \lambda(m')$, so the fork structure can be used to assert coherence as well.[3]

2.2 From RCCS to CCS and back

Our calculus is clearly only a "decoration" of CCS. That decoration can be erased by way of the forgetful map $\varphi : \text{RCCS} \to \text{CCS}$, defined as:

$$\begin{aligned} \varphi(m \triangleright P) &= P \\ \varphi(R \mid S) &= \varphi(R) \mid \varphi(S) \\ \varphi((a)R) &= (a)\varphi(R) \end{aligned}$$

Conversely one can lift any CCS process to RCCS with the map $\ell(P) = \langle\rangle \triangleright P$.

One has that $\varphi \circ \ell$ is the identity but not the converse ! If we go back to our first example, we see that $\ell(\varphi(R)) = \langle\rangle \triangleright (b.\bar{c}.0 \mid c.0)$. The transformation $\ell \circ \varphi$ is blanking all memories in a monitored process.

2.3 RCCS structural congruence

We now want to extend the additive structural congruence defined earlier on simple processes, to monitored processes. The most important additional rule is the following:

$$m \triangleright (P \mid Q) \equiv (\langle 1 \rangle \cdot m \triangleright P \mid \langle 2 \rangle \cdot m \triangleright Q) \tag{1}$$

[3] An additional coherence requirement could be that for any memory m occurring in a process R, $\lambda(m)$ is exactly the forking address of m in R, where by the forking address of m in R we mean the binary string over $\{1, 2\}$ locating the thread with memory m in R. For an example of a process violating this extra condition, consider $R = (\langle *, a \rangle \langle 1 \rangle \langle 1 \rangle \triangleright 0 \mid \langle 2 \rangle \triangleright 0)$, indeed one has that $\lambda(\langle *, a \rangle \langle 1 \rangle \langle 1 \rangle)$ is not $\langle 1 \rangle$ as it should.

It explains how memory is distributed when a process divides in two sub-threads. We see that each sub-thread inherits the father memory together with a fork number indicating which of the two sons the thread is. Another rule we need is:

$$m \triangleright (a)P \equiv (a)m \triangleright P \tag{2}$$

Both rules have a natural left to right orientation corresponding to forward computation. Take note also that both these memory rearrangements are invisible in CCS, *e.g.*, $\varphi(m \triangleright (P \mid Q))$ is actually equal to $\varphi(\langle 1 \rangle \cdot m \triangleright P \mid \langle 2 \rangle \cdot m \triangleright Q)$.

Structural congruence on monitored processes is then obtained by combining these two identities together with additive congruence. In other words, two processes are *structurally equivalent* if related by the smallest equivalence relation generated by the identities (1), (2) above, by additive congruence identities, and closed under all syntactical constructs.

Lemma 3 *Consistency is preserved by structural congruence.*

The only case where memories are modified is in using identity (1) where a given m is split in $\langle 1 \rangle \cdot m$ and $\langle 2 \rangle \cdot m$. By definition $\langle 1 \rangle \cdot m \frown \langle 2 \rangle \cdot m$ and an m' is coherent with m iff it is coherent with both $\langle 1 \rangle \cdot m$ and $\langle 2 \rangle \cdot m$. □

Usual identities associated with the product, such as $P \mid Q \equiv Q \mid P$ are not considered here because memories are using the actual product structure of the term to memorize the fork actions. A quotient would force the manipulation of terms up to tree isomorphisms on memories. That is possible and perhaps even interesting if one wants a more mathematical view on the calculus, but certainly results in a less convenient syntax.

2.4 Transition Rules

It remains to define a proper notion of computation. To this effect, we use a labelled transition system, LTS for short, that is to say a family of binary relations over monitored processes. Specifically, transitions are of the form:

$$R \xrightarrow{\mu:\zeta} S,$$

where R, S are monitored processes, ζ is a *directed action*, that is either a forward action or a backward action, while μ is an *identifier*, that is either a memory or a pair of memories:

$$\begin{array}{lr} \zeta ::= \alpha \mid \alpha_* & \text{Directed Actions} \\ \mu ::= m \mid m, m & \text{Identifiers} \end{array}$$

Basic Rules. First we have the basic transitions concerning threads:

$$\text{act}\ \frac{}{m \triangleright \alpha.P + Q \xrightarrow{m:\alpha} \langle *, \alpha, Q \rangle \cdot m \triangleright P} \qquad \frac{}{\langle *, \alpha, Q \rangle \cdot m \triangleright P \xrightarrow{m:\alpha_*} m \triangleright a.P + Q}\ \text{act}_*$$

The first transition is a forward transition whereby the thread does an action α. A memory triplet $\langle *, \alpha, Q\rangle$ containing this action, as well as the leftover part Q is pushed on top of the memory, and the thread proceeds further down its code. The first element in the memory triplet '$*$' stands for an unidentified partner. The transition itself is indexed by the action α so that it can be synchronized with a transition bearing a complementary action, and the memory m, which will be used to identify oneself in the synchronization.

The second transition goes backward. The process is now undoing an action which is on the top of its memory and is therefore the last action it took. As we discussed already, many actions may have happened in the meantime in the global computation, but none originated from that particular thread, or they were undone before. Backward actions are treated on a par with forward actions and in particular, backtracking also involves communicating.

Contextual Rules. We have seen what transitions a thread was able to trigger. These transitions can also be done in a context:

$$\text{par-l}\ \frac{R \xrightarrow{\mu:\zeta} R'}{R \mid S \xrightarrow{\mu:\zeta} R' \mid S} \qquad \frac{R \xrightarrow{\mu:\zeta} R'}{S \mid R \xrightarrow{\mu:\zeta} S \mid R'}\ \text{par-r}$$

$$\text{res}\ \frac{R \xrightarrow{\mu:\zeta} R' \quad \zeta \not\ni a}{(a)R \xrightarrow{\mu:\zeta} (a)R'} \qquad \frac{R_1 \equiv R \xrightarrow{\mu:\zeta} R' \equiv R_1'}{R_1 \xrightarrow{\mu:\zeta} R_1'}\ \equiv$$

where in the (res) rule, $\zeta \not\ni a$ means that ζ is none of a, $\bar{a}$, a_* or $\bar{a}_*$. The last rule says that one can freely choose a representative in the structural congruence class before triggering a transition. It is used to push memories down threads using identities (1), (2) with their natural orientation.

Synchronization Rules. We end the description of transitions with the forward and backward synchronization rules.

Both kinds of synchronizations use a notion of *address instanciation* in monitored processes. Given a monitored process R and memories m_1, m_2, a new process $R_{m_2@m_1}$ is obtained by replacing in R all memories of the form $\langle *, \alpha, Q\rangle \cdot m_1$ with $\langle m_2, \alpha, Q\rangle \cdot m_1$. This is used in forward synchronization to let the thread know the name of the other thread it synchronized with.

The complete definition is as follows:

$$\begin{array}{lll}
((a)R)_{m_2@m_1} & := (a)(R_{m_2@m_1}) & \\
(R \mid S)_{m_2@m_1} & := (R_{m_2@m_1} \mid S_{m_2@m_1}) & \\
(\langle\rangle \triangleright P)_{m_2@m_1} & := \langle\rangle \triangleright P & \\
(\langle i\rangle \cdot m \triangleright P)_{m_2@m_1} & := \langle i\rangle \cdot m \triangleright P & \\
(\langle m', \alpha, Q\rangle \cdot m \triangleright P)_{m_2@m_1} & := \langle m', \alpha, Q\rangle \cdot m \triangleright P & \\
(\langle *, \alpha, Q\rangle \cdot m \triangleright P)_{m_2@m_1} & := \langle *, \alpha, Q\rangle \cdot m \triangleright P & \text{if } m \neq m_1 \\
(\langle *, \alpha, Q\rangle \cdot m \triangleright P)_{m_2@m_1} & := \langle m_2, \alpha, Q\rangle \cdot m_1 \triangleright P & \text{if } m = m_1
\end{array}$$

When R is a coherent process, there can be at most one memory of the appropriate form, and therefore $R_{m_2@m_1}$ and R differ at most at that particular location. With this definition in place, we can formulate neatly the synchronization rules:

$$\frac{R \stackrel{m_1:\alpha}{\longrightarrow} R' \qquad S \stackrel{m_2:\bar{\alpha}}{\longrightarrow} S'}{R \mid S \stackrel{m_1,m_2:\tau}{\longrightarrow} {R'}_{m_2@m_1} \mid {S'}_{m_1@m_2}}\ \mathsf{syn}$$

$$\frac{R \stackrel{m_1:\alpha_*}{\longrightarrow} R' \qquad S \stackrel{m_2:\bar{\alpha}_*}{\longrightarrow} S'}{R_{m_2@m_1} \mid S_{m_1@m_2} \stackrel{m_1,m_2:\tau_*}{\longrightarrow} R' \mid S'}\ \mathsf{syn}_*$$

Backward synchronization discussed. As one can see in the definition, backward synchronization is also a communication.

One important thing to notice is that once a thread T in a process R has synchronized forwardly with another using (syn), then it is instanciated and the resulting $T_{m_1@m_2}$ can no longer backtrack without rule (syn_*). In other words, once one has synched an action, the only way to go backwards is to synch again ... backwards. One doesn't roll back a synchronization all alone.

Let us illustrate this "locking effect" with the simplest possible example. Starting with $R = \langle\rangle \rhd a.0 \mid \bar{a}.0$ one has the following computation:

$$\langle\rangle \rhd (a.0 \mid \bar{a}.0) \equiv \langle 1\rangle \rhd a.0 \mid \langle 2\rangle \rhd \bar{a}.0 \stackrel{\langle 1\rangle,\langle 2\rangle:\tau}{\longrightarrow} \langle\langle 2\rangle, a, 0\rangle \cdot \langle 1\rangle \rhd 0 \mid \langle\langle 1\rangle, \bar{a}, 0\rangle \cdot \langle 2\rangle \rhd 0$$

Both threads are locked together, and the only way backward for them is to synchronize again:

$$\frac{\mathsf{act}_*\ \dfrac{}{\langle *, a, 0\rangle \cdot \langle 1\rangle \rhd 0 \stackrel{\langle 1\rangle:a_*}{\longrightarrow} \langle 1\rangle \rhd a.0} \qquad \dfrac{}{\langle *, \bar{a}, 0\rangle \cdot \langle 2\rangle \rhd 0 \stackrel{\langle 2\rangle:\bar{a}_*}{\longrightarrow} \langle 1\rangle \rhd a.0}\ \mathsf{act}_*}{\langle\langle 2\rangle, a, 0\rangle \cdot \langle 1\rangle \rhd 0 \mid \langle\langle 1\rangle, \bar{a}, 0\rangle \cdot \langle 2\rangle \rhd 0 \stackrel{\langle 1\rangle,\langle 2\rangle:\tau_*}{\longrightarrow} \langle 1\rangle \rhd a.0 \mid \langle 2\rangle \rhd \bar{a}.0}\ \mathsf{syn}_*$$

Thus the intuitive picture here is that a backward synch is obtained by letting loose two threads and see if they do complementary backward actions.

Relation to CCS. Write $R \rightarrow^\star S$ if there exists a trace from R to S in RCCS and likewise $P \rightarrow^\star Q$ if there exists a trace from P to Q in CCS.

Lemma 4 *If $P \rightarrow^\star Q$ and $\varphi(R) = P$, then $R \rightarrow^\star S$ for some S such that $Q = \varphi(R)$.*

To see this, it is enough to forget about backward actions in the LTS. Then it becomes the ordinary LTS for CCS with memories being useless additional notations. □

Thus any RCCS term "above" P can simulate P's behaviour in a purely forward manner. Of course, what one would like some form of converse guaranteeing the consistency of the backtracking mechanism. We will provide such a converse later in the paper (corollary 1).

Coherence. The transition system which we defined gives means of defining computation traces, also sometimes called the "reduction semantics". Especially convenient is the fact that one doesn't have to work up to associativity and commutativity of the product and therefore gets a simple naming scheme for threads. Coming to this matter, we have to verify that transitions don't disturb this naming scheme and preserve coherence.

Lemma 5 *If $R \xrightarrow{\mu:\zeta} S$ and R is coherent, then so is S.*

Basic transitions are only adding to, or chopping off the stack, a triplet of the form $\langle _, \alpha, Q\rangle$. Doing so leaves the memory forking structure invariant. None of the other rules, with the exception of the structural congruence rule and the synchronization rules modify memories. The case of structural congruence was already dealt with in a previous lemma. Finally, synchronization only instantiate or de-instantiate one memory top at a time (because the process undergoing this is coherent), and this is again a transformation that leaves the forking structure of the memory unchanged. $\square$

As a consequence, any process obtained from some $\ell(P)$ with P a CCS process, is coherent. Thereafter we will assume all processes to be coherent.

3 Interlude

Let us have a short example before going to the causality section. Three convenient bits of notation will ease the reading. First, and although only the canonical notation gives a robust naming scheme, we use fork structures $\lambda(m)$ instead of full memories m as identifiers. Second we use n-ary forks though this is not official notation. Last, when the choice left over in a choice action is 0, we just don't write it. That said we can run our example:

$$\begin{aligned}
\langle\rangle \triangleright (x.\underline{a}.Q \mid \bar{y}.\bar{x} \mid y.P) \equiv\ & \langle 1\rangle \triangleright x.\underline{a}.Q \mid \langle 2\rangle \triangleright \bar{y}.\bar{x} \mid \langle 3\rangle \triangleright y.P \\
\xrightarrow{\langle 2\rangle,\langle 3\rangle:\tau}\ & \langle 1\rangle \triangleright x.\underline{a}.Q \mid \langle\langle 3\rangle, \bar{y}\rangle\langle 2\rangle \triangleright \bar{x} \mid \langle\langle 2\rangle, y\rangle\langle 3\rangle \triangleright P \\
\xrightarrow{\langle 1\rangle,\langle 2\rangle:\tau}\ & \langle\langle 2\rangle, x\rangle\langle 1\rangle \triangleright \underline{a}.Q \mid \langle\langle 1\rangle, \bar{x}\rangle\langle\langle 3\rangle, \bar{y}\rangle\langle 2\rangle \triangleright 0 \mid \langle\langle 2\rangle, y\rangle\langle 3\rangle \triangleright P \\
\xrightarrow{\langle 1\rangle:\bar{a}}\ & \langle *, \underline{a}\rangle\langle\langle 2\rangle, x\rangle\langle 1\rangle \triangleright Q \mid \langle\langle 1\rangle, \bar{x}\rangle\langle\langle 3\rangle, \bar{y}\rangle\langle 2\rangle \triangleright 0 \mid \langle\langle 2\rangle, y\rangle\langle 3\rangle \triangleright P
\end{aligned}$$

One sees how the repeated synchronizations create a causal bottleneck: the synch on y caused the synch on x which in turn caused the a action. Therefore, this sequence of three events is completely sequential and cannot be rearranged in any other order. Nothing is concurrent in this computation. That much is obvious. Less obvious is the fact that one doesn't need to go through the whole computation to realize this. It can be read off the stacks directly in the final process. The rightmost thread wants to backtrack on y with $\langle 2\rangle$ (identifying the middle thread), the middle thread wants to backtrack on $\bar{x}$ with $\langle 1\rangle$ (identifying the leftmost thread), while the leftmost thread wants to backtrack on a all alone.

The point of the next section is to prove that this property holds in general. Backtracking an event is possible when and only when a causally equivalent trace

would have brought this event as the last one. In our example there is no other equivalent trace and indeed no other action than a can be backtracked.

This example illustrates another point made later. Suppose a is declared to be unbacktrackable, or irreversible, then the process now behaves as $Q \mid P$. The subject matter of the last section will be to integrate such irreversible actions in RCCS.

4 Causality

4.1 Transitions and Traces

We need first to go through a few standard definitions.

Recall that a transition t is given by a triplet $R \xrightarrow{\mu:\zeta} S$ with R, S monitored processes, ζ an identifier (that is either a memory or a pair of memories) and μ a directed action. One says then that R is the *source* and S the *target* of t and that S and R are its *ends*. Transitions will be ranged over in this section with t, t', ... and similar symbols. Two transitions are said to be *coinitial* if they have the same source, *cofinal* if they have the same target, *composable* if the source of the second is the target of the first. A transition is said to be *forward* or *backward* depending on whether its associated action μ is forward or backward.

Sequences of pairwise composable transitions will be called *computation traces* or simply traces, and will be ranged over by r, s, etc. All notions just defined for transitions extend readily to traces. In particular a trace will be said forward if all transitions it is composed of are forward. The empty trace with source R is denoted by ϵ_R and when s and s' are composable, their composition is written $s; s'$. A derivable transition is one than can be derived using the LTS of the second section. We have had examples of these already. From now on we will assume all transitions and traces to be derivable and with coherent ends (equivalently coherent sources) since these are the ones of interest.

If t is such a transition with identifier $\mu = m_1, m_2$, then it must have been obtained by a synchronization, and hence m_1 and m_2 have to be distinct, since one is assuming all processes to be coherent. So identifiers can be considered as sets (with either one or two elements) and be handled with set-theoretic operations. Transitions involving a one element memory will be termed *unary*, others will termed *binary*.

Another noteworthy point is that a derivable transition is essentially derivable in a unique way. The only freedom is in the application of the $\equiv$ rule and apart from the additive congruence, it only involves pushing memories past restrictions and products.

4.2 Causal equivalence

Lemma 6 (Loop) *For any forward transition $t : R \xrightarrow{\mu:\alpha} S$ there exists a backward transition $t_* : S \xrightarrow{\mu:\alpha_*} R$ and conversely.*

Given s a forward trace, one can then obtain s_* a backward trace, by applying repeatedly the lemma and reversing all transitions in s.

Definition 7 *Let* $t_1 = R \xrightarrow{\mu_1:\zeta_1} S_1$ *and* $t_2 = R \xrightarrow{\mu_2:\zeta_2} S_2$ *be two coinitial transitions,* t_1 *and* t_2 *are said to be* concurrent *if* $\mu_1 \cap \mu_2 = \varnothing$.

Lemma 8 (Square) *Let* $t_1 = R \xrightarrow{\mu_1:\zeta_1} S_1$ *and* $t_2 = R \xrightarrow{\mu_2:\zeta_2} S_2$ *be concurrent transitions, there exists two cofinal transitions* $t_2/t_1 = S_1 \xrightarrow{\mu_2:\zeta_2} T$ *and* $t_1/t_2 = S_2 \xrightarrow{\mu_1:\zeta_1} T$.

Definition 9 *Keeping with the notation of the square and loop lemmas above, one defines the* causal equivalence*, written* $\sim$*, as the least equivalence relation between traces closed under composition and such that:*

$$t_1; t_2/t_1 \sim t_2; t_1/t_2 \quad (3)$$

$$t; t_* \sim \epsilon_R \quad (4)$$

$$t_*; t \sim \epsilon_R \quad (5)$$

As said earlier, this is the analogue in CCS of the Berry and Lévy's notion of equivalence of computation traces "by permutation" [2]. The "square" part of the definition was already considered by Boudol and Castellani [11, 13, 14] and they were well aware of the connection to Lévy's theory.

4.3 RCCS as a Lévy labelling

Theorem 1 *Let* s_1 *and* s_2 *be coinitial, then* $s_1 \sim s_2$ *iff* s_1 *and* s_2 *are cofinal.*

By construction if $s_1 \sim s_2$, then s_1 and s_2 must be coinitial and cofinal, so the only if part is trivial. The if part is not. We first need a lemma.

Lemma 10 *Let* s *be a trace, there exists* r, r' *both forward such that* $s \sim r_*; r'$.

We prove this by lexicographic induction on the length of s, and the distance to the beginning of s of the earliest pair of transitions in s contradicting the property. If there is no such contradicting pair, then we are done. If there is one, say $t'; t_*$, and $\mu(t') \cap \mu(t_*) = \varnothing$, then t' and t_* can be swapped by virtue of the square lemma, resulting in a later earliest contradicting pair, and by induction the result follows since swapping keeps the length constant.

Suppose now $\mu(t') \cap \mu(t_*) \neq \varnothing$, and suppose further that t' is unary and write m for $\mu(t')$. By definition of a unary transition t' is pushing some triplet $\langle *, \alpha, Q \rangle$ on top of the stack at m, a triplet which t_* is then popping. This forces t_* to be unary as well, since there is no partner it may synch backwards with. But then $t = t'$ and one can apply the loop lemma, hence the total length decreases and again by induction the result follows.

Suppose finally t' is binary, then t_* has to be as well, because of the locking effect discussed above, and again $t = t'$ and the loop lemma applies and so does the inductive hypothesis.□

Intuitively, the lemma says that, up to causal equivalence, one can always reach for the maximum freedom of choice, going backward, and only then go forward. Even more intuitively, one could picture such traces as parabolas, the process first draws potential energy from its memories and only then moves onward. So let us call such traces *parabolic*.[4]

Our lemma already has an interesting corollary, namely that backtracking is consistent with respect to CCS in the weaker sense that RCCS will not generate traces reaching processes the projections of which are unreachable in CCS.

Corollary 1. *$P \rightarrow^\star \varphi(R)$ if and only if $\ell(P) \rightarrow^\star R$.*

The only if part is easy, basically saying that forward CCS computations can be simulated by RCCS ones, a fact that was recorded as an earlier lemma. The if part follows from the lemma above. Indeed, suppose s is an RCCS trace with source $\ell(P)$ and target R, then the lemma says $s \sim r_*; r'$ for some well chosen forward traces r and r', but then surely r must be empty since its source is $\ell(P)$ a process that has an empty memory and therefore is incapable of backtrack. So that there is forward trace r' equivalent to s. Since forward computations coincide in both systems, P reduces to $\varphi(R)$. □[5]

Lemma 11 *Let s_1, s_2 be coinitial and cofinal traces and s_2 be forward, then there exists a forward trace s_1', shorter than s_1 and such that $s_1' \sim s_1$.*

We prove this last lemma by induction on the length of s_1. If s_1 is forward we are done. Else by lemma 10 we may suppose s_1 is parabolic. Let $t_*; t'$ be the only two successive transitions in s_1 with opposite directions and call μ the identifiers of t_*. Whatever t_* takes off the memories in μ has to be put back later down s_1 by some forward transition. Else, because s_2 is forward the difference will stay visible. Call t the earliest such transition in s_1. For the same reason, that transition t has to be the exact inverse of t_*. One can then bubble up t to meet with t_*. Indeed, if a transition is conflicting with t on its way to t_*, by construction it must be some forward transition t'' the application of which results in a memory in μ, which is impossible since no backward transitions has happened since t_*. So by applying repeatedly the square lemma, and then applying (5), one obtains a shorter equivalent s_1' and conclude by induction. □

Proof of theorem. With our lemmas 10 and 11 in place, we can handle the theorem.

[4] The converse decomposition as $r; r'_*$ would not work. Indeed, a backward transition can create some new forward possibility. Consider for instance $\langle *, a, b.P\rangle \triangleright 0$ that can go to $\langle *, b, a.0\rangle \triangleright P$ in a backward step followed by a forward one. There is no way these two steps can be swapped or be cancelling each other as in the argument above.

[5] The fact that $\ell(P)$ has an empty memory is essential in the statement. We may consider again the same example $\langle *, a, P\rangle \triangleright 0$ which can move, while its projection $\varphi(\langle *, a, P\rangle \triangleright 0) = 0$ can't. One can also make a note that this statement closely resembles an adjoint situation, where the lifting map ℓ would be left adjoint to the forgetful map φ.

Let then s_1, s_2 be two traces with same source and target. We prove these are causally equivalent, using a lexicographic induction on a pair consisting of the sum of the lengths of s_1 and s_2, and the depth of the earliest disagreement between them. By lemma 10, we may suppose these are parabolic. Call t_1 and t_2 the earliest transitions were they disagree. There are three main cases in the argument depending on whether these are forward or backward.

1. If t_1 is forward and t_2 is backward, one has $s_2 = r_*; t_2; u$ and $s_1 = r_*; t_1; v$ for some r, u, v. Lemma 11 applies to $t_1; v$ which is forward, and $t_2; u$ which is parabolic, so $t_2; u$ has a shorter forward equivalent, and so s_2 has a shorter equivalent and the result follows by induction.

2. If t_1, t_2 are both forward, they can't possibly be conflicting.

Suppose they take different additive choices in a same thread, say $m \triangleright \alpha.P + \alpha'.P' + Q$, respectively pushing on the memory $\langle *, \alpha, \alpha'.P' + Q\rangle$ and $\langle *, \alpha', \alpha.P + Q\rangle$. Since by assumption t_1 and t_2 are not the same transition, P and Q have to be different (not even additively congruent that is), and this difference would stay visible onward contradicting s_1 and s_2 being cofinal.

So whenever they work on a same thread they have to make the same additive choices. In particular, this rules out the case where they are both unary, and also the case where they are both binary and working on the same two threads. Let us check the other cases. If one of the transitions is unary and the other is binary, their actions can't coincide on their common memory, since one will push a triplet of the form $\langle *, a, Q\rangle$ and the other will push one of the form $\langle m, a, Q'\rangle$ for some $m \neq *$. If both are binary and only intersect on one memory, then again the triplets are distinct since the other partner identifiers, say m_1 and m_2 are different, because these identifiers come from different threads and all processes are assumed coherent. In any case, there would be a visible and permanent difference.

So we know they are indeed concurrent. Call μ_2 the identifier of t_2. Since t_2 is forward and s_2 is parabolic, whatever t_2 pushes to μ_2 is staying there onward in s_2. Hence, there must be a transition at μ_2 in s_1 as well, call it t'_2, so that $s_1 = s; t_1; u; t'_2; v$. That transition has to be of the same arity as t_2, and with $\mu(t'_2) = \mu_2$, else again the difference stays visible onward. Since s_1 is parabolic as well, all transitions in u, standing in between t_1 and t'_2 are forward and one can apply repeatedly the square lemma and bubble up t'_2, to get s'_1 a trace equivalent to s_1 and of the form $s'_1 = s; t_1; t'_2; u; v$. A last application of the same square lemma yields an equivalent s''_1 with a later earliest divergence with s_2 and same length so that one can call on the induction hypothesis.

3. Suppose finally both t_1 and t_2 are backward. They can't possibly be conflicting either, because of the locking effect, so they must be concurrent. Either actions undone by t_2 are redone later in s_2, in which case one argues as in lemma 11 and shortens s_2 into an equivalent s'_2, or they are not redone and therefore these same actions must be undone also in s_1, else there would be a visible difference between their visible targets, and one argues as in 2 above. □

5 Irreversible Actions

Having finally reached for the theoretical property we wanted, we now turn to a more practical issue, namely that of integrating unbactrackable or irreversible actions in RCCS. These will be written $\underline{a}$, $\underline{\bar{a}}$, etc. The transition system is adapted by adding rules for the new actions:

$$\text{commit}\ \frac{}{m \triangleright \kappa.P + Q \xrightarrow{m:\kappa} \langle\circ\rangle \cdot m \triangleright P} \qquad \frac{R \xrightarrow{m_1:\kappa} R' \qquad S \xrightarrow{m_2:\bar{\kappa}} S'}{R \mid S \xrightarrow{m_1,m_2:\tau_\circ} R' \mid S'}\ \text{syn}_\circ$$

with κ ranging over irreversible actions. Since an irreversible action will never be backtracked, there is no need to remember anything, and we use instead of the usual triplet, a placeholder $\langle\circ\rangle$. For the same reason, there is no rule inverse to (commit), and no longer a need to instanciate R' and S' in the synchronization rule. One sees that the syntax hardly changes, and that when all actions are taken to be irreversible, it essentially becomes CCS. What was proved in the last section remains valid.

We have seen in Section 3 that all actions that caused $\underline{a}$ became *de facto* unbacktrackable, even if they were themselves reversible. It seems therefore interesting to understand how irreversible actions can "domino-effect" in a process and this is what we do now.

5.1 Propagating irreversible actions

Let $\mathcal{M}_R$ stand for the set of occurrences of memories in R. Say a memory is *locked* in R if, whenever $R \rightarrow^\star S$, there is an $m' \in \mathcal{M}_S$ such that $m \leq m'$. In other words, m is locked if no trace can take from it. Let $\mathcal{L}_R$ be the subset of $\mathcal{M}_R$ formed of these locked memories.

Lemma 12 *$\mathcal{L}_R$ satisfies:*

1. $\langle\circ\rangle \cdot m \in \mathcal{M}_R \Rightarrow \langle\circ\rangle \cdot m \in \mathcal{L}_R$
2. $m \in \mathcal{L}_R,\ m' \leq m \Rightarrow m' \in \mathcal{L}_R$
3. $\langle m, \alpha, P\rangle \cdot m' \in \mathcal{L}_R \Rightarrow \langle m', \bar{\alpha}, Q\rangle \cdot m \in \mathcal{L}_R$
4. $\langle i\rangle \cdot m \in \mathcal{L}_R \Rightarrow \langle j\rangle \cdot m \in \mathcal{L}_R$

Points 1 and 2 are obvious, point 3 is saying that in order to backtrack a synch, one needs to do a synch, a fact which we called the locking effect, and point 4 is saying that to undo some action in $\langle i\rangle \cdot m$, one has first to fuse back with one's former sibling in the fork. □

Coherence alone is not strong enough for these closure properties to capture all locked memories. For instance, m is locked in $\langle 1\rangle \cdot m \triangleright P$ simply because the fork sibling is missing. However the converse holds for CCS reachable processes.

5.2 An example

To illustrate how the lemma works in practice to simplify processes, let us consider the usual encoding of internal sum in CCS:

$$R = \langle\rangle \triangleright (x)(\bar{x}.0|x.\underline{a}.P|x.\underline{b}.Q),$$

with x chosen *reversible* and a, b irreversible. With the usual notational simplifications, and representing locked memories with heavy angles $\lbag _ \rbag$:

$$\begin{aligned} R \rightarrow^\star (x)(\langle\langle 2\rangle, \bar{x}\rangle\langle 1\rangle \triangleright 0 \mid \lbag \circ \rbag\langle\langle 1\rangle, x\rangle\langle 2\rangle \triangleright P \mid \langle 3\rangle \triangleright x.\underline{b}.Q) =: R_a \\ R \rightarrow^\star (x)(\langle\langle 3\rangle, \bar{x}\rangle\langle 1\rangle \triangleright 0 \mid \langle 2\rangle \triangleright x.\underline{a}.P \mid \lbag \circ \rbag\langle\langle 1\rangle, x\rangle\langle 3\rangle \triangleright Q) =: R_b \end{aligned}$$

Using clauses 2, 3, 4, one can tag further memories as locked:

$$\begin{aligned} R_a &= (x)(\lbag\langle 2\rangle, \bar{x}\rbag\lbag 1\rbag \triangleright 0 \mid \lbag \circ \rbag\lbag\langle 1\rangle, x\rbag\lbag 2\rbag \triangleright P \mid \lbag 3\rbag \triangleright x.\underline{b}.Q) \\ R_b &= (x)(\lbag\langle 3\rangle, \bar{x}\rbag\lbag 1\rbag \triangleright 0 \mid \lbag 2\rbag \triangleright x.\underline{a}.P \mid \lbag \circ \rbag\lbag\langle 1\rangle, x\rbag\langle 3\rangle \triangleright Q) \end{aligned}$$

so that in the end all memories are locked in $R_{\underline{a}}$ and $R_{\underline{b}}$, and R has been turned into a weak external sum ![6]

6 Conclusion

We have proposed a enrichment of CCS with memories which processes can use to backtrack. Memories are themselves distributed and the syntax stays close to the original concept of CCS. On the theoretical side, we have proved that backtracking is done in exact accordance with the true concurrency concept of causal equivalence. We also have shown how to integrate irreversible actions, and have given a procedure collecting obsolete memories.

There are many directions in which this study can be extended. First, one should rephrase this construction in terms of Winskel's event structures for CCS. RCCS seems to provide an internal language for reversible event structure and this should be made a precise statement. Second, one should seriously study the notion of bisimulation that is generated by the LTS and study process composition in RCCS. Third, preliminary investigations show that the whole strategy developed here can be imported in π-calculus. There are more things to be remembered and the amount of symbol-pushing needed for a comparable theory gets daunting. So, beginning within the simpler framework of CCS might have been good start. But π is a lot more expressive and this is a strong incentive to develop a notion of reversible π-calculus. Besides, one could think of bootstraping the system and encode reversible π into π.

Finally, the example of the internal sum given in the last section, is strongly suggesting that transactional mechanisms can be understood in terms of RCCS. One starts with a rough encoding of the external sum, which is the simplest

[6] By weak external sum, we mean that R has infinite traces where it constantly hesitates between its irreversible actions a and b.

possible transaction in some sense. And since RCCS provides a foolproof deadlock escape mechanism on reversible actions, carefully chosing them results in a correct encoding. We feel that one contribution of the current paper is to lay down the foundations to explore this matter further.

References

1. Robin Milner. *Communication and Concurrency.* International Series on Computer Science. Prentice Hall, 1989.
2. Jean-Jacques Lévy. Réductions optimales en λ-calcul. PhD, 1978.
3. Gérard Berry and Jean-Jacques Lévy. Minimal and optimal computation of recursive programs. *JACM*, 26:148–175, 1979.
4. Aviv Regev, William Silverman, and Ehud Shapiro. Representation and simulation of biochemical processes using the π-calculus process algebra. In R. B. Altman, A. K. Dunker, L. Hunter, and T. E. Klein, editors, *Pacific Symposium on Biocomputing*, volume 6, pages 459–470, Singapore, 2001. World Scientific Press.
5. Corrado Priami, Aviv Regev, Ehud Shapiro, and William Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 2001.
6. Vincent Danos and Jean Krivine. Formal molecular biology done in CCS. In *Proceedings of BIO-CONCUR'03, Marseille, France*, volume ? of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2003. To appear.
7. K.V.S Prasad. Combinators and bisimulation proofs for restartable systems. PhD, 1987.
8. Jan A. Bergstra, Alban Ponse, and Jos van Wamel. Process algebra with backtracking. In *REX School Symposium*, pages 46–91, 1993.
9. Gérard Boudol and Ilaria Castellani. Permutation of transitions : an event structure semantics for ccs. *Technical Report, INRIA*, 798, 1988.
10. Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. A partial ordering semantics for CCS. *Theoretical Computer Science*, 75:223–262, 1990.
11. Gérard Boudol, Ilaria Castellani, Matthew Hennesy, and Astrid Kiehn. Observing localities. In *Proceedings MFCS'91*, volume 114, pages 31–61, 1991.
12. Pierpaolo Degano and Corrado Priami. Proved trees. In *Automata, Languages and Programming*, volume 623 of *LNCS*, pages 629–640. Springer Verlag, 1992.
13. Gérard Boudol, Ilaria Castellani, Matthew Hennesy, and Astrid Kiehn. A theory of processes with localities. *Formal Aspect of Computing*, 1992.
14. Ilaria Castellani. Observing distribution in processes: static and dynamic localities. *INRIA, Research Report*, 2276, 1994.
15. Pierpaolo Degano and Corrado Priami. Non interleaving semantics for mobile processes. In *Automata, Languages and Programming*, volume 944 of *LNCS*, pages 660–667. Springer Verlag, 1995.
16. Michele Boreale and Davide Sangiorgi. A fully abstract semantics for causality in the π-calculus. *Acta Informatica*, 35, 1998.
17. E.N Elnozahy, D.B Johnson, and Y.M Wang. A survey of rollback recovery protocols in message passing systems. *CMU-CS-96-181*, 1996.

7 Supplementary Material

An equivalent and perhaps more intuitive way of presenting short computations is by using a graphical representation. We develop here a small example, starting with the process:

$$\begin{array}{ll} R & = \langle 1 \rangle \triangleright P_0 \mid \langle 2 \rangle \triangleright P'_0 \mid \langle 3 \rangle \triangleright P''_0 \\ P_0 & = \bar{x}.(z.0 \mid y.0) \\ P'_0 & = \bar{y}.\bar{t}.0 \\ P''_0 & = \bar{x}.t.\bar{z}.0 \end{array}$$

One gets the following computation consisting only of forward synchronizations (with the usual shorthand notations for identifiers):

$$\begin{array}{ll} R \xrightarrow{\langle 1 \rangle, \langle 3 \rangle : \tau} & \langle \langle 3 \rangle, \bar{x} \rangle \langle 1 \rangle \triangleright P_1 \mid \langle 2 \rangle \triangleright P'_0 \mid \langle \langle 1 \rangle, x \rangle \langle 3 \rangle \triangleright P''_1 \\ \xrightarrow{\langle 2 \rangle, \langle 1 \rangle \langle 1 \rangle : \tau} & \langle \langle 2 \rangle, y \rangle \langle 1 \rangle \langle \langle 3 \rangle, \bar{x} \rangle \langle 1 \rangle \triangleright Q_1 \mid \langle 2 \rangle \langle \langle 3 \rangle, \bar{x} \rangle \langle 1 \rangle \triangleright Q'_0 \mid \langle \langle 1 \rangle \langle 1 \rangle, \bar{y} \rangle \langle 2 \rangle \triangleright P'_1 \mid \langle \langle 1 \rangle, x \rangle \langle 3 \rangle \triangleright P''_1 \\ \xrightarrow{\langle 2 \rangle, \langle 3 \rangle : \tau} & \langle \langle 2 \rangle, y \rangle \langle 1 \rangle \langle \langle 3 \rangle, \bar{x} \rangle \langle 1 \rangle \triangleright Q_1 \mid \langle 2 \rangle \langle \langle 3 \rangle, \bar{x} \rangle \langle 1 \rangle \triangleright Q'_0 \\ & \mid \langle \langle 3 \rangle, \bar{t} \rangle \langle \langle 1 \rangle \langle 1 \rangle, \bar{y} \rangle \langle 2 \rangle \triangleright P'_2 \mid \langle \langle 2 \rangle, t \rangle \langle \langle 1 \rangle, x \rangle \langle 3 \rangle \triangleright P''_2 \\ \xrightarrow{\langle 21 \rangle, \langle 3 \rangle : \tau} & \langle \langle 2 \rangle, y \rangle \langle 1 \rangle \langle \langle 3 \rangle, \bar{x} \rangle \langle 1 \rangle \triangleright Q_1 \mid \langle \langle 3 \rangle, z \rangle \langle 2 \rangle \langle \langle 3 \rangle, \bar{x} \rangle \langle 1 \rangle \triangleright Q'_1 \\ & \mid \langle \langle 3 \rangle, \bar{t} \rangle \langle \langle 1 \rangle \langle 1 \rangle, \bar{y} \rangle \langle 2 \rangle \triangleright P'_2 \mid \langle \langle 21 \rangle, \bar{z} \rangle \langle \langle 2 \rangle, t \rangle \langle \langle 1 \rangle, x \rangle \langle 3 \rangle \triangleright P''_3 \end{array}$$

Each computation state can be mapped back to a unique *recovery line* (that terminology is taken from recovery protocols in distributed operating systems design [17]), that is to say one of the blue (or gray) lines in Fig. 1. Memories are seen to describe completely the events of each thread in inverse order of appearance.

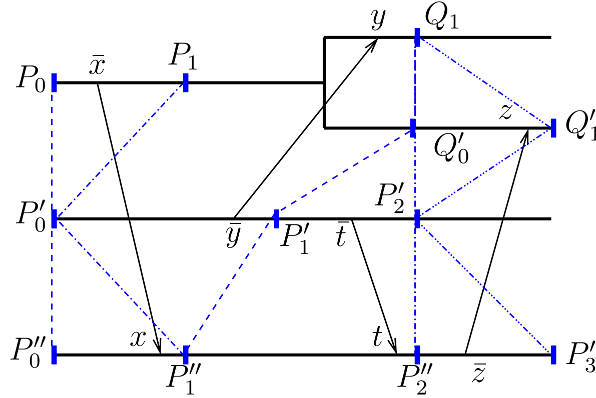

Fig. 1. Consistent recovery lines