Comparing Three Semantics for Linda-like Languages ⋆

Nadia Busi, Roberto Gorrieri and Gianluigi Zavattaro [1]

Dipartimento di Scienze dell'Informazione, Università di Bologna, Mura Anteo Zamboni 7, I-40127 Bologna, Italy

---

Abstract

A simple calculus based on generative communication is introduced; among its primitives, it contains a conditional input operation that tests for presence (or absence) of an output, reminiscent of the *inp* predicate of Linda. We study three different semantics for the output operation, called *instantaneous*, *ordered* and *unordered*, and we compare these approaches from two different points of view.

First, we investigate the associated behavioural semantics by characterizing the coarsest congruence contained in the barbed bisimulation. We obtain the following results: in the *instantaneous* case the coarsest congruence is a variant of asynchronous bisimulation while, for the *ordered* and *unordered* semantics, we obtain a small variant of the classic (synchronous) bisimulation. Moreover, the three obtained congruences are pairwise different.

Then, we compare the expressiveness of the three approaches. We first list a class of coordination primitives that are directly implementable in our calculus under the *instantaneous* semantics but not under the *ordered* one. Finally, we show that the calculus is Turing powerful under the *instantaneous* and *ordered* approaches, whereas this is not the case for the *unordered* semantics. Thus, we conclude that there exists a strict expressiveness hierarchy among the three semantics.

Keywords: Coordination languages, Semantics of Linda, Behavioural equivalences, Expressiveness of concurrent languages.

---

1 Introduction

Generative communication, realized by means of the insertion and withdrawal of elements from a shared multiset, is the peculiar feature of a family of co-

⋆ Work supported by Esprit Working Group Coordina n. 24512. Extended and revised version of [BGZ97b].

[1] Correspoding author. E-mail: zavattar@cs.unibo.it.

ordination languages [GC92], of which Linda [Gel85] is the most prominent representative. Recently, this communication mechanism has been adopted also by several proposals of coordination platforms for the Java programming language, such as the Sun JavaSpaces [JS98] or the IBM T Spaces [WLLF98]. Generative communication is based on the following principles: a sender communicates with a receiver through a shared data space (called *tuple space*, TS for short), where emitted messages are collected; the receiver can consume the message from TS; a message generated by a process has an independent existence in the tuple space until it is explicitly withdrawn by a receiver; in fact, after its insertion in TS, a message becomes equally accessible to all processes, but it is bound to none. Hence, the communication is asynchronous because the sender may proceed just after performing the emission of a message to the TS. Similarly, the receiver can input a message present in TS at any time: a hand-shake synchronization between TS and the receiver completes the communication between the sender and the receiver, with the side-effect of removing the message from TS.

Besides the non-blocking output operation $out(a)$ (that sends message a to the tuple space) and the blocking input operation $in(a)$ (that removes message a from TS), Linda also offers a conditional input predicate, called $inp(a)$, that checks the current status of TS; if the required message a is absent, the value $false$ is returned; on the other hand, if the message is found, its behavior is the same as the in operation and the value $true$ is returned. We represent this predicate by means of an *if–then–else* construct $inp(a)?P\_Q$; it directs the flow of control to P or to Q, depending on the presence or absence of message a in TS, respectively.

The paper presents an investigation of possible semantics for generative communication in a process algebraic setting, with particular care to the output operation that, in our opinion, has not yet received enough attention.

Conceptually, the execution of the Linda-like output primitive $out(a)$ can be seen as composed of two phases: the *emission* of the message a (sending a to the TS) and the *rendering* of a (actual presence of a in the TS, we denote with $\langle a\rangle$). The three semantics we are going to investigate are inspired by previous related proposals (e.g., of the asynchronous object calculus of [HT91]), as well as by the informal semantics of Linda reported in the reference manual [SCA95]. The three different semantics may be summarized as follows:

- *Instantaneous*: With $out(a)$ we mean that the message is already in the TS. Hence, $out(a).P = \langle a\rangle|P$, where $|$ is the parallel composition operator. For instance, consider a process P that wants to input a and a process $out(a)$; if composed in parallel, P can immediately input message a. This approach has been adopted in the asynchronous π-calculus [HT91,Bou92]; it is obtained by means of a simple syntactic restriction to that language: outputs cannot be used as prefixes. Intuitively, this semantics is a bit strange, as the execution complexity of certain actions depends on their syntactic continuation. E.g., consider $P = \eta.out(a_1)$ and $Q = \eta.(out(a_1)|out(a_2)|\ldots|out(a_n))$, where η is any non output prefix; in *one* single *atomic* step, P executes action η and puts one tuple, $\langle a_1\rangle$, in the TS, while Q executes action η and puts the n messages $\langle a_1\rangle, \ldots, \langle a_n\rangle$ in the TS.

- *Ordered*: The emission and the rendering of one message form together one single autonomous atomic action: $out(a).P$ becomes in one (internal) step the agent $\langle a\rangle|P$. In this way, the order of emission is respected by the rendering order. The implementation of $out(a)$ is simple: the sender sends the message a and then waits for the acknowledgement from the TS; hence, the emission of a message is realized by means of a synchronous hand–shake communication between the sender and the TS. This approach has already received an operational treatment in [HKH95,CGZ96,BGZ98].
- *Unordered*: The emission and the rendering of one message are distinct autonomous actions. Hence, $out(a).P$ emits message a becoming the agent $\langle\langle a\rangle\rangle|P$ in one (internal) step, where P is free to proceed, but message a is not yet present in the TS; indeed, $\langle\langle a\rangle\rangle$ takes one further internal step to become $\langle a\rangle$. The implementation of the $out(a)$ operation is trivial: the process $out(a).P$ sends the message a to the TS, and proceeds without waiting for the message to reach the TS; hence, the emission is realized by means of an asynchronous communication between the sender and the TS. Thus the order of emission may not to be respected by the rendering order: for instance, if a process executes the sequence $out(a).out(b)$, then a may be rendered before or after the emission of b, or even after the rendering of b. To the best of our knowledge, we do not know of any paper in process algebra studying the semantics of this approach.

As the above approaches are equally interesting, we think it is worthwhile to compare them formally.

The aim of the paper is twofold. On the one hand we compare the three interpretations for the output operator with respect to the behavioural semantics; on the other hand, we analyse their relative expressive power. The final result is that the three interpretations are basically different from the point of view of both the behavioural semantics and the expressive power.

In detail, we investigate the behavioural semantics by following a commonly used approach: we characterize the coarsest congruence contained in the barbed bisimulation [MS92], a rather coarse equivalence that equates processes that are bisimilar on reduction steps and offer, at any pair of related states, the same observable actions. We prove that a variant of the asynchronous bisimulation [ACS98] is the right semantics for the instantaneous semantics, while the correct semantics for the other two cases is a variant of the classic (synchronous) bisimulation [Mil89], where inputs and outputs are treated symmetrically. The resulting three congruences are not only pairwise different, but also none of them is included in any one of the others.

Regarding the expressive power, we show that there is a precise hierarchy among the three variants: the calculus under the instantaneous semantics is more expressive than the calculus under the ordered semantics, which in turn is more expressive than the calculus under the unordered semantics. The first separation result is achieved by showing a series of constructs that can be directly implemented in our process algebra only under the instantaneous semantics. Namely, we discuss the possibility of implementing the Linda *rd* and *rdp* operators (the non-consuming counterparts of *in* and *inp* respectively) and a *test-and-set* operator. The second separation result is even more basic. We show that, for the instantaneous and ordered semantics, it is possible to encode any Random Access Machine (RAM) [SS63], a Turing equivalent formalism. On the other hand, it is possible to prove that our calculus is not Turing powerful under the unordered semantics; here we simply sketch the proof idea, based on a Petri net semantics for the calculus; the interested reader can consult [BGZ99].

The paper is structured as follows. In Section 2 we introduce the syntax of our process algebra and the three operational semantics for the *out* primitive, comparing their differences with one instructive example. Section 3 studies the behavioural semantics, while Section 4 discusses the expressiveness of the calculus. Section 5 reports some conclusive remarks and the Appendix collects the proofs of the Theorems in Section 3.

2 The Language and its Operational Semantics

Let $Mess$, ranged over by a, b, ..., be a denumerable set of message names, and let Var, ranged over by X, Y, ..., be the set of program variables. We define agents, denoted by P, Q, ..., the terms obtained by the following grammar:

$$P ::= \langle a \rangle \;\mid\; C \;\mid\; P|P \;\mid\; P\backslash a$$

$$C ::= \mathbf{0} \;\mid\; out(a).C \;\mid\; in(a).C \;\mid\; inp(a)?C\_C \;\mid\; C|C \;\mid\; X \;\mid\; rec X.C$$

Agents consist of the parallel composition of the messages already in the TS (each one denoted by an agent $\langle a \rangle$) and the concurrent programs denoted by C, D, ..., sharing the tuples. We use also a restriction operator $P\backslash a$ in order to have the possibility of defining the scope of message names. A program C can be a terminated program $\mathbf{0}$ (which is usually omitted for the sake of simplicity), a program starting with a coordination primitive (*in*, *out*, and *inp*), or the parallel composition of two programs.

(i)	$P\|Q \equiv Q\|P$		
(ii)	$(P\|Q)\|R \equiv P\|(Q\|R)$		
(iii)	$P\|\mathbf{0} \equiv P$		
(iv)	$\mathbf{0}\backslash a \equiv \mathbf{0}$		
(v)	$(P\backslash a)\backslash b \equiv (P\backslash b)\backslash a$		
(vi)	$(P\|Q)\backslash a \equiv P\|(Q\backslash a)$	$a \notin fn(P)$	
(vii)	$P\backslash a \equiv P[b/a]\backslash b$	b fresh	
$(viii)$	$rec\,X.P \equiv P[rec\,X.P/X]$		

Table 1
Structural congruence.

The coordination primitives $out(a)$ and $in(a)$ can be represented as usual prefixes, while $inp(a)$ requires a sort of *if-then-else* construct. In fact, $inp(a)?C\_D$ is a program which requires the message a to be consumed; if a is present, it is removed and the program C is executed, otherwise D is chosen. Recursive agents are defined by using agent variables and the standard operator for recursion $rec\,X.C$. As usual, we restrict to closed terms and guarded recursion [Mil89]. In the following $Agent$ denotes the set containing all possible agents.

The set of free names in P, denoted by $fn(P)$, is defined as follows:

$$
\begin{aligned}
&fn(\mathbf{0}) = fn(X) = \emptyset\\
&fn(P|Q) = fn(P) \cup fn(Q)\\
&fn(\langle a \rangle) = \{a\}\\
&fn(P\backslash a) = fn(P) \setminus \{a\}\\
&fn(in(a).P) = fn(out(a).P) = \{a\} \cup fn(P)\\
&fn(inp(a)?P\_Q) = \{a\} \cup fn(P) \cup fn(Q)\\
&fn(rec\,X.P) = fn(P)
\end{aligned}
$$

We present three different operational semantics for our language, one for each kind of output prefix sketched in the Introduction. The semantics are presented in two steps. First, we define *structural congruences* over agents; this relation captures the fact that, for example, the order of the terms in a parallel composition has no effects on its behaviour. Next, we define *labeled transition*

$$
\begin{array}{llll}
(1) & \langle a \rangle \xrightarrow{\overline{a}} \mathbf{0} & (2) & in(a).P \xrightarrow{a} P \\[2ex]
(3) & inp(a)?P\_Q \xrightarrow{a} P & (4) & inp(a)?P\_Q \xrightarrow{\neg a} Q \\[2ex]
(5) & \dfrac{P \xrightarrow{\alpha} P'}{P \backslash a \xrightarrow{\alpha} P' \backslash a} \quad \alpha \neq a, \overline{a}, \neg a & (6) & \dfrac{P \xrightarrow{\neg a} P'}{P \backslash a \xrightarrow{\tau} P' \backslash a} \\[3ex]
(7) & \dfrac{P \xrightarrow{\neg a} P' \quad Q \stackrel{\overline{a}}{\not\longrightarrow}}{P|Q \xrightarrow{\neg a} P'|Q} & (8) & \dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \\[3ex]
(9) & \dfrac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad \alpha \neq \neg a & (10) & \dfrac{P \equiv Q \quad Q \xrightarrow{\alpha} Q' \quad P' \equiv Q'}{P \xrightarrow{\alpha} P'}
\end{array}
$$

Table 2
Operational semantics.

systems specifying how agents evolve by means of the actions performed by some program in it.

The structural congruence for the *instantaneous* semantics $\equiv_i$ is the smallest congruence satisfying the rules (i), ..., $(viii)$ of Table 1 and (ix) of Table 3. The structural congruences for the *ordered* and *unordered* semantics denoted with $\equiv_o$ and $\equiv_u$ are instead defined both as the smallest congruence satisfying only (i), ..., $(viii)$.

The labelled transition systems are of the kind ($Agent$, $Label$, $\longrightarrow$) where $Label = \{\tau\} \cup \{a, \overline{a}, \neg a \mid a \in Mess\}$ (ranged over by α, β, ...) is the set of the possible labels, where τ represents internal action, while a, $\overline{a}$, and $\neg a$ stand for input, output and test for absence operations, respectively. The labelled transition relation $\longrightarrow_i$ for the *instantaneous* semantics is the smallest one satisfying the axioms and rules from (1) to (10) in Table 2; $\longrightarrow_o$ for the *ordered* semantics is the one satisfying (1)...(10) and the axiom (11) of Table 3; while $\longrightarrow_u$ for the *unordered* semantics considers the axioms and rules (1)...(10), (12) and (13). The indexes i, o and u, distinguishing the three structural congruences and transition relations, are omitted when they are clear from the context.

Axiom (1) shows that the tuple $\langle a \rangle$ is able to give its contents to the environment, by performing an action labeled with $\overline{a}$. Axioms (2) and (3) define the possible input actions on the message a (action labeled with a), according to the execution of an *in* or a successful *inp* operation, respectively. If a process executing an *inp* does not find the required message a, it can guess its absence by performing an action labeled with $\neg a$ (axiom (4)). Rule (5) states that no actions containing the name a can be performed by the agent $P \backslash a$. When an agent P willing to perform a $\neg a$ action is restricted on the name a, its $\neg a$ operation becomes a local step of computation (i.e. labeled with τ) because

Instantaneous:	(ix)	$out(a).P \equiv \langle a \rangle \vert P$
Ordered:	(11)	$out(a).P \xrightarrow{\tau} \langle a \rangle \vert P$
Unordered:	(12)	$out(a).P \xrightarrow{\tau} \langle\langle a \rangle\rangle \vert P$
	(13)	$\langle\langle a \rangle\rangle \xrightarrow{\tau} \langle a \rangle$

Table 3
Three semantics for the *out* primitive.

no further agents can offer message a; in other words, the search for a has finished because it has become a local name (rule (6)). On the other hand, if P is composed in parallel with another agent Q, the executability of $\neg a$ by $P|Q$ depends on the inability of Q to offer message a. Otherwise, the guess of P is wrong and $\neg a$ cannot be executed (rule (7)). The other rules are the usual for synchronization between complementary actions (8), for local actions in parallel composed agents (9), and for the possibility of executing the same actions for structurally congruent agents (10). There are no rules for recursion because its semantics is defined by the congruence rule $(viii)$ which applies one unfolding step to a recursively defined program.

Rule (7) uses a negative premise; it is easy to see that our transition system specification is strictly stratifiable [Gro93], thus there exists a unique transition system agreeing with it.

The rules which differentiate the three semantics are presented in Table 3. Following the *instantaneous* approach messages have to be considered already available at the moment an output operation has to be performed. This is obtained by introducing a further rule for the structural congruence stating that a program starting with the prefix $out(a)$ is the same as putting the tuple $\langle a \rangle$ in parallel with the continuation of the program. As rule (ix) may relate guarded terms to unguarded ones (e.g., $rec\, X.out(a).X \equiv rec\, X.(\langle a \rangle | X)$) we will consider as guarded only terms in which each program variable occurs inside an *in* prefix or an *inp* construct.

In the *ordered* approach the output operation consists of one local non-blocking action labeled with τ which creates the tuple $\langle a \rangle$. In this way, when a sequence of output is executed, the messages are rendered in the same order they are emitted.

In the *unordered* approach, the execution of an output operation emitting the message a does not directly generate the corresponding tuple $\langle a \rangle$, but it creates an agent which will make message a available only after a non predictable delay. This agent, denoted by $\langle\langle a \rangle\rangle$,[2] is only able to perform an internal action labelled with τ becoming $\langle a \rangle$.

Example 2.1 An example, inspired by [SCA95], allows us to show the differences among the three semantics. Consider P and Q below where the only difference between them is the order of emission of the messages a and b:

$$P \stackrel{def}{=} (out(a).out(b) \mid in(a).inp(b)?C\_D)\backslash a\backslash b$$
$$Q \stackrel{def}{=} (out(b).out(a) \mid in(a).inp(b)?C\_D)\backslash a\backslash b$$

Observe that a and b are restricted names; this ensures that the in and inp operations on these names are executed locally.

In the *instantaneous* semantics the messages a and b becomes available in the same instant, hence when the testing process consumes a and executes the $inp(b)$ primitive the required message is found and consumed. Hence, the inp continuation is C for both P and Q.

Under the *ordered* semantics the messages a and b become available in the same order they are emitted. In this case the test performed by the inp operation in P and Q gives rise to two different results. The presence of the message b is ensured only in Q, where b becomes available before a; hence the continuation of the inp operation is D. Instead, in P the presence or the absence of the tuple b at the instant the inp is executed, depends on the order of execution of the operations: if the inp primitive is performed before the $out(b)$ operation, then the message b is not found (the continuation is D), otherwise it is found and consumed (the continuation is C).

The *unordered* semantics shows a third kind of behaviour because the messages a and b become available in an unpredictable order, hence the search performed by the inp operation can give rise to a success or a failure in both P and Q.

The behaviours of the agents P and Q under the three different semantics are summarized by showing the possible continuations of the inp operator:

	Instantaneous	Ordered	Unordered
P	C	C or D	C or D
Q	C	C	C or D

[2] The syntax for the agents P is extended in the case of unordered output semantics by allowing P to be also the agent $\langle\langle a \rangle\rangle$; the function returning the free names is also extended adding $fn(\langle\langle a \rangle\rangle) = \{a\}$.

3 Behavioural Semantics

In this section the problem of defining observational semantics for the three different operational semantics is considered.

We first show, by means of the following example, that the standard notion of bisimulation [Mil89], is not satisfactory for our language.

Example 3.1 In this example the term $rec\,X.\tau.X$ is an agent whose observable behaviour consists of an infinite sequence of τ steps. Even if this term is not part of the syntax of our language, it can be easily encoded; e.g., the agent $(rec\,X.out(b).in(b).X)\backslash b$ is a possible solution. Consider now the agents:

$$P \stackrel{def}{=} rec\,Y.inp(a)?Y\_Y|rec\,X.\tau.X \qquad Q \stackrel{def}{=} rec\,Y.in(a).Y|rec\,X.\tau.X$$

Agent P recursively performs three possible transitions labelled with a, $\neg a$, or τ respectively; on the other hand, agent Q only has outgoing transitions labelled with a or τ.
It is easy to see that the standard bisimulation distinguishes the two terms, indeed, agent P has a derivation labelled with $\neg a$ that is not allowed by Q. This extra transition of P does not add further possible behaviours to the agent. In particular, it is not difficult to see that P and Q present the same behaviour in all possible contexts; in other words, they cannot be distinguished by any external observer. Indeed, the extra $\neg a$ transition of P can be performed only if no $\langle a \rangle$ is available in the environment and the environment is left unchanged by the execution of this step. Agent Q is able to mimic the same behaviour by performing its τ labelled transition. $\square$

Example 3.2 The above example shows that a standard bisimulation that treats $\neg a$ as a standard label is too strong for our calculus. In order to overcome this limitation, we could consider a modified bisimulation that treats the label $\neg a$ as a τ. This approach could be justified also by the fact that the label $\neg a$ has been introduced only for helping an SOS formulation of the semantics, while it is conceptually an internal step (that can be performed only in particular contexts). A bisimulation of this kind equates the above agents P and Q, but is not satisfactory in general. For example, consider the following terms:

$$R \stackrel{def}{=} in(a)|\tau.out(b) \qquad S \stackrel{def}{=} inp(a)?(\tau.out(b))\_(in(a)|out(b))$$

where $\tau.out(b)$ represents an output operation following an internal step (this happens, e.g., in the term $(\langle b \rangle|in(b).out(a))\backslash b$). The agents R and S are equated by a bisimulation that does not make any distinction between the labels τ and $\neg a$. Nevertheless, these agents are distinguished by the term

$T \stackrel{def}{=} \langle a\rangle | in(b).in(a).out(c)$, as $R|T$ could produce the message $\langle c\rangle$, while $S|T$ cannot. This happens because S does not produce $\langle b\rangle$, at least as soon as $\langle a\rangle$ is available. $\Box$

The above examples shows the need to investigate a new notion of bisimuation, more abstract than a standard bisimulation but also more concrete than a bisimulation that does not distinguish the labels τ and $\neg a$.

The idea is to follow a commonly used approach which consists of investigating the coarsest congruence contained in the *barbed bisimulation* [MS92], a rather coarse equivalence that equates processes bisimilar on reduction steps that offer, at any pair of related states, the same observable actions. In [MS92] it is proved that, for CCS [Mil89], the obtained equivalence corresponds to the classical notion of bisimulation. Instead, we will show that in our setting the coarsest congruence contained in the barbed bisimulation is more abstract than bisimulation (e.g. it will equate the agents P and Q of Example 3.1).

In order to define barbed bisimulation we have to introduce the notion of *reduction* and *commitments*.

In our language, we consider as reductions not only the usual derivations labeled with τ, but also those labeled with $\neg a$. In fact, a derivation $P \xrightarrow{\neg a} P'$ indicates that P can become P' if no tuples $\langle a\rangle$ are available in the external environment. Hence, if P is stand–alone (i.e. without external environment), it can be considered able to become P'. Formally:

$P \longrightarrow P'$ iff $P \xrightarrow{\tau} P'$ *or* $P \xrightarrow{\neg a} P'$ for some a

We consider also a weak notion of reduction $P \Longrightarrow P'$, that abstracts away from the number of derivations needed by P to become P':

$P \Longrightarrow P'$ iff $P \longrightarrow^* P'$

More attention must be paid in order to identify what is observable or not. In Linda-like languages, based on the notion of uncoupled interaction via a shared data space (the TS), it is natural to consider the TS as the observable part of a system. In other words, an external observer is not allowed to directly interact with the processes, but it can only communicate with them by introducing, consuming, or testing the actual state of the TS. We model this by permitting to observe only the presence of a certain kind of tuple $\langle a\rangle$ corresponding to the ability of performing a transition labelled with $\overline{a}$:

$P \downarrow \overline{a}$ iff $P \xrightarrow{\overline{a}} P'$ for some P'

It is interesting to observe that this notion of commitment is essentially the same as the one defined in [ACS98] in the setting of asynchronous π-calculus, where channel based communication is considered instead of generative communication via a shared data space.

We also define a *weak* commitment $\Downarrow \overline{a}$ in order to be able to denote the possibility of a certain commitment after some reduction steps:

$P \Downarrow \overline{a}$ iff $P \Longrightarrow P'$ *and* $P' \downarrow \overline{a}$ for some P'

The resulting definition of barbed bisimulation is the following:

Definition 3.3 A binary, symmetric relation $\mathcal{R}$ on *Agent* is a *barbed bisimulation* if $(P, Q) \in \mathcal{R}$ implies:

- if $P \longrightarrow P'$ then there exists Q' such that $Q \longrightarrow Q'$ and $(P', Q') \in \mathcal{R}$;
- if $P \downarrow \overline{a}$ then $Q \downarrow \overline{a}$.

Two agents P and Q are *barbed bisimilar*, written $P \stackrel{\bullet}{\sim} Q$, if there exists a barbed bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. $\Box$

As already stated, we investigate the coarsest congruence contained in the barbed bisimulation for the three semantics.

3.1 The Ordered Case

In this part of the paper we take into account only the ordered semantics, and we prove that in this case the coarsest congruence contained in $\stackrel{\bullet}{\sim}$ is the following $\neg$-bisimulation. [3]

Definition 3.4 A binary, symmetric relation $\mathcal{R}$ on *Agent* is a $\neg$-*bisimulation* if $(P, Q) \in \mathcal{R}$ implies:

- if $P \stackrel{\alpha}{\longrightarrow} P'$, with $\alpha \neq \neg a$, then there exists Q' such that $Q \stackrel{\alpha}{\longrightarrow} Q'$ and $(P', Q') \in \mathcal{R}$;
- if $P \stackrel{\neg a}{\longrightarrow} P'$ then there exists Q' such that
  · either $Q \stackrel{\neg a}{\longrightarrow} Q'$ and $(P', Q') \in \mathcal{R}$
  · or $Q \stackrel{\tau}{\longrightarrow} Q'$ and $(P', Q') \in \mathcal{R}$.

Two agents P and Q are $\neg$-*bisimilar*, written $P \sim Q$, if there exists a $\neg$-bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. $\Box$

The coarsest congruence contained in the barbed bisimulation for the asynchronous π-calculus is the asynchronous bisimulation [ACS98]. This bisimulation allows an input action to be matched also by an internal τ step, while in our $\neg$-bisimulation this is not true. This difference is due to the fact that in the asynchronous π-calculus, a process can receive a message and then

[3] A similar proof already appeared in [BGZ98]; we recall it here because it will be used in some of the consequent proofs.

immediately emit it once again. This allows a process to simulate an input action (followed by the instantaneous emission of the consumed message) with an internal τ action. This cannot happen under the ordered output because the instantaneous emission is not allowed. This is formalized by the following proposition.

Proposition 3.5 Let P be an agent such that $P \not\downarrow \overline{a}$. If $P \xrightarrow{b} P'$ then also $P' \not\downarrow \overline{a}$.

Proof By induction on the proof of the transition $P \xrightarrow{b} P'$. $\square$

In the following we reason up to the structural congruence $\equiv$; moreover, $\prod_n Q$ is used as a shorthand for $\mathbf{0}$, if $n = 0$, or for n copies of the agent Q composed in parallel, while $\prod_{l \in L} P_l$ stands for $\mathbf{0}$ if $L = \emptyset$, or for $P_{a_1}|\ldots|P_{a_n}$ if $L = \{a_1, \ldots, a_n\}$.

We need also the following two propositions. The former indicates that an agent having a derivation labelled with $\neg a$ cannot perform a step labeled with $\overline{a}$, i.e., it contains no tuple $\langle a \rangle$. The latter shows that if an agent is able to perform consecutively n steps labeled with $\overline{a}$, then at least n occurrences of the tuple $\langle a \rangle$ are contained in it.

Proposition 3.6 Given the agent P, if $P \xrightarrow{\neg a} P'$ then $P \not\downarrow \overline{a}$.

Proof By induction on the proof of the transition $P \xrightarrow{\neg a} P'$. $\square$

Proposition 3.7 Let P be an agent. Given $n \geq 0$, if $P \xrightarrow{\overline{a}} P_1 \xrightarrow{\overline{a}} \ldots \xrightarrow{\overline{a}} P_n$ then $P \equiv P_n | \prod_n \langle a \rangle$.

Proof The proof uses double induction; first on the number n of successive derivations labeled with $\overline{a}$, then we proceed by induction on the proof of the n-th derivation $P_{n-1} \xrightarrow{\overline{a}} P_n$. $\square$

In order to prove that the $\neg$-bisimulation is the coarsest congruence contained in the barbed bisimulation for the unordered case, we first assert that $\sim$ is a congruence and then we prove that if two agents are barbed bisimilar under every context, they must also be $\neg$-bisimilar.

Proposition 3.8 $\neg$-bisimulation is a congruence. $\square$

The proof of the congruence result is omitted here as it is standard [Mil89].

Theorem 3.9 Let P and Q be agents. If $P|R \stackrel{\bullet}{\sim} Q|R$ for every agent R, then $P \sim Q$.

Proof The complete proof can be found in the appendix; here, we sketch its

structure.
Let P and Q be two agents satisfying the premises of the theorem. Let $L = fn(P) \cup fn(Q)$; observe that L is finite. We show that the pair (P, Q) is contained in a $\neg$-bisimulation (up to $\equiv$), hence $P \sim Q$. In particular, we define an agent R such that the relation:

$$\mathcal{R} = \{(S, T) \mid S|R \stackrel{\bullet}{\sim} T|R \textit{ and } fn(S), fn(T) \subseteq L\}$$

is a $\neg$-bisimulation (up to $\equiv$). The pair (P, Q) is in $\mathcal{R}$ because $P|R$ is barbed bisimilar to $Q|R$ and both $fn(P)$ and $fn(Q)$ are subsets of L. $\Box$

Corollary 3.10 $\neg$-bisimulation is the coarsest congruence contained in the barbed bisimulation for the *ordered* semantics.

Proof Let $\asymp$ be a congruence contained in $\stackrel{\bullet}{\sim}$. We show that $\asymp \subseteq \sim$. In fact, if $P \asymp Q$ then $P|R \asymp Q|R$ for every agent R because $\asymp$ is a congruence. By $\asymp \subseteq \stackrel{\bullet}{\sim}$ it follows that $P|R \stackrel{\bullet}{\sim} Q|R$. By Theorem 3.9 also $P \sim Q$ holds. $\Box$

3.2 The Unordered Case

Also for the *unordered* semantics $\neg$-bisimulation is the coarsest congruence contained in the barbed bisimulation.

It easy to see that the propositions from 3.5 to 3.8 hold also under the unordered semantics. In this case also other facts are required.

Fact 3.11 Let P, P', and P'' be three agents such that $P \longrightarrow P' \longrightarrow P''$. If both $P \not\downarrow \overline{a}$ and $P'' \not\downarrow \overline{a}$ while $P' \downarrow \overline{a}$, then $P \stackrel{a}{\longrightarrow} P''|\langle\langle a\rangle\rangle$. $\Box$

Fact 3.12 Let P be an agent such that $P \not\downarrow \overline{a}$. If

$$P|\prod_n \langle\langle a\rangle\rangle \underbrace{\longrightarrow \ldots \longrightarrow}_{n \text{ times}} P'|\prod_n \langle a\rangle$$

then $P \equiv P'$. $\Box$

We can now present the new version of the Theorem 3.9 adapted to the unordered semantics.

Theorem 3.13 Let P and Q be agents. If $P|R \stackrel{\bullet}{\sim} Q|R$ for every agent R, then $P \sim Q$.

Proof The proof is reported in the appendix and follows the structure of the proof of Theorem 3.9. $\Box$

Corollary 3.14 $\neg$-bisimulation is the coarsest congruence contained in the barbed bisimulation for the *unordered* semantics.

Proof As the proof of Corollary 3.10, where Theorem 3.13 is used instead of Theorem 3.9. $\Box$

3.3 The Instantaneous Case

For the *instantaneous* semantics the coarsest congruence contained in the barbed bisimulation is the following *asynchronous* $\neg$*-bisimulation*.

Definition 3.15 A binary, symmetric relation $\mathcal{R}$ on *Agent* is an *asynchronous* $\neg$*-bisimulation* if $(P,Q) \in \mathcal{R}$ implies:

- if $P \xrightarrow{\overline{a}} P'$ then there exists Q' such that $Q \xrightarrow{\overline{a}} Q'$ and $(P',Q') \in \mathcal{R}$;
- if $P \xrightarrow{\tau} P'$ then there exists Q' such that
  - · either $Q \xrightarrow{\tau} Q'$ and $(P',Q') \in \mathcal{R}$
  - · or there exist b and Q'' such that $Q \xrightarrow{b} Q'$, $(P'|\langle b\rangle, Q') \in \mathcal{R}$, $Q \xrightarrow{\neg b} Q''$ and $(P',Q'') \in \mathcal{R}$;
- if $P \xrightarrow{\neg a} P'$ then there exists Q' such that
  - · either $Q \xrightarrow{\neg a} Q'$ and $(P',Q') \in \mathcal{R}$
  - · or $Q \xrightarrow{\tau} Q'$ and $(P',Q') \in \mathcal{R}$
  - · or there exist b and Q'' such that $Q \xrightarrow{b} Q'$, $(P'|\langle b\rangle, Q') \in \mathcal{R}$, $Q \xrightarrow{\neg b} Q''$ and $(P',Q'') \in \mathcal{R}$;
- if $P \xrightarrow{a} P'$ then there exists Q' such that
  - · either $Q \xrightarrow{a} Q'$ and $(P',Q') \in \mathcal{R}$
  - · or $Q \xrightarrow{\tau} Q'$ and $(P',Q'|\langle a\rangle) \in \mathcal{R}$
  - · or there exist b and Q'' such that $Q \xrightarrow{b} Q'$, $(P'|\langle b\rangle, Q'|\langle a\rangle) \in \mathcal{R}$, $Q \xrightarrow{\neg b} Q''$ and $(P',Q''|\langle a\rangle) \in \mathcal{R}$.

Two agents P and Q are *asynchronous* $\neg$*-bisimilar*, written $P \sim_a Q$, if there exists an asynchronous $\neg$-bisimulation $\mathcal{R}$ such that $(P,Q) \in \mathcal{R}$. $\Box$

As already stated, in the asynchronous bisimulation of [ACS98] an input step can be matched also by a τ labelled transition, provided that a new tuple is emitted immediately after the consumption step. Similarly, under the *instantaneous* semantics a new tuple can be rendered immediately after a consumption step; hence the asynchronous $\neg$-bisimulation allows the same matching between input operations and internal τ steps of the asynchronous bisimulation. This is the reason why we have called this equivalence asynchronous.

All the other new matchings allowed by the asynchronous $\neg$-bisimulation can be understood considering P and Q as below, where a is a name not appearing in R:

$P \stackrel{def}{=} inp(b)?out(b).R\_R$

$Q \stackrel{def}{=} (\langle a\rangle | in(a).R)\backslash a$

The agents P and Q cannot be distinguished under the *instantaneous* semantics because they both perform an internal step of computation having no influence on the environment, and then become R. In order to equate the agents above, the asynchronous $\neg$-bisimulation must introduce new matchings. For example a τ labelled step can be matched by a transition labeled with b. This is possible if the tuple $\langle b\rangle$ is generated immediately after having consumed it, and if an equivalent continuation can be chosen also if the message b is not present in the environment.

In order to prove that the asynchronous $\neg$-bisimulation is the coarsest congruence contained in the barbed bisimulation for the instantaneous case, we proceed as for the ordered and unordered semantics. We first assert that $\sim_a$ is a congruence and then we prove that if two agents are barbed bisimilar under every context, they must also be asynchronous $\neg$-bisimilar.

Proposition 3.16 Asynchronous $\neg$-bisimulation is a congruence. $\Box$

Theorem 3.17 Let P and Q be agents. If $P|R \stackrel{\bullet}{\sim} Q|R$ for every agent R, then $P \sim_a Q$.

Proof The proof is reported in the appendix and follows the structure of the proof of Theorems 3.9. $\Box$

Corollary 3.18 Asynchronous $\neg$-bisimulation is the coarsest congruence contained in the barbed bisimulation for the *instantaneous* semantics.

Proof As the proof of Corollary 3.10 where Theorem 3.17 is used instead of Theorem 3.9. $\Box$

It is interesting to note that neither the *inp* construct nor the restriction operator are used in the definition of the context R in any of the proof of coarsest congruence; this allows us to conclude that the results we have obtained are valid also if we eliminate these operators from the language. Moreover, if we drop the *inp* construct from the syntax, it is immediately clear that the label $\neg a$ becomes meaningless; thus the $\neg$-*bisimulation* and the *asynchronous* $\neg$-*bisimulation* collapse to the standard [Mil89] and the asynchronous bisimulation [ACS98], respectively. We can conclude that if *inp* is removed from the language, the coarsest congruence contained in the barbed bisimulation is the standard bisimulation, if the intended semantics is *ordered* or *unordered*, or the asynchronous bisimulation, in the case of *instantaneous* interpretation.

The equivalence $\sim_a$ on the *instantaneous* semantics and $\sim$ on the *ordered* and *unordered* semantics, infer three different equivalences on the language:

$P \sim_1 Q$ iff $P \sim_a Q$ in the *instantaneous* semantics

$P \sim_2 Q$ iff $P \sim Q$ in the *ordered* semantics

$P \sim_3 Q$ iff $P \sim Q$ in the *unordered* semantics

We show by examples that the three congruences are all different and no one of them is included in any one of the others.

Example 3.19 Consider the following agents:

$P \stackrel{def}{=} (out(a).out(b) \mid in(b).inp(a)?out(c)\_out(d))\backslash a\backslash b$

$Q \stackrel{def}{=} (out(b).out(a) \mid in(b).inp(a)?out(c)\_out(d))\backslash a\backslash b$

$R \stackrel{def}{=} (out(a).out(b) \mid in(b).in(a).out(c))\backslash a\backslash b$

Under the *instantaneous* semantics the three agents are equivalent; $P \sim_1 Q \sim_1 R$ because the unique behaviour the three agents can have is the one in which both the messages a and b are consumed and the new message c is emitted. In the case of *ordered* semantics only P and R are equivalent; $Q \not\sim_2 P \sim_2 R \not\sim_2 Q$ because Q can generate the message d (i.e. $Q \Downarrow \overline{d}$) while P and R can only emit c (i.e. $P \not\Downarrow \overline{d}, R \not\Downarrow \overline{d}$). In the *unordered* semantics only P and Q are equivalent; $R \not\sim_3 P \sim_3 Q \not\sim_3 R$ because P and Q can generate the message d (i.e. $P \Downarrow \overline{d}, Q \Downarrow \overline{d}$) while R can only emit c (i.e. $R \not\Downarrow \overline{d}$). $\square$

This example shows that $\sim_1 \not\subseteq \sim_2$, $\sim_1 \not\subseteq \sim_3$, and $\sim_2 \not\subseteq \sim_3$. It is possible to prove that also the inverse inclusions are false, as shown by the agents reported in the following example.

Example 3.20 For the sake of readability, in this example we will use a τ prefix and an internal choice $P \oplus Q$ operator; even if they are not part of the syntax of our language they can be encoded. In particular, given a program P, the term $\tau.P$ is an agent that is forced to perform an internal τ labelled step before activating P. One possible encoding is represented by the following agent:
$(\langle a\rangle | in(a).P)\backslash a$
where a is not a free name of P. On the other hand, given two programs P and Q, the term $P \oplus Q$ can activate either P or Q by performing a τ labelled step. A possible encoding of $P \oplus Q$ is the following:
$(\langle a\rangle | in(a).P | in(a).Q)\backslash a$
where the name a does not appear free neither in P nor in Q.

Consider now the following agents:

$P \stackrel{def}{=} (out(a).out(b) \mid in(b).inp(a)?out(c)\_out(d))\backslash a\backslash b$

$Q \stackrel{def}{=} \tau.(\tau.\tau.\tau.\tau.out(c) \oplus (\tau.\tau.\tau.out(c) \oplus (\tau.\tau.out(c) \oplus$
$(\tau.out(c) \oplus (\tau|out(d))))))$

It is not difficult to see that under the *unordered* semantics $P \sim_3 Q$: in particular, Q gives rise exactly to the same transitions of P (in other words, it is a syntactic representation of its derivation tree). Instead, P and Q cannot be equivalent in the case of *instantaneous* or *ordered* semantics because $Q \Downarrow \overline{d}$ while $P \not\Downarrow \overline{d}$ in both cases.

We also prove that $\sim_2 \not\subseteq \sim_1$. Let:

$P \stackrel{def}{=} (out(a).out(b) \mid in(a).inp(b)?out(c)\_out(d))\backslash a\backslash b$

$Q \stackrel{def}{=} \tau.(\tau.\tau.out(c) \oplus (\tau.out(c) \oplus (\tau|out(d))))$

In the case of *ordered* semantics, Q has the same transitions of P; hence $P \sim_2 Q$. Instead, under *instantaneous* semantics P and Q cannot be equivalent because $Q \Downarrow \overline{d}$ while $P \not\Downarrow \overline{d}$. $\square$

4 Expressiveness of the Language

In this section we analyze the expressive power of our calculus under the three different semantics.

We first compare the *instantaneous* and the *ordered* semantics. We proceed by studying the possibility of encoding the remaining Linda primitives *rd* and *rdp*, the non-consuming versions of *in* and *inp*, respectively. We show that under the *instantaneous* semantics, the possibility of emitting instantaneously new tuples permits the implementation of these operators, while the *ordered* semantics does not. Moreover, we show that even if the language is extended with the explicit *rd* and *rdp* operators, there exist other coordination primitives not provided in Linda, e.g., a sort of atomic *test-and-set* operator, that can be encoded under the *instantaneous* semantics and not in the *ordered* one.

After, we recall the result presented in [BGZ99]: the language is Turing powerful under the *instantaneous* and *ordered* semantics because it is expressive enough to model any Random Access Machine (RAM) [SS63], while in the case of *unordered* semantics the language is no more Turing powerful.

4.1 Comparing Instantaneous and Ordered Semantics

We first compare the *instantaneous* and the *ordered* semantics by analyzing the problem of implementing the Linda rd coordination primitive.

4.1.1 Encoding the rd primitive of Linda

The language Linda provides also a non-consuming input operator rd. According to [BGZ98], this operator can be modelled by means of a new prefix $rd(a)$ that we introduce by extending the definition of the concurrent programs as follows:

$$C \;::=\; \ldots \mid rd(a).C$$

The semantics is also extended by considering the following axiom and rule:

$$(14) \qquad rd(a).P \xrightarrow{\underline{a}} P$$

$$(15) \qquad \frac{P \xrightarrow{\underline{a}} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q}$$

Axiom (14) introduces the new label $\underline{a}$ as the observable action performed by the $rd(a)$ prefix.[4] The label $\underline{a}$ is different from the standard label a for the way it synchronizes with $\overline{a}$ labels: rule (15) indicates that the process performing the $\overline{a}$ derivation is left unchanged, in this way no tuple $\langle a \rangle$ is consumed.

The rd coordination primitive can be encoded under the *instantaneous* semantics simply by considering an input operation followed by the immediate reintroduction in the TS of the consumed tuple. It is easy to see, for example, that $rd(a).P|\langle a \rangle \overset{\bullet}{\sim} in(a).out(a).P|\langle a \rangle$: in fact, both the agents are able to perform only a reduction step leading to an agent structural congruent to $P|\langle a \rangle$.

In general, we inductively define an encoding $[\![\,]\!]_{rd}$ mapping agents to terms that does not contain the new $rd(a)$ prefix, and we prove the adequacy of this encoding.

For the basic terms $\mathbf{0}$ and $\langle a \rangle$ the encoding simply returns the agents themselves; for composed agents different from $rd(a).P$ the encoding is applied in-

[4] The definition of reduction remains the same also after the introduction of the new label $\underline{a}$; indeed, a transition labelled with $\underline{a}$ cannot be performed in the empty environment, as at least one tuple $\langle a \rangle$ is required.

ductively on the subagents (e.g $[\![inp(a)?P\_Q]\!]_{rd} = inp(a)?[\![P]\!]_{rd}\_[\![Q]\!]_{rd}$). The unique non trivial case is:

$$[\![rd(a).P]\!]_{rd} = in(a).out(a).[\![P]\!]_{rd}$$

The correctness of the encoding under the *instantaneous* semantics is proved by showing that P and $[\![P]\!]_{rd}$ are indistinguishable in every context $C[\,]$. A context is an agent with a hole and $C[P]$ denotes the term obtained by filling the hole with P. Before proving the adequacy of the encoding we need to point out the following facts.

Fact 4.1 Let P be an agent. If $P \xrightarrow{\overline{a}} P'$ then $P \equiv P'|\langle a\rangle$. $\square$

Fact 4.2 Let P be an agent. Under the *instantaneous* semantics

$$\text{if } P \xrightarrow{\alpha} P' \text{ then } \begin{cases} [\![P]\!]_{rd} \xrightarrow{\alpha} [\![P']\!]_{rd} & \text{or} \\ [\![P]\!]_{rd} \xrightarrow{a} \langle a\rangle|[\![P']\!]_{rd} & (\text{with } \alpha = \underline{a}) \end{cases}$$

Moreover,

$$\text{if } [\![P]\!]_{rd} \xrightarrow{\alpha} P' \text{ then } \begin{cases} P \xrightarrow{\alpha} P'' & (\text{with } P' \equiv [\![P'']\!]_{rd}) \text{ or} \\ P \xrightarrow{\underline{a}} P''' & (\text{with } P' = \langle a\rangle|[\![P''']\!]_{rd} \text{ and } \alpha = a) \end{cases}$$

$\square$

Theorem 4.3 Let P be an agent. For every context $C[\,]$, under the *instantaneous* semantics $C[P] \stackrel{\bullet}{\sim} C[[\![P]\!]_{rd}]$.

Proof We show that the relation:

$$\mathcal{R} \;=\; \{(C[P], C[[\![P]\!]_{rd}]), (C[[\![P]\!]_{rd}], C[P]) \mid \text{for every } C[\,] \text{ and } P\}$$

is a barbed bisimulation. We reason up to the structural congruence $\equiv$.
Let $(Q, R) \in \mathcal{R}$. By Fact 4.2 immediately follows that if $Q \downarrow \overline{a}$ then also $R \downarrow \overline{a}$. It is now sufficient to show that if $Q \longrightarrow Q'$ then also $R \longrightarrow R'$ with $(Q', R') \in \mathcal{R}$. We consider the case $Q = C[P]$ and $R = C[[\![P]\!]_{rd}]$ (the other case is treated in a similar way).
If the agent P is not involved in the reduction step, then only the context changes, i.e. $Q' = C'[P]$ for some context $C'[\,]$. But also $R \longrightarrow C'[[\![P]\!]_{rd}]$ and $(C'[P], C'[[\![P]\!]_{rd}]) \in \mathcal{R}$.
Instead, if the agent P is involved then $Q' = C'[P']$ for some context $C'[\,]$ and agent P' such that $P \xrightarrow{\alpha} P'$. By Fact 4.2 there are two cases to analyse.
In the first case $[\![P]\!]_{rd} \xrightarrow{\alpha} [\![P']\!]_{rd}$, hence also $C[[\![P]\!]_{rd}] \longrightarrow C'[[\![P']\!]_{rd}]$ and $(C'[P'], C'[[\![P']\!]_{rd}]) \in \mathcal{R}$.
In the second case $\alpha = \underline{a}$ and $[\![P]\!]_{rd} \xrightarrow{a} \langle a\rangle|[\![P']\!]_{rd}$. The reduction $Q \longrightarrow Q'$ consists of a synchronization between the read operation that P performs, and an output action $T \xrightarrow{\overline{a}} T'$ where T is a subagent of the context $C[\,]$. The fact

that $T \xrightarrow{\overline{a}} T'$ ensures, by Fact 4.1, that $T \equiv T'|\langle a\rangle$.
By applying the structural rules it is possible to put the agents T and P syntactically in contact, hence, without loss of generality, we suppose $C[P] \equiv C''[T|P]$. The fact that $T \equiv T'|\langle a\rangle$ ensures, by substitutivity, that $C''[T|P] \equiv C''[T'|\langle a\rangle|P]$. Thus, the synchronization between T and R leads to the agent $Q' \equiv C''[T'|\langle a\rangle|P']$ (observe that T is left unchanged).
Consider now the agent R; as $Q \equiv C''[T'|\langle a\rangle|[\![P]\!]_{rd}]$ also R is structurally congruent to $C''[T'|\langle a\rangle|[\![P]\!]_{rd}]$. Because of the derivation $[\![P]\!]_{rd} \xrightarrow{a} \langle a\rangle|[\![P']\!]_{rd}$, the agent R is able to perform the reduction step $R \longrightarrow C''[T'|\langle a\rangle|[\![P']\!]_{rd}]$. Finally, it is sufficient to observe that $(C''[T'|\langle a\rangle|P'], C''[T'|\langle a\rangle|[\![P']\!]_{rd}]) \in \mathcal{R}$. $\square$

In order to prove that the *rd* primitive cannot be implemented under the *ordered* semantics, we consider the problem of implementing a particular agent called *tester for presence*, and we show that it can be implemented under the *ordered* semantics only by making use of the *rd* primitive.

Definition 4.4 An agent P_a is a *tester for presence of tuple* $\langle a\rangle$ iff:

$$\begin{array}{lll} (1) & P_a & \not\Downarrow \overline{p_a} \\ (2) & P_a|\langle a\rangle & \Downarrow \overline{p_a} \\ (3) & P_a|\langle a\rangle|inp(a)?\mathbf{0}\_out(t_a) & \not\Downarrow \overline{t_a} \end{array}$$

where p_a and t_a are two message names indicating the presence of the tuple $\langle a\rangle$ and the fact that its absence has been tested, respectively. $\square$

The following agent is clearly a tester for presence of tuple $\langle a\rangle$:

$$P'_a \stackrel{def}{=} rd(a).out(p_a)$$

Instead, its encoding $[\![P'_a]\!] = in(a).out(a).out(p_a)$ is not a tester under the *ordered* semantics. It is enough to observe the following sequence of reduction steps, permitted under the *ordered* interpretation, that invalidates item (3) of Definition 4.4:

$$\begin{array}{ll} in(a).out(a).out(p_a)|\langle a\rangle|inp(a)?\mathbf{0}\_out(t_a) & \longrightarrow \\ out(a).out(p_a)|inp(a)?\mathbf{0}\_out(t_a) & \longrightarrow \\ out(a).out(p_a)|out(t_a) & \longrightarrow \\ out(a).out(p_a)|\langle t_a\rangle \ \downarrow \overline{t_a} & \end{array}$$

In the following we prove that there is no way to implement a tester for presence under the *ordered* semantics without making use of the *rd* primitive.

We first point out a fact and recall Proposition 3.5.

Fact 4.5 Let P be an agent. If $P \Longrightarrow P'$ then for every a, Q_1 and Q_2 also $P|in(a).Q_1 \Longrightarrow P'|in(a).Q_1$ and $P|inp(a)?Q_1\_Q_2 \Longrightarrow P'|inp(a)?Q_1\_Q_2$. $\Box$

Proposition 3.5 states that, given an agent P interpreted following the *ordered* semantics, if $P \not\downarrow \overline{a}$ and $P \xrightarrow{b} P'$ then also $P' \not\downarrow \overline{a}$. This means that under the *ordered* semantics it is not possible to instantaneously emit new tuples after a consumption step. This is not true under the *instantaneous* semantics; e.g., consider the program $in(b).out(a)$ having the following derivation

$$in(b).out(a) \xrightarrow{b} out(a) \ \equiv \ \langle a \rangle \ \downarrow \overline{a}$$

while it is clear that $in(b).out(a) \not\downarrow \overline{a}$.

The following lemma analyzes each agent P such that $P \downarrow \overline{a}$ and $P \xrightarrow{a} P'$. The fact that $P \downarrow \overline{a}$ indicates that the agent P contains a tuple $\langle a \rangle$; the derivation $P \xrightarrow{a} P'$ ensures that it is also able to consume another tuple $\langle a \rangle$. We prove that P can also consume its internal tuple without looking for an external one. Formally, also the derivation $P \xrightarrow{\tau} P''$ with $P''|\langle a \rangle \equiv P'$ holds.

Lemma 4.6 Let P be an agent interpreted under the *ordered* semantics. If $P \xrightarrow{a} P'$ and $P \downarrow \overline{a}$, then also $P \xrightarrow{\tau} P''$ with $P''|\langle a \rangle \equiv P'$. $\Box$

We now prove that if an agent P satisfies (1) and (2) of Definition 4.4, then it is able to consume a tuple $\langle a \rangle$ if present in TS.

Proposition 4.7 Let P be an agent interpreted under the *ordered* semantics. If $P \not\Downarrow \overline{p_a}$ and $P|\langle a \rangle \Downarrow \overline{p_a}$, then there exists P' such that $P|\langle a \rangle \Longrightarrow P'$ with $P' \not\downarrow \overline{a}$.

Proof If $P|\langle a \rangle \Downarrow \overline{p_a}$ then there exists $n \geq 0$ such that

$$P|\langle a \rangle \longrightarrow P_1 \longrightarrow \ldots \longrightarrow P_n \quad \text{with } P_n \downarrow \overline{p_a}$$

We proceed by induction on n.
In the base case $n = 0$, then $P|\langle a \rangle = P_n$ and $P|\langle a \rangle \downarrow \overline{p_a}$. It is easy to see that this implies the contradiction $P \downarrow \overline{p_a}$. Hence, n is strictly greater than 0.
In the inductive case we consider the first step $P|\langle a \rangle \longrightarrow P_1$. There are two cases to analyze.
In the first case $P_1 \equiv P_1'|\langle a \rangle$ with $P \longrightarrow P_1'$. The thesis directly follows by the inductive hypothesis that can be applied to P_1'; in fact, $P_1'|\langle a \rangle \Downarrow \overline{p_a}$ and also $P_1' \not\Downarrow \overline{p_a}$, otherwise P could weakly commit $\overline{p_a}$.
In the second case $P_1 \equiv P_1'$ with $P \xrightarrow{a} P_1'$. We first suppose that $P \downarrow \overline{a}$, and after we analyze tha case in which $P \not\downarrow \overline{a}$.
If $P \downarrow \overline{a}$, the fact that also $P \xrightarrow{a} P_1'$ ensures that, by Lemma 4.6, $P \longrightarrow P_1''$ with $P_1' \equiv P_1''|\langle a \rangle$. The thesis directly follows by the inductive hypothesis, that can be applied to P_1''; in fact, $P_1''|\langle a \rangle \Downarrow \overline{p_a}$ (as $P_1' \equiv P_1$ and $P_1 \Downarrow \overline{p_a}$) and also $P_1'' \not\Downarrow \overline{p_a}$, otherwise P could weakly commit $\overline{p_a}$.

If $P \not\downarrow \overline{a}$, the fact that $P \xrightarrow{a} P_1'$ ensures that also $P_1' \not\downarrow \overline{a}$ by Proposition 3.5. Thus, also $P_1 \not\downarrow \overline{a}$ as $P_1 \equiv P_1'$. $\Box$

We are now able to prove that it is not possible, under the *ordered* semantics, to implement a tester for presence of tuple $\langle a \rangle$ for any a.

Theorem 4.8 Let P be an agent interpreted under the *ordered* semantics. For any message name a, P is not a tester for presence of tuple $\langle a \rangle$.

Proof By contradiction suppose that P is a tester for presence of tuple $\langle a \rangle$. Item (1) and (2) of Definition 4.4 ensure that $P \not\Downarrow \overline{p_a}$ and $P|\langle a \rangle \Downarrow \overline{p_a}$. By Proposition 4.7 there exists P' such that $P|\langle a \rangle \Longrightarrow P'$ with $P' \not\downarrow \overline{a}$.
By Fact 4.5 also $P|\langle a \rangle|inp(a)?\mathbf{0}\_out(t_a) \Longrightarrow P'|inp(a)?\mathbf{0}\_out(t_a)$. The fact that $P' \not\downarrow \overline{a}$ ensures that the else branch of the inp could be chosen; in this way the tuple $\langle t_a \rangle$ could be emitted invalidating (3) of Definition 4.4. $\Box$

In the following we show that the introduction of the rd operator is not sufficient to cover the gap of expressivity between the *instantaneous* and the *ordered* semantics. In order to prove this, we consider the problem of encoding the Linda rdp primitive.

4.1.2 Encoding the rdp *primitive of Linda*

The language Linda provides also a non-consuming input predicate rdp. According to [BGZ98], we could model this operator by extending the definition of the concurrent programs as follows:

$$C \ ::= \ \ldots \mid rdp(a)?C\_C$$

The semantics is also extended by adding the axioms:

$$(16) \qquad rdp(a)?P\_Q \xrightarrow{a} P$$
$$(17) \qquad rdp(a)?P\_Q \xrightarrow{\neg a} Q$$

Axiom (16) indicates that if a tuple $\langle a \rangle$ is available, the agent $rdp(a)?P\_Q$ becomes P, otherwise it behaves like Q (axiom (17)).

The rdp coordination primitive can be encoded under the *instantaneous* semantics simply by considering an inp operation: if the required tuple is consumed then it is immediately reintroduced in the TS. For example, it is easy to see that under the *instantaneous* semantics $rdp(a)?P\_Q \overset{\bullet}{\sim} inp(a)?(out(a).P)\_Q$ and also $(rdp(a)?P\_Q)|\langle a \rangle \overset{\bullet}{\sim} (inp(a)?(out(a).P)\_Q)|\langle a \rangle$. In the first case only a reduction step leading to Q is permitted; in the second case both the terms become structurally congruent to $P|\langle a\rangle$.

Also for the *rdp* operator, we inductively define an encoding $[\![\,]\!]_{rdp}$ mapping generic agents in terms without the new *rdp* construct. Also for this encoding there exists a unique non trivial case:

$$[\![rdp(a)?P\_Q]\!]_{rdp} \;=\; inp(a)?(out(a).[\![P]\!]_{rdp})\_[\![Q]\!]_{rdp}$$

Theorem 4.9 Let P be an agent. For every context $C[\,]$, under the *instantaneous* semantics $C[P] \stackrel{\bullet}{\sim} C[[\![P]\!]_{rdp}]$.

Proof First of all we observe that Fact 4.2 holds also if $[\![\,]\!]_{rdp}$ is substituted for $[\![\,]\!]_{rd}$. After, the same reasoning followed in the proof of Theorem 4.3 can be used to prove that also the relation:

$$\mathcal{R} \;=\; \{(C[P], C[[\![P]\!]_{rdp}]), (C[[\![P]\!]_{rdp}], C[P]) \mid \text{for every } C[\,] \text{ and } P\}$$

is a barbed bisimulation up to $\equiv$. $\Box$

In order to prove that the *rdp* primitive cannot be implemented under the *ordered* semantics, we consider the problem of implementing a particular agent called *tester for absence*, and we show that it can be implemented under the *ordered* interpretation only by making use of the new *rdp* primitive.

Definition 4.10 An agent A_a is a *tester for absence of tuple* $\langle a\rangle$ iff:

$$\begin{array}{ll}
(1)\ A_a & \Downarrow \overline{a_a} \\
(2)\ A_a|\langle a\rangle & \not\Downarrow \overline{a_a} \\
(3)\ A_a|\langle a\rangle|inp(a)?\mathbf{0}_out(t_a) & \not\Downarrow \overline{t_a}
\end{array}$$

where a_a is a message name indicating the absence of tuple $\langle a\rangle$. $\Box$

The following agent is clearly a tester for absence of tuple $\langle a\rangle$:

$$A'_a \stackrel{def}{=} rdp(a)?\mathbf{0}\_out(a_a)$$

Instead, its encoding $[\![A'_a]\!] = inp(a)?out(a)\_out(a_a)$ is not a tester under the *ordered* semantics. It is enough to observe the following sequence of reduction steps, permitted under the *ordered* interpretation, that invalidates item (3) of

Definition 4.10:

$$\begin{array}{ll} rdp(a)?\mathbf{0}\_out(a_a)|\langle a\rangle|inp(a)?\mathbf{0}\_out(t_a) & \longrightarrow \\ out(a)|inp(a)?\mathbf{0}\_out(t_a) & \longrightarrow \\ out(a)|out(t_a) & \longrightarrow \\ out(a)|\langle t_a\rangle \ \ \downarrow \overline{t_a} & \end{array}$$

In the following we prove that there is no way to implement a tester for absence under the *ordered* semantics without making use of the rdp primitive. In order to prove this we will consider the initial language extended only with the $rd(a)$ prefix.

Before proceeding in our proof we need to observe that if an agent P is able to perform a derivation labeled with $\neg a$ then $P \not\downarrow \overline{a}$.

Fact 4.11 Let P be an agent. If there exists P' such that $P \xrightarrow{\neg a} P'$ then $P \not\downarrow \overline{a}$. $\square$

The following lemma states that each agent that does not contain any rdp constructs is able to perform a $\neg a$ derivation, it is also able to perform another one labeled with a. This is due to the fact that the derivations labeled with $\neg a$ are induced by an $inp(a)?P\_Q$ term; this term also allows a derivation labeled with a. In other words, every agent testing the absence of a certain tuple is also able to consume it.

Lemma 4.12 Let P be an agent. If $P \xrightarrow{\neg a} P'$ then there exists P'' such that $P \xrightarrow{a} P''$. $\square$

We now prove that if an agent P satisfies (1) and (2) of Definition 4.10, then it is able to consume a tuple $\langle a\rangle$ if present in TS.

Proposition 4.13 Let P be an agent interpreted under the *ordered* semantics. If $P \Downarrow \overline{a_a}$ and $P|\langle a\rangle \not\Downarrow \overline{a_a}$, then there exists P' such that $P|\langle a\rangle \Longrightarrow P'$ with $P' \not\downarrow \overline{a}$.

Proof If $P \Downarrow \overline{a_a}$ then there exists $n \geq 0$ such that
$P \longrightarrow P_1 \longrightarrow \ldots \longrightarrow P_n$ with $P_n \downarrow \overline{a_a}$
We proceed by induction on n.
In the base case $n = 0$, then $P = P_n$ and $P \downarrow \overline{a_a}$. This implies the contradiction $P|\langle a\rangle \downarrow \overline{a_a}$; hence, n is strictly greater than 0.
In the inductive case, we consider the reduction step $P \longrightarrow P_1$ and the corresponding labeled transition $P \xrightarrow{\alpha} P_1$ with $\alpha = \tau$ or $\alpha = \neg b$ for some b.
If $\alpha = \tau$ or $\alpha = \neg b$ with $b \neq a$ the thesis directly follows by the inductive hypothesis. In fact, it can be applied on P_1' because $P_1' \Downarrow \overline{a_a}$ and also $P_1'|\langle a\rangle \not\Downarrow \overline{a_a}$;

otherwise $P|\langle a\rangle$ could weakly commit $\overline{a_a}$.
If $\alpha = \neg a$ then by Fact 4.11 $P \not\Downarrow \overline{a}$ and by Lemma 4.12 there exists P'' such that $P \xrightarrow{a} P''$. In this way a tuple $\langle a\rangle$ can be consumed, i.e. $P|\langle a\rangle \longrightarrow P''$. The fact that $P \not\Downarrow \overline{a}$ ensures that $P'' \not\Downarrow \overline{a}$ by Proposition 3.5 (that holds also in the presence of the *rd* operation). $\square$

We are now able to prove that it is not possible, under the *ordered* semantics, to implement a tester for absence of tuple $\langle a\rangle$ for any a.

Theorem 4.14 Let P be an agent interpreted under the *ordered* semantics. For any message name a, P is not a tester for absence of tuple $\langle a\rangle$.

Proof Similar to the proof of Theorem 4.8, where Proposition 4.13 is used instead of Proposition 4.7. $\square$

In the following we show that the introduction of the *rdp* operator is not sufficient to cover the gap of expressivity between the *instantaneous* and the *ordered* semantics. In other words, even if all the Linda coordination primitives are considered, the *instantaneous* semantics makes the language more expressive. In order to prove this, we consider the problem of encoding a sort of *test-and-set* operator that atomically tests the absence of a tuple and produces it in the case it is not available.

4.1.3 Encoding an atomic test-and-set *primitive*

We consider a *test-and-set* operator that atomically verifies the absence of a certain tuple and, if it is not available, produces a new occurrence of it.

This atomic test-and-set operator can be modelled in our language by extending the definition of the concurrent programs as follows:

$$C \ ::= \ ... \mid t\&s(a)?C\_C$$

The semantics is also extended by adding the following axioms:

$$\begin{array}{ll}(18) & t\&s(a)?P\_Q \xrightarrow{a} P \\ (19) & t\&s(a)?P\_Q \xrightarrow{\neg a} \langle a\rangle|Q\end{array}$$

If a tuple $\langle a\rangle$ is available then the agent $t\&s(a)?P\_Q$ evolves to P (axiom (18)); otherwise, a new occurrence of $\langle a\rangle$ is emitted and the agent evolves to Q in a single step (axiom (19)).

The test-and-set primitive can be encoded under the *instantaneous* semantics simply by using a *rdp* operation that instantaneously produce a new tuple after having tested its absence. It is easy to see, for example, that $t\&s(a)?P\_Q$ and $rdp(a)?P\_(out(a).Q)$ have the same possible derivations, leading to structurally equivalent agents. In fact, the second agent has only the following derivations:

$$rdp(a)?P\_(out(a).Q) \xrightarrow{a} P$$
$$rdp(a)?P\_(out(a).Q) \xrightarrow{\neg a} out(a).Q \equiv \langle a\rangle|Q$$

In general, we inductively define an encoding $[\![\,]\!]_{t\&s}$ mapping agents to terms that does not contain the test-and-set operator, and we prove the adequacy of this encoding. Also in this encoding there exists a unique non trivial case:

$$[\![t\&s(a)?P\_Q]\!]_{t\&s} \;=\; rdp(a)?[\![P]\!]_{t\&s}\_(out(a).[\![Q]\!]_{t\&s})$$

This time, the correctness of the encoding is stronger: every agent P and its encoding $[\![P]\!]_{t\&s}$ have the same derivations leading to structurally congruent terms.

Theorem 4.15 Let P be an agent. Under the *instantaneous* semantics $P \xrightarrow{\alpha} P'$ if and only if $[\![P]\!]_{t\&s} \xrightarrow{\alpha} P''$ with $[\![P']\!]_{t\&s} \equiv P''$.

Proof By induction on the proof of the derivation. □

In order to prove that the atomic test-and-set primitive cannot be implemented under the *ordered* semantics, we consider the problem of implementing a particular agent called *mutually exclusive producer*, and we show that it cannot be implemented under the *ordered* interpretation.

Definition 4.16 An agent M_a is a *mutually exclusive producer of tuple* $\langle a\rangle$ iff:

(1)	M_a	$\Downarrow \overline{a}$
(2)	$M_a\|M_a\|in(a).in(a).out(t_a)$	$\not\Downarrow \overline{t_a}$

where t_a is a message name indicating that at least two occurrences of the tuple $\langle a\rangle$ have been detected. □

The idea underlying a mutually exclusive producer is that it is able to produce a new occurrence of a tuple; but, in the presence of another producer, only one is enabled to perform the emission operation. Before producing the tuple, a mutually exclusive producer must in some way verify the presence of another concurrent producer.

The following agent is clearly a mutually exclusive producer of tuple $\langle a \rangle$:

$$M'_a \stackrel{def}{=} t\&s(b)?\mathbf{0}\_out(a)$$

At the beginning of the computations no tuple $\langle b \rangle$ and $\langle a \rangle$ are present. When one mutually exclusive producer begins its computation, it verifies the absence of $\langle b \rangle$, immediately produces a first occurrence of $\langle b \rangle$, and is then enabled to produce $\langle a \rangle$; the second agent verifies the presence of $\langle b \rangle$ and terminates without producing any $\langle a \rangle$.

Instead, its encoding $[\![M'_a]\!]_{t\&s} = rdp(b)?\mathbf{0}\_(out(b).out(a))$ is not a mutually exclusive producer under the *ordered* semantics. It is enough to observe the following sequence of reduction steps, permitted following the *ordered* interpretation, that invalidates item (2) of Definition 4.16:

$$\begin{array}{ll}
rdp(b)?\mathbf{0}_(out(b).out(a))|rdp(b)?\mathbf{0}_(out(b).out(a))|in(a).in(a).out(t_a) & \longrightarrow \\
out(b).out(a)|rdp(b)?\mathbf{0}_(out(b).out(a))|in(a).in(a).out(t_a) & \longrightarrow \\
out(b).out(a)|out(b).out(a)|in(a).in(a).out(t_a) & \Longrightarrow \\
\langle b \rangle|\langle b \rangle|\langle a \rangle|\langle a \rangle|in(a).in(a).out(t_a) & \Longrightarrow \\
\langle t_a \rangle \quad \downarrow \overline{t_a} &
\end{array}$$

In the following we prove that there is no way to implement such a producer in our calculus (extended with rd and rdp) under the *ordered* semantics.

The following lemma states that under the *ordered* semantics no new tuples can be emitted during the execution of a $\neg a$ derivation; in other words, there is no way to instantaneously emit new tuples after the execution of a test for absence.

Lemma 4.17 Let P be an agent interpreted under the *ordered* semantics such that $P \stackrel{\neg a}{\longrightarrow} P'$ for some P'. If $P \not\downarrow \overline{b}$ then also $P' \not\downarrow \overline{b}$. $\square$

We now prove that under the *ordered* semantics, whenever an agent able to produce a tuple $\langle a \rangle$ is composed in parallel with another occurrence of itself, then at least two tuples $\langle a \rangle$ can be produced.

Proposition 4.18 Let P be an agent interpreted under the *ordered* semantics. If $P \Downarrow \overline{a}$ then there exist P', P'', and P''' such that
$P|P \Longrightarrow P' \stackrel{\overline{a}}{\longrightarrow} P'' \stackrel{\overline{a}}{\longrightarrow} P'''$

Proof If $P \Downarrow \overline{a}$ then there exists $n \geq 0$ such that
$P \longrightarrow P_1 \longrightarrow \ldots \longrightarrow P_n \quad$ with $P_n \downarrow \overline{a}$

We proceed by induction on n.
In the base case $n = 0$, then $P = P_n$ and $P \downarrow \overline{a}$. This ensures that there exists Q such that $P \xrightarrow{\overline{a}} Q$. Hence, $P|P \xrightarrow{\overline{a}} Q|P \xrightarrow{\overline{a}} Q|Q$.
In the inductive case, we consider the reduction step $P \longrightarrow P_1$ and the corresponding labeled transition $P \xrightarrow{\alpha} P_1$ with $\alpha = \tau$ or $\alpha = \neg b$ for some b.
If $\alpha = \tau$ then $P|P \xrightarrow{\tau} P_1|P \xrightarrow{\tau} P_1|P_1$. The thesis directly follows by the inductive hypothesis as it can be applied to P_1.
If $\alpha = \neg b$ then by Fact 4.11 (holding also in the presence of the *rdp* operator) $P \not\downarrow \overline{b}$; as $P \xrightarrow{\neg b} P_1$, by Lemma 4.17 also $P_1 \not\downarrow \overline{b}$. This ensures

$$P|P \xrightarrow{\neg b} P_1|P \xrightarrow{\neg b} P_1|P_1$$

The thesis directly follows by the inductive hypothesis. $\square$

We are now able to prove that it is not possible, under the *ordered* semantics also in the presence of the *rd* and *rdp* operators, to implement a mutually exclusive producer of tuple $\langle a \rangle$ for any a.

Theorem 4.19 Let P be an agent interpreted under the *ordered* semantics. For any message name a, P is not a mutually exclusive producer of tuple $\langle a \rangle$.

Proof By contradiction suppose that P is the mutually exclusive producer. Item (1) of Definition 4.16 ensures that $P \Downarrow \overline{a}$. By Proposition 4.18 there exist P', P'' and P''' such that $P|P \Longrightarrow P' \xrightarrow{\overline{a}} P'' \xrightarrow{\overline{a}} P'''$. By Fact 4.5 (holding also in the presence of the *rd* and *rdp* operators) then also

$$\begin{array}{ll} P|P|in(a).in(a).out(t_a) & \Longrightarrow \\ P'|in(a).in(a).out(t_a) & \longrightarrow \\ P''|in(a).out(t_a) & \longrightarrow \\ P'''|out(t_a) & \longrightarrow \\ P'''|\langle t_a \rangle \quad \downarrow \overline{t_a} & \end{array}$$

invalidating item (2) of Definition 4.16. $\square$

We can conclude that in the complete language Linda, it is not possible to implement the above atomic test-and-set operator if the intended semantics is the *ordered* one.

Besides this atomic test-and-set coordination primitive, it is interesting to point out the existence of an entire class of coordination primitives that can be embedded under the *instantaneous* semantics and not under the *ordered* one. In particular we could think to generalize the test-and-set operator in the following way. Given a positive natural number n, an atomic *test-and-n-set* operator is able to atomically test the absence of the tuple $\langle a \rangle$ and, in the case it is absent, produce n occurrences of that tuple.

Under the instantaneous semantics a test-and-n-set operator can be embedded following the approach described above for the test-and-set operator. Consider, for example, the agent

$$rdp(a)?P\_(\underbrace{out(a).out(a).\ldots.out(a)}_{n \text{ times}}.Q)$$

which is able to produce n occurrences of the tuple $\langle a \rangle$ immediately after having tested the absence of such this tuple.

Following the same proof technique used for the standard test-and-set operator, it is possible to prove that also a test-and-n-set operator cannot be embedded, for any positive natural number n, if the intended semantics is the *ordered* one.

4.2 Comparing Ordered and Unordered Semantics

A RAM is a computational model consisting of a finite set of registers that can hold arbitrary large natural numbers and of a program, that is a sequence of simple numbered instructions, like arithmetical operations on the contents of registers or conditional jumps.

To perform a computation, the inputs are provided in registers $r_1, \ldots, r_n$; if other registers are used in the program, they are supposed to contain the value 0 at the beginning of the computation. The execution of the program begins with the first instruction and continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached; this happens if the program was executing the last instruction of the program and this instruction does not require a jump, or if the current instruction requires a jump to an instruction number not appearing in the program. If the program terminates, the result of the computation is the content of the registers specified as outputs.

In [Min67] it is shown that the following two instructions are sufficient to model every recursive function:

- $Succ(r_j)$: add 1 to the content of register r_j;
- $DecJump(r_j, s)$: if the contents of register r_j is not zero, then decrease it by 1 and go to the next instruction, otherwise jump to instruction s.

For example, the following program computes the sum of registers r_1 and r_2, putting the result in register r_1 (note that the third instruction corresponds to an unconditional jump, because register r_3 contains the value 0 at the beginning of the computation and its contents is never modified by the program):

$1 : DecJump(r_2, 4)$

$2 : Succ(r_1)$

$3 : DecJump(r_3, 1)$

In the following we will present how to encode any RAM in our language under the ordered or the instantaneous interpretations. In this translation we model the contents of registers and the program counter by means of tuples: if register r_j contains the number n, then n tuples $\langle r_j \rangle$ are in the tuple space; if the next instruction to execute is the i^{th}, then TS contains the tuple $\langle p_i \rangle$.

To model the instruction we proceed in the following way: a $Succ$ instruction on register r_j at position i is represented by an agent that consumes the "program counter tuple", adds a tuple $\langle r_j \rangle$ and updates the program counter by adding a tuple $\langle p_{i+1} \rangle$; an instruction $DecJump(r_j, s)$ at position i is modeled by an agent that, after consuming the "program counter tuple", performs an inp on message r_j; if the operation succeeds, then a tuple $\langle r_j \rangle$ has been withdrawn from the tuple space and the agent updates the program counter by adding $\langle p_{i+1} \rangle$, otherwise a jump to the s^{th} instruction is performed by adding $\langle p_s \rangle$. The use of the recursion operator in the representation of the instructions permits to reuse them.

$$\begin{aligned}
\llbracket i : Succ(r_j) \rrbracket &\stackrel{def}{=} rec\, X.(in(p_i).out(r_j).out(p_{i+1}).X) \\
\llbracket i : DecJump(r_j, s) \rrbracket &\stackrel{def}{=} rec\, X.(in(p_i).inp(r_j)?(out(p_{i+1}).X)_(out(p_s).X))
\end{aligned}$$

The agent modeling the program $I_1 \ldots I_k$ with inputs $n_1, \ldots, n_m$ is:

$$\langle p_1 \rangle | \underbrace{\langle r_1 \rangle | \ldots | \langle r_1 \rangle}_{n_1 \text{ times}} | \ldots | \underbrace{\langle r_m \rangle | \ldots | \langle r_m \rangle}_{n_m \text{ times}} | \llbracket I_1 \rrbracket | \ldots | \llbracket I_k \rrbracket$$

The unordered case

The implementation of the RAM we have presented, is not correct for the unordered semantics because of problems in updating the program counter. Consider an execution of a program with instructions:

$i \quad : Succ(r_j)$

$i + 1 : DecJump(r_j, s)$

in which the register r_j is empty at the moment the i^{th} instruction is executed. The implementation of the $Succ(r_j)$ instruction creates two tuples: $\langle r_j \rangle$ and the new "program counter tuple" $\langle p_{i+1}\rangle$. If the tuple $\langle p_{i+1}\rangle$ becomes available before $\langle r_j\rangle$, the following $DecJump(r_j, s)$ instruction could execute the jump because no tuple $\langle r_j\rangle$ is available. In the *instantaneous* and *ordered* semantics this kind of problems cannot arise because the "program counter tuple" becomes available simultaneously (in the instantaneous case) or only after (in the ordered case) the tuple $\langle r_j\rangle$. Under *unordered* semantics the order of rendering of the tuples is not predictable, hence the wrong jump could be performed.

We not only say that the implementation we have presented is not correct under *unordered* semantics, but we also assert that the RAM is not implementable in any way. In fact, under this semantics, the language is no more Turing powerful; in [BGZ99] we show that the problem of termination is decidable under the unordered semantics. [5]

The proof is divided into two steps: we first define a net semantics in terms of contextual P/T nets (i.e., P/T nets extended with arcs testing for presence or absence of tokens in a place; see, e.g., [MR95,BP95]). This semantics, defined following the style of [BG95,BGZ97a], preserves the interleaving behaviour, hence also the possibility of deadlock. Then, given the contextual P/T net semantics, we present a mapping on finite (standard) P/T nets that preserves deadlock. As deadlock is decidable on finite P/T nets, we conclude that the termination problem is decidable under the unordered semantics.

5 Conclusion and Future Research

Three different interpretations of the output operation are studied: we compare them from the point of view of both their behavioural semantics and their expressive power.

We think this is a first necessary step, not only in order to equip with a formal semantics Linda itself, but also to have a deeper understanding of a class of Linda based coordination models, such as JavaSpaces [JS98] or T Spaces [WLLF98]. For instance, in the reference manual [SCA95] of Linda, it is often unclear which is the real interpretation of the *out* primitive. As an illustrative example, consider $Q = (out(b).out(a) \mid in(a).inp(b)?C\_D)$. If we assume the ordered semantics, then the input of a is possible only if $\langle b\rangle$ is already in TS; hence, the execution of *inp* will always enable C. Differently, if we assume the unordered semantics, then no guarantee is given that $\langle b\rangle$ is in TS; hence,

[5] Even if the language adopted in [BGZ99] is a slight variation of the calculus considered here (it uses input guarded replication instead of guarded recursion and does not contain restriction), the proof of non Turing equivalence can be easily adapted.

it is sometimes possible that D is executed instead. The choice between the two semantics is not solved in [SCA95] and other similar publications (e.g., [Nar90]). For instance, on page 2-6 of [SCA95] we can read: "*out* returns after the tuple has been added to tuple space", hence supporting the claim that the intended semantics is ordered. On the other hand, on page 2-26 of [SCA95] a comment to a program reported on page 2-25 expresses a concern very similar to the above about the possible executability of D, hence validating that the intended semantics is the ordered one. Similar contradictions, regarding shared memory implementations, can be found in [Nar90]: on lines 19-20 of page 4 it is said that it is not ensured that "an out followed by a predicate operation on the same tuple will succeed", hence supporting that the intended semantics is unordered. Instead, on lines 7-9 of the same page we can read: "The time at which the tuple is visible to other processes is indeterminate. In the existing shared memory implementations, the operation is completed immediately.", hence validating the ordered semantics approach.

Besides the two interpretations discussed above, we consider also the *instantaneous* semantics because it corresponds to the way asynchronous communication is modeled in previous proposals for asynchronous process algebras [HT91,Bou92]. Nevertheless, in our setting, this approach has only a theoretical relevance, because an implementation seems unrealistic: indeed, in some circumstances it requires the atomic emission of an unbounded amount of new tuples.

From the point of view of the behavioural semantics we have found out that the three obtained congruences are pairwise different and none of them is includes in any one of the others. Regarding the expressive power, we have shown a precise hierarchy among the three semantics. In particular, we have listed some coordination primitives that have a direct implementation under the *instantaneous* semantics and not under the *ordered* one, and we have shown that the calculus is Turing powerful if the intended semantics is *instantaneous* or *ordered*, while this is not the case under the *unordered* interpretation. Indeed, under this approach the problem of termination/deadlock becomes decidable.

We observe that a direct comparison of the three operational semantics can help in deciding properties of programs. For instance, it is clear that three transition systems for any program P under the three semantics are in a very precise relation: the transition system of the ordered semantics is obtained by pruning some transitions from the one for the unordered and, in turn, the transition system for the instantaneous semantics is obtained by pruning some transitions of the one for the ordered semantics. Therefore, if P is deadlock-free under the unordered semantics (and we have proven that this is a decidable property), then P has no deadlock also under the ordered and the instantaneous semantics. We plan to investigate further this issue.

We would like to mention that we have left for future research the investigation of further, interesting implementations of the output operation like, e.g. [Nar90], according to which the tuple generated by a process by means of an *out* operation "will be visible to the same process by the time the next Linda operation executes", while the tuple "is not guaranteed to be visible to another process until some variable latency period has past".

The study carried out in this paper can be extended to cope with further coordination primitives, as, e.g., the *rd* and *rdp* Linda operators described in Section 4; as far as the behavioural semantics is concerned, these have been already studied in [BGZ98] for the ordered case only.

A Proofs of Section 3

Theorem 3.9 Let P and Q be agents interpreted under the *ordered* semantics. If $P|R \stackrel{\bullet}{\sim} Q|R$ for every agent R, then $P \sim Q$.

Proof Let P and Q be two agents satisfying the premises of the theorem. Let $L = fn(P) \cup fn(Q)$; observe that L is finite.

We show that the pair (P, Q) is contained in a $\neg$-bisimulation (up to $\equiv$), hence $P \sim Q$. In particular, we define the following agent R:

$$R \stackrel{def}{=} \prod_{l \in L} Ag_l^1 | \prod_{l \in L} Ag_l^2 | \prod_{l \in L} Ag_l^3 | \prod_{l \in L} Ag_l^4 | \prod_{l \in L} Ag_l^5$$

such that the relation:

$$\mathcal{R} = \{(S, T) \mid S|R \stackrel{\bullet}{\sim} T|R \ and \ fn(S), fn(T) \subseteq L\}$$

is a $\neg$-bisimulation (up to $\equiv$). The pair (P, Q) is in $\mathcal{R}$ because $P|R$ is barbed bisimilar to $Q|R$ and both $fn(P)$ and $fn(Q)$ are subsets of L.

The agents Ag_l^i are defined as follows:

$$\begin{aligned}
Ag_l^1 &\stackrel{def}{=} rec\,X.in(l).out(b_l^1).in(b_l^1).X \\
Ag_l^2 &\stackrel{def}{=} rec\,X.in(l).out(b_l^2).out(c_l).in(b_l^2).X \\
Ag_l^3 &\stackrel{def}{=} rec\,X.out(l).out(b_l^3).in(b_l^3).X \\
Ag_l^4 &\stackrel{def}{=} rec\,X.in(c_l).out(b_l^4).out(l).in(b_l^4).X \\
Ag_l^5 &\stackrel{def}{=} rec\,X.out(l).out(b_l^5).in(l).in(b_l^5).X
\end{aligned}$$

where b_l^i and c_l are all fresh and distinct names for every i and l. The tuples $\langle b_l^i \rangle$ are called *presence tokens*: each agent Ag_l^i (and only it) is able to generate and consume the corresponding presence token $\langle b_l^i \rangle$. Moreover, for every agent Ag_l^i, if $Ag_l^i \xrightarrow{\alpha} R'$, then $R' \longrightarrow R'' \downarrow \overline{b_l^i}$, i.e., if one of the subagents of R performs a transition step, then the corresponding presence token can be produced after one single reduction step.

In order to prove that $\mathcal{R}$ is a $\neg$-bisimulation we first observe that $\mathcal{R}$ is symmetric because of the symmetricity of $\overset{\bullet}{\sim}$. After, it is enough to proceed by case analysis on the possible derivations $S \xrightarrow{\alpha} S'$ proving that in each case T is able to reply according to the definition of the $\neg$-bisimulation.

- $S \xrightarrow{\overline{a}} S'$:
  Consider the following sequence of derivations that the agent $S|R$ can perform because of the presence of the agent Ag_a^1. Let R' be the term:

  $$\prod_{l \in L \setminus \{a\}} Ag_l^1 | \prod_{l \in L} Ag_l^2 | \prod_{l \in L} Ag_l^3 | \prod_{l \in L} Ag_l^4 | \prod_{l \in L} Ag_l^5$$

  then:

  $$\begin{aligned} S|R &\longrightarrow S'|out(b_a^1).in(b_a^1).Ag_a^1|R' & (\stackrel{def}{=} V_1) \\ &\longrightarrow S'|\langle b_a^1 \rangle|in(b_a^1).Ag_a^1|R' & (\stackrel{def}{=} V_2) \\ &\longrightarrow S'|R & (\stackrel{def}{=} V_3) \end{aligned}$$

  Observe that $V_2 \downarrow \overline{b_a^1}$ while $V_3 \not\downarrow \overline{b_a^1}$. The agent $T|R$ is barbed bisimilar to $S|R$, hence:

  $$T|R \longrightarrow W_1 \longrightarrow W_2 \longrightarrow W_3$$

  where $V_i \overset{\bullet}{\sim} W_i$. Then, also $T|R$ must generate and then consume the presence token $\langle b_a^1 \rangle$ (because $W_2 \downarrow \overline{b_a^1}$ and $W_3 \not\downarrow \overline{b_a^1}$). It is immediately clear that the agent Ag_a^1 is involved in all the reductions of $T|R$; hence, the first reduction step must consist of the consumption of one tuple $\langle a \rangle$ performed by the prefix $in(a)$ of Ag_a^1. The consumed tuple $\langle a \rangle$ is present in T because R does not contain any tuple. Hence, $T \xrightarrow{\overline{a}} T'$ and $W_3 \equiv T'|R$ because the second and the third reduction steps consist of the generation and the withdrawal of the presence token $\langle b_a^1 \rangle$, respectively. Observe that $S'|R \overset{\bullet}{\sim} T'|R$ and $fn(S'), fn(T') \subseteq L$, then also $(S', T') \in \mathcal{R}$.
- $S \xrightarrow{a} S'$:
  It is not restrictive to suppose $S \equiv S_1 | \prod_n \langle a \rangle$ (with $n \geq 0$) where $S_1 \not\downarrow \overline{a}$. It is easy to see that $S_1 \xrightarrow{a} S_1'$ because the term $\prod_n \langle a \rangle$ can not infer such a derivation. Moreover, $S' \equiv S_1' | \prod_n \langle a \rangle$.

  We first consider a sequence of reduction steps that renames the n tuples $\langle a \rangle$ appearing in $S|R$ in tuples $\langle c_a \rangle$ with c_a fresh. The renaming is performed by the agent Ag_a^2. Let R' be the term:

$$\prod_{l \in L} Ag_l^1 | \prod_{l \in L \setminus \{a\}} Ag_l^2 | \prod_{l \in L} Ag_l^3 | \prod_{l \in L} Ag_l^4 | \prod_{l \in L} Ag_l^5$$

then:

$$\begin{array}{rll} S|R & \longrightarrow S_1 | \coprod_{n-1} \langle a \rangle | out(b_a^2).out(c_a).in(b_a^2).Ag_a^2|R' & (\stackrel{def}{=} V_1) \\ & \longrightarrow S_1 | \coprod_{n-1} \langle a \rangle | \langle b_a^2 \rangle | out(c_a).in(b_a^2).Ag_a^2|R' & (\stackrel{def}{=} V_2) \\ & \longrightarrow S_1 | \coprod_{n-1} \langle a \rangle | \langle b_a^2 \rangle | \langle c_a \rangle | in(b_a^2).Ag_a^2|R' & (\stackrel{def}{=} V_3) \\ & \longrightarrow S_1 | \coprod_{n-1} \langle a \rangle | \langle c_a \rangle | R & (\stackrel{def}{=} V_4) \\ & \ldots & \\ & \longrightarrow S_1 | \coprod_{n} \langle c_a \rangle | R & (\stackrel{def}{=} V_{4n}) \end{array}$$

Observe that $V_{4n} \not\downarrow \overline{a}$ and that alternatively, every two steps, the presence token $\langle b_a^2 \rangle$ is generated and consumed. The agent $T|R$ is barbed bisimilar to $S|R$, hence:

$$T|R \longrightarrow W_1 \longrightarrow \ldots \longrightarrow W_{4n}$$

where $V_i \stackrel{\bullet}{\sim} W_i$. Also in the sequence of reductions performed by $T|R$, the presence token $\langle b_l^2 \rangle$ must be alternatively generated and consumed, every two steps, for n times. Only the agent Ag_a^2 is able to do this, then it is involved in all the $4n$ steps, its guard $in(a)$ is performed n times, and n tuples $\langle c_a \rangle$ are created. This requires $T \xrightarrow{\overline{a}} T^1 \xrightarrow{\overline{a}} \ldots \xrightarrow{\overline{a}} T^n$. Let $T_1 \stackrel{def}{=} T^n$, by Proposition 3.7 we have $T \equiv T_1 | \coprod_n \langle a \rangle$. Hence $W_{4n} \equiv T_1 | \coprod_n \langle c_a \rangle | R$ where $W_{4n} \not\downarrow \overline{a}$.

The agent V_{4n} is now able to generate a new tuple $\langle a \rangle$ (because of its subagent Ag_a^3) which is then consumed by S_1 performing the derivation $S_1 \xrightarrow{a} S_1'$. Let R'' be the term:

$$\prod_{l \in L} Ag_l^1 | \prod_{l \in L} Ag_l^2 | \prod_{l \in L \setminus \{a\}} Ag_l^3 | \prod_{l \in L} Ag_l^4 | \prod_{l \in L} Ag_l^5$$

then:

$$\begin{array}{rll} V_{4n} & \longrightarrow S_1 | \coprod_{n} \langle c_a \rangle | \langle a \rangle | out(b_a^3).in(b_a^3).Ag_a^3|R'' & (\stackrel{def}{=} V_{4n+1}) \\ & \longrightarrow S_1 | \coprod_{n} \langle c_a \rangle | \langle a \rangle | \langle b_a^3 \rangle | in(b_a^3).Ag_a^3|R'' & (\stackrel{def}{=} V_{4n+2}) \\ & \longrightarrow S_1' | \coprod_{n} \langle c_a \rangle | \langle b_a^3 \rangle | in(b_a^3).Ag_a^3|R'' & (\stackrel{def}{=} V_{4n+3}) \\ & \longrightarrow S_1' | \coprod_{n} \langle c_a \rangle | R & (\stackrel{def}{=} V_{4n+4}) \end{array}$$

Observe that $V_{4n+2} \downarrow \overline{a}$ while $V_{4n+3} \not\downarrow \overline{a}$. This follows from Proposition 3.5 because $S_1 \not\downarrow \overline{a}$ and $S_1 \xrightarrow{a} S_1'$. Also W_{4n} must offer equivalent reduction steps:

$$W_{4n} \longrightarrow W_{4n+1} \longrightarrow \ldots \longrightarrow W_{4n+4}$$

where $V_i \stackrel{\bullet}{\sim} W_i$. The fact that the presence token $\langle b_a^3 \rangle$ appears after two reduction steps implies that the first two steps are performed by Ag_a^3. Moreover, $W_{4n+3} \not\downarrow \overline{a}$ implies that the tuple $\langle a \rangle$, generated by Ag_a^3, must be consumed during the following reduction step. The consumption can be performed by the agent T_1 or by the context. In the second case, one of the agents Ag_a^i with $i = 1, 2$ that performs the operation $in(a)$ is involved. This implies the contradiction $V_{4n+3} \stackrel{\bullet}{\not\sim} W_{4n+3}$ because (see the observations about the presence tokens at the beginning of this proof) $W_{4n+3} \longrightarrow W \downarrow \overline{b_a^i}$, while V_{4n+3} requires at least two steps in order to generate a new presence token $\langle b_a^i \rangle$. Then, the tuple $\langle a \rangle$ is removed by T_1 by means of the derivation $T_1 \xrightarrow{a} T_1'$. We can conclude that $W_{4n+4} \equiv T_1' | \prod_n \langle c_a \rangle | R$.

The n tuples $\langle c_a \rangle$ in V_{4n+4} can be now renamed in $\langle a \rangle$ by the agent Ag_a^4. Let R''' be the term:

$$\prod_{l \in L} Ag_l^1 | \prod_{l \in L} Ag_l^2 | \prod_{l \in L} Ag_l^3 | \prod_{l \in L \setminus \{a\}} Ag_l^4 | \prod_{l \in L} Ag_l^5$$

then:

$$\begin{array}{lll}
V_{4n+4} & \longrightarrow \; S_1' | \prod_{n-1} \langle c_a \rangle | out(b_a^4).out(a).in(b_a^4).Ag_a^4 | R''' & (\stackrel{def}{=} V_{4n+5}) \\
& \longrightarrow \; S_1' | \prod_{n-1} \langle c_a \rangle | \langle b_a^4 \rangle | out(a).in(b_a^4).Ag_a^4 | R''' & (\stackrel{def}{=} V_{4n+6}) \\
& \longrightarrow \; S_1' | \prod_{n-1} \langle c_a \rangle | \langle b_a^4 \rangle | \langle a \rangle | in(b_a^4).Ag_a^4 | R''' & (\stackrel{def}{=} V_{4n+7}) \\
& \longrightarrow \; S_1' | \prod_{n-1} \langle c_a \rangle | \langle a \rangle | R & (\stackrel{def}{=} V_{4n+8}) \\
& \ldots & \\
& \longrightarrow \; S_1' | \prod_n \langle a \rangle | R & (\stackrel{def}{=} V_{8n+4})
\end{array}$$

Observe that alternatively, every two steps, the presence token $\langle b_a^4 \rangle$ is generated and consumed. The agent W_{4n+4} is barbed bisimilar to V_{4n+4}, hence:

$$W_{4n+4} \longrightarrow W_{4n+5} \longrightarrow \ldots \longrightarrow W_{8n+4}$$

where $V_i \stackrel{\bullet}{\sim} W_i$. Also in the sequence of reductions performed by W_{4n+4}, the presence token $\langle b_l^4 \rangle$ must be alternatively generated and consumed, every two steps, for n times. Only the agent Ag_a^4 is able to do this, then it is activated for n times. Hence, $W_{8n+4} \equiv T_1' | \prod_n \langle a \rangle | R$. This ensures $S_1' | \prod_n \langle a \rangle | R \stackrel{\bullet}{\sim} T_1' | \prod_n \langle a \rangle | R$, hence also $(S_1' | \prod_n \langle a \rangle, T_1' | \prod_n \langle a \rangle) \in \mathcal{R}$. We have already observed that $S' \equiv S_1' | \prod_n \langle a \rangle$. Moreover, $T_1 \xrightarrow{a} T_1'$ ensures $T \xrightarrow{a} T_1' | \prod_n \langle a \rangle$.

- $S \xrightarrow{\tau} S'$:
Let $L' \stackrel{def}{=} \{a \mid T \xrightarrow{\neg a} T'\}$ be a subset of L and n be the cardinality of L'. The

set L' is used to avoid that the derivation $S \stackrel{\tau}{\longrightarrow} S'$ is matched by T with a reduction step with a label different from τ: we introduce in TS all the tuples $\langle a \rangle$, for each $a \in L'$, thus disallowing T to perform $\neg a$ derivations.

By Proposition 3.6, we have that $T \not\downarrow \overline{a}$ for each $a \in L'$.

The agent $S|R$ can execute the following sequence of reduction steps in which the agents $Ag^5_{l'}$ (with $l' \in L'$) generate the tuples $\langle l' \rangle$. Let $l_1 \in L'$ and let R' be the term:

$$\prod_{l \in L} Ag^1_l | \prod_{l \in L} Ag^2_l | \prod_{l \in L} Ag^3_l | \prod_{l \in L} Ag^4_l | \prod_{l \in L \setminus \{l_1\}} Ag^5_l$$

and R'' the agent:

$$\prod_{l \in L} Ag^1_l | \prod_{l \in L} Ag^2_l | \prod_{l \in L} Ag^3_l | \prod_{l \in L} Ag^4_l | \prod_{l \in L \setminus L'} Ag^5_l$$

then:

$$\begin{array}{rll}
S|R & \longrightarrow S|\langle l_1 \rangle | out(b^5_{l_1}).in(l_1).in(b^5_{l_1}).Ag^5_{l_1}|R' & (\stackrel{def}{=} V_1) \\
 & \longrightarrow S|\langle l_1 \rangle | \langle b^5_{l_1} \rangle | in(l_1).in(b^5_{l_1}).Ag^5_{l_1}|R' & (\stackrel{def}{=} V_2) \\
 & \ldots & \\
 & \longrightarrow S| \prod_{l' \in L'} \langle l' \rangle | \prod_{l' \in L'} \langle b^5_{l'} \rangle | \prod_{l' \in L'} in(l').in(b^5_{l'}).Ag^5_{l'}|R'' & (\stackrel{def}{=} V_{2n})
\end{array}$$

The barbed bisimilar agent $T|R$ must allow the sequence of reductions:

$$T|R \longrightarrow W_1 \longrightarrow \ldots \longrightarrow W_{2n}$$

where $V_i \stackrel{\bullet}{\sim} W_i$ and $W_{2n} \downarrow \overline{l'}$ for every $l' \in L'$. In order to generate the presence tokens $\langle b^5_{l'} \rangle$ the agents $Ag^5_{l'}$ must be involved, then $W_{2n} \equiv T| \prod_{l' \in L'} \langle l' \rangle | \prod_{l' \in L'} \langle b^5_{l'} \rangle | \prod_{l' \in L'} in(l').in(b^5_{l'}).Ag^5_{l'}|R''$.

The agent V_{2n} is now able to perform the reduction due to $S \stackrel{\tau}{\longrightarrow} S'$:

$$V_{2n} \longrightarrow S'| \prod_{l' \in L'} \langle l' \rangle | \prod_{l' \in L'} \langle b^5_{l'} \rangle | \prod_{l' \in L'} in(l').in(b^5_{l'}).Ag^5_{l'}|R'' \qquad (\stackrel{def}{=} V_{2n+1})$$

The agent W_{2n} must allow the reduction step $W_{2n} \longrightarrow W_{2n+1}$ with $V_i \stackrel{\bullet}{\sim} W_i$. The usual observations on the presence tokens allow us to state that the subagent R'' cannot be involved in this reduction step as well as the agents $in(l').in(b^5_{l'}).Ag^5_{l'}$ because as first step they remove one of the tuples $\langle l' \rangle$ (observe that $W_{2n+1} \downarrow \overline{l'}$ while $T \not\downarrow \overline{l'}$).

Thus, the step must be inferred by the agent T, but it cannot be neither an output nor an input step, because no agent in the environment can synchronize with one of these steps. Also a derivation labeled with $\neg a$ cannot be performed because of the presence in the environment of the tuple $\langle a \rangle$

(remember that by definition of L', $T \xrightarrow{\neg a} T'$ implies $a \in L'$). Hence, the derivation must be labeled with τ and:

$$W_{2n+1} \equiv T' | \prod_{l' \in L'} \langle l' \rangle | \prod_{l' \in L'} \langle b^5_{l'} \rangle | \prod_{l' \in L'} in(l').in(b^5_{l'}).Ag^5_{l'} | R''$$

with $T \xrightarrow{\tau} T'$.

The tuples $\langle l' \rangle$ and the presence tokens $\langle b^5_{l'} \rangle$ can be now removed. Let R''' be the term:

$$\prod_{l' \in L' \setminus \{l_1\}} in(l').in(b^5_{l'}).Ag^5_{l'} | R''$$

then:

$$\begin{aligned} V_{2n+1} &\longrightarrow S' | \textstyle\prod_{l' \in L' \setminus \{l_1\}} \langle l' \rangle | \prod_{l' \in L'} \langle b^5_{l'} \rangle | in(b^5_{l_1}).Ag^5_{l_1} | R''' && (\stackrel{def}{=} V_{2n+2}) \\ &\longrightarrow S' | \textstyle\prod_{l' \in L' \setminus \{l_1\}} \langle l' \rangle | \prod_{l' \in L' \setminus \{l_1\}} \langle b^5_{l'} \rangle | Ag^5_{l_1} | R''' && (\stackrel{def}{=} V_{2n+3}) \\ &\dots \\ &\longrightarrow S' | R && (\stackrel{def}{=} V_{4n+1}) \end{aligned}$$

Because of $V_{2n+1} \stackrel{\bullet}{\sim} W_{2n+1}$, then also:

$$W_{2n+1} \longrightarrow W_{2n+2} \longrightarrow \dots \longrightarrow W_{4n+1}$$

where $V_i \stackrel{\bullet}{\sim} W_i$. The consumption of the presence tokens in $2n$ steps ensures that the agent T' is not involved in no one of them. Then $W_{4n+1} \equiv T' | R$. Finally, observe that $S' | R \stackrel{\bullet}{\sim} T' | R$ ensures $(S', T') \in \mathcal{R}$.

- $S \xrightarrow{\neg a} S'$:
  The proof is the same as the previous case with the only difference that the set L' is defined as $L' \stackrel{def}{=} \{b \mid T \xrightarrow{\neg b} T' \textit{ and } b \neq a\}$. In this case the message name a is not an element of L' because T can simulate the step of S also with a derivation labeled with $\neg a$. In fact, with this new definition of L', the possible derivations of T are $T \xrightarrow{\tau} T'$ or $T \xrightarrow{\neg a} T'$, where $(S', T') \in \mathcal{R}$. □

Theorem 3.13 Let P and Q be agents interpreted under the *unordered* semantics. If $P|R \stackrel{\bullet}{\sim} Q|R$ for every agent R, then $P \sim Q$.

Proof Let P and Q be two agents satisfying the premises of the theorem, let $L = fn(P) \cup fn(Q)$; observe that L is finite. Consider the term R defined as in the proof of Theorem 3.9.

We show that the relation:

$$\mathcal{R} = \{(S, T) \mid S|R \stackrel{\bullet}{\sim} T|R \textit{ and } fn(S), fn(T) \subseteq L\}$$

is a $\neg$-bisimulation (up to $\equiv$), hence proving that $P \sim Q$ because $(P, Q) \in \mathcal{R}$.

The main differences with the proof of the ordered case are due to the fact that whenever the context R emits a tuple a further τ step is required before the tuple is effectively rendered in TS.

For example, the observations on the presence tokens must be adapted. Each agent Ag_l^i (and only it) is able to generate and consume the corresponding presence token $\langle \underline{b_l^i} \rangle$; moreover, for every agent Ag_l^i, if $Ag_l^i \xrightarrow{\alpha} R'$, then $R' \longrightarrow R'' \longrightarrow R''' \downarrow \overline{b_l^i}$, i.e., if one of the subagents of R performs a transition step, then the corresponding presence token can be produced after *two* reduction steps.

In order to prove that $\mathcal{R}$ is a $\neg$-bisimulation we observe that $\mathcal{R}$ is symmetric and we proceed by case analysis on the possible derivations $S \xrightarrow{\alpha} S'$.

We consider only the case $S \xrightarrow{a} S'$ because all the other cases are easy adaptations to the *unordered* semantics of the corresponding cases treated in the proof of Theorem 3.9.

- $S \xrightarrow{a} S'$:
  It is not restrictive to suppose $S \equiv S_1 | \prod_n \langle a \rangle$ (with $n \geq 0$) where $S_1 \not\downarrow \overline{a}$. It is easy to see that $S_1 \xrightarrow{a} S_1'$ and $S' \equiv S_1' | \prod_n \langle a \rangle$.

  We consider the following sequence of reduction steps divided in five phases. We use the terms R', R'', and R''' as defined in the analysis of the case $S \xrightarrow{a} S'$ of the proof of Theorem 3.9.

  In the first phase each of the n tuples $\langle a \rangle$ appearing in $S|R$ is replaced by a new term $\langle\langle c_a \rangle\rangle$:

$$
\begin{array}{rll}
S|R & \longrightarrow S_1 | \prod_{n-1} \langle a \rangle | out(b_a^2).out(c_a).in(b_a^2).Ag_a^2 | R' & (\stackrel{def}{=} V_1) \\
& \longrightarrow S_1 | \prod_{n-1} \langle a \rangle | \langle\langle b_a^2 \rangle\rangle | out(c_a).in(b_a^2).Ag_a^2 | R' & (\stackrel{def}{=} V_2) \\
& \longrightarrow S_1 | \prod_{n-1} \langle a \rangle | \langle b_a^2 \rangle | out(c_a).in(b_a^2).Ag_a^2 | R' & (\stackrel{def}{=} V_3) \\
& \longrightarrow S_1 | \prod_{n-1} \langle a \rangle | \langle b_a^2 \rangle | \langle\langle c_a \rangle\rangle | in(b_a^2).Ag_a^2 | R' & (\stackrel{def}{=} V_4) \\
& \longrightarrow S_1 | \prod_{n-1} \langle a \rangle | \langle\langle c_a \rangle\rangle | R & (\stackrel{def}{=} V_5) \\
& \ldots & \\
& \longrightarrow S_1 | \prod_{n} \langle\langle c_a \rangle\rangle | R & (\stackrel{def}{=} V_{5n})
\end{array}
$$

  All $5n$ reduction steps of this first phase are performed by the subagent Ag_a^2. At the end of this phase, no more tuples $\langle a \rangle$ are present, i.e., $V_{5n} \not\downarrow \overline{a}$.

In the second phase a term $\langle\langle a\rangle\rangle$ is emitted:

$$\begin{array}{rll}
V_{5n} & \longrightarrow S_1|\,\Pi_n\langle\langle c_a\rangle\rangle|\langle\langle a\rangle\rangle|out(b_a^3).in(b_a^3).Ag_a^3|R'' & (\stackrel{def}{=} V_{5n+1}) \\
& \longrightarrow S_1|\,\Pi_n\langle\langle c_a\rangle\rangle|\langle\langle a\rangle\rangle|\langle\langle b_a^3\rangle\rangle|in(b_a^3).Ag_a^3|R'' & (\stackrel{def}{=} V_{5n+2}) \\
& \longrightarrow S_1|\,\Pi_n\langle\langle c_a\rangle\rangle|\langle\langle a\rangle\rangle|\langle b_a^3\rangle|in(b_a^3).Ag_a^3|R'' & (\stackrel{def}{=} V_{5n+3}) \\
& \longrightarrow S_1|\,\Pi_n\langle\langle c_a\rangle\rangle|\langle\langle a\rangle\rangle|R & (\stackrel{def}{=} V_{5n+4})
\end{array}$$

This phase requires four reduction steps, all performed by the subagent Ag_a^3.

The third phase is the one in which the term S_1 is involved: the tuple $\langle a\rangle$ is rendered and then consumed:

$$\begin{array}{rll}
V_{5n+4} & \longrightarrow S_1|\,\Pi_n\langle\langle c_a\rangle\rangle|\langle a\rangle|R & (\stackrel{def}{=} V_{5n+5}) \\
& \longrightarrow S_1'|\,\Pi_n\langle\langle c_a\rangle\rangle|R & (\stackrel{def}{=} V_{5n+6})
\end{array}$$

In these two steps only the terms S_1 and $\langle\langle a\rangle\rangle$ are involved.

In the fourth phase each of the n terms $\langle\langle c_a\rangle\rangle$ is replaced by a term $\langle\langle a\rangle\rangle$:

$$\begin{array}{rll}
V_{5n+6} & \longrightarrow S_1'|\,\Pi_{n-1}\langle\langle c_a\rangle\rangle|\langle c_a\rangle|R & (\stackrel{def}{=} V_{5n+7}) \\
& \longrightarrow S_1'|\,\Pi_{n-1}\langle\langle c_a\rangle\rangle|out(b_a^4).out(a).in(b_a^4).Ag_a^4|R''' & (\stackrel{def}{=} V_{5n+8}) \\
& \longrightarrow S_1'|\,\Pi_{n-1}\langle\langle c_a\rangle\rangle|\langle\langle b_a^4\rangle\rangle|out(a).in(b_a^4).Ag_a^4|R''' & (\stackrel{def}{=} V_{5n+9}) \\
& \longrightarrow S_1'|\,\Pi_{n-1}\langle\langle c_a\rangle\rangle|\langle b_a^4\rangle|out(a).in(b_a^4).Ag_a^4|R''' & (\stackrel{def}{=} V_{5n+10}) \\
& \longrightarrow S_1'|\,\Pi_{n-1}\langle\langle c_a\rangle\rangle|\langle b_a^4\rangle|\langle\langle a\rangle\rangle|in(b_a^4).Ag_a^4|R''' & (\stackrel{def}{=} V_{5n+11}) \\
& \longrightarrow S_1'|\,\Pi_{n-1}\langle\langle c_a\rangle\rangle|\langle\langle a\rangle\rangle|R & (\stackrel{def}{=} V_{5n+12}) \\
& \ldots & \\
& \longrightarrow S_1'|\,\Pi_n\langle\langle a\rangle\rangle|R & (\stackrel{def}{=} V_{11n+6})
\end{array}$$

In this phase only the agent Ag_a^4 and the n terms $\langle\langle c_a\rangle\rangle$ are involved.

During the last phase all the n terms $\langle\langle a\rangle\rangle$ are reduced to $\langle a\rangle$:

$$\begin{array}{rll}
V_{11n+6} & \longrightarrow S_1'|\,\Pi_n\langle\langle a\rangle\rangle|R & (\stackrel{def}{=} V_{11n+7}) \\
& \longrightarrow S_1'|\,\Pi_{n-1}\langle\langle a\rangle\rangle|\langle a\rangle|R & (\stackrel{def}{=} V_{11n+8}) \\
& \ldots & \\
& \longrightarrow S_1'|\,\Pi_n\langle a\rangle|R & (\stackrel{def}{=} V_{12n+6})
\end{array}$$

The fact that $T|R$ is barbed bisimilar to $S|R$ ensures that:

$$T|R \longrightarrow W_1 \longrightarrow \ldots \longrightarrow W_{12n+6}$$

where $V_i \stackrel{\bullet}{\sim} W_i$.

The first two phases can be treated in the same way we proceeded in the proof of Theorem 3.9. Indeed:

$$\begin{aligned} W_{5n} &\equiv T_1 | \prod_n \langle\langle c_a \rangle\rangle | R \\ W_{5n+4} &\equiv T_1 | \prod_n \langle\langle c_a \rangle\rangle | \langle\langle a \rangle\rangle | R \end{aligned}$$

with $T \equiv T_1 | \prod_n \langle a \rangle$ and $T_1 \not\downarrow \overline{a}$. Observe that the first phase can be used to prove that in general, given two terms Q_1 and Q_2 such that $Q_1 | R \stackrel{\bullet}{\sim} Q_2 | R$ and $Q_1 \equiv Q'_1 | \prod_n \langle a \rangle$ with $Q'_1 \not\downarrow \overline{a}$, then also $Q_2 \equiv Q'_2 | \prod_n \langle a \rangle$ with $Q'_2 \not\downarrow \overline{a}$. This result will be used in the following.

During the third phase the context R is not involved (remember the observations about the presence tokens) as well as the n terms $\langle\langle c_a \rangle\rangle$. Hence the two reductions are performed by the term $T_1 | \langle\langle a \rangle\rangle$:

$$T_1 | \langle\langle a \rangle\rangle \longrightarrow T''_1 \longrightarrow T'_1$$

with $W_{5n+6} \equiv T'_1 | \prod_n \langle\langle c_a \rangle\rangle | R$. The fact that both $T_1 | \langle\langle a \rangle\rangle \not\downarrow \overline{a}$ and $T'_1 \not\downarrow \overline{a}$ while $T''_1 \downarrow \overline{a}$, permits to apply Fact 3.11. Thus, $T_1 \stackrel{a}{\longrightarrow} T'_1$. By Proposition 3.7 this ensures that also $T \stackrel{a}{\longrightarrow} T'_1 | \prod_n \langle a \rangle$.

The fourth phase can be treated as the first two. In fact, the context R and the n terms $\langle\langle c_a \rangle\rangle$ are surely involved in all the reduction steps. In this way it is easy to see that:

$$W_{11n+6} \quad \equiv \quad T'_1 | \prod_n \langle\langle a \rangle\rangle | R$$

During the last phase the context R is not involved, then:

$$T'_1 | \prod_n \langle\langle a \rangle\rangle \underbrace{\longrightarrow \ldots \longrightarrow}_{n \text{ times}} T'$$

such that $S' | R \stackrel{\bullet}{\sim} T' | R$. This ensures $(S', T') \in \mathcal{R}$.

Remember that $S' \equiv S_1 | \prod_n \langle a \rangle$ with $S_1 \not\downarrow \overline{a}$. The observations we made during the analysis of the first phase allows us to state that also $T' \equiv T''' | \prod_n \langle a \rangle$. Hence:

$$T'_1 | \prod_n \langle\langle a \rangle\rangle \underbrace{\longrightarrow \ldots \longrightarrow}_{n \text{ times}} T''' | \prod_n \langle a \rangle$$

By Fact 3.12 then $T'_1 \equiv T''$ hence, by substitutivity of $\equiv$, $T' \equiv T'_1 | \prod_n \langle a \rangle$. In the analysis of the third phase we proved that $T \stackrel{a}{\longrightarrow} T'_1 | \prod_n \langle a \rangle$, then also $T \stackrel{a}{\longrightarrow} T'$. $\square$

Theorem 3.17 Let P and Q be agents interpreted under the *instantaneous* semantics. If $P | R \stackrel{\bullet}{\sim} Q | R$ for every agent R, then $P \sim_a Q$.

Proof Let P and Q be two agents satisfying the premises of the theorem. Let $L = fn(P) \cup fn(Q)$; observe that L is finite.

We show that the pair (P, Q) is contained in an asynchronous $\neg$-bisimulation (up to $\equiv$), hence $P \sim_a Q$. In particular, we define the following agent R:

$$R \stackrel{def}{=} \prod_{l \in L} Ag_l^1 | \prod_{l \in L} Ag_l^2$$

such that the relation:

$$\mathcal{R} = \{(S, T) \mid S|R \stackrel{\bullet}{\sim} T|R \ and \ fn(S), fn(T) \subseteq L\}$$

is an asynchronous $\neg$-bisimulation (up to $\equiv$). The pair (P, Q) is in $\mathcal{R}$ because $P|R$ is barbed bisimilar to $Q|R$ and both $fn(P)$ and $fn(Q)$ are subsets of L.

The agents Ag_l^i are defined as follows:

$$\begin{aligned} Ag_l^1 &\stackrel{def}{=} rec\, X.in(l).out(b_l^1).in(b_l^1).X \\ Ag_l^2 &\stackrel{def}{=} rec\, X.out(b_l^2).in(b_l^2).out(l).out(b_l^3).in(b_l^3).X \end{aligned}$$

where b_l^i and c_l are all fresh and distinct names for every i and l. Observe that the agents Ag_l^1 are defined as in the proof of Theorem 3.9.

The observation on the *presence tokens* must be changed not only because we changed the semantics, but also because the agents Ag_l^2 are defined in a different way. In the proof for the *ordered* case, the presence tokens ensured the possible detection of each step of the context R because a presence token can be rendered after a reduction step. Instead, the steps of the new agent R are immediately detected. If Ag_l^1 moves, then the presence token b_l^1 instantaneously appears. Otherwise, if Ag_l^2 is involved in some computation, the initially available presence token b_l^2 is consumed.

As for the *ordered* and the *unordered* semantics, we first observe that $\mathcal{R}$ is symmetric and then we proceed by case analysis on the possible derivations $S \xrightarrow{\alpha} S'$.

We omit the analysis of the case $S \xrightarrow{\overline{a}} S'$ because it is treated in the same way as in the proof for the *ordered* semantics. That analysis can be adapted even if the agent R is different because it considers only the agents Ag_l^1 that, as we have already observed, are defined in the same way as in the proof of Theorem 3.9.

- $S \xrightarrow{a} S'$:

Let $L' \stackrel{def}{=} \{b \mid T \stackrel{\neg b}{\longrightarrow} T''$ with $S'|R \stackrel{\bullet}{\sim} T''|\langle a\rangle|R$ *and* $b \neq a\}$ be a subset of L and n be the cardinality of L'. The set L' is used to avoid that the derivation $S \stackrel{\tau}{\longrightarrow} S'$ is matched by T with a reduction step with a label $\neg b$ for some b. For this reason, we consider a computation in which all the tuples $\langle b\rangle$, for each $b \in L'$, are produced.

The agent $S|R$ can execute the following sequence of reduction steps in which the agents $Ag^2_{l'}$ (with $l' \in L'$) generate the tuples $\langle l'\rangle$. Let $l_1 \in L'$, then :

$$
\begin{array}{rll}
S|R & \longrightarrow S|\langle l_1\rangle|\langle b^3_{l_1}\rangle|in(b^3_{l_1}).Ag^2_{l_1}|\prod_{l\in L} Ag^1_l|\prod_{l\in L\setminus\{l_1\}} Ag^2_l & (\stackrel{def}{=} V_1) \\
& \ldots & \\
& \longrightarrow S|\prod_{l'\in L'}\langle l'\rangle|\prod_{l'\in L'}\langle b^3_{l'}\rangle|\prod_{l'\in L'} in(b^3_{l'}).Ag^2_{l'} & \\
& \quad |\prod_{l\in L} Ag^1_l|\prod_{l\in L\setminus L'} Ag^2_l & (\stackrel{def}{=} V_n) \\
& \longrightarrow S|\prod_{l'\in L'}\langle l'\rangle|\prod_{l'\in L'\setminus\{l_1\}}\langle b^3_{l'}\rangle|\prod_{l'\in L'\setminus\{l_1\}} in(b^3_{l'}).Ag^2_{l'} & \\
& \quad |\prod_{l\in L} Ag^1_l|\prod_{l\in L\setminus L'\cup\{l_1\}} Ag^2_l & (\stackrel{def}{=} V_{n+1}) \\
& \ldots & \\
& \longrightarrow S|\prod_{l'\in L'}\langle l'\rangle|R & (\stackrel{def}{=} V_{2n})
\end{array}
$$

The barbed bisimilar agent $T|R$ must allow the sequence of reductions:

$$T|R \longrightarrow W_1 \longrightarrow \ldots \longrightarrow W_{2n}$$

where $V_i \stackrel{\bullet}{\sim} W_i$; hence $W_n \downarrow \overline{b^3_{l'}}$ and $W_{2n} \not\downarrow \overline{b^3_{l'}}$ for every $l' \in L'$. In order to produce all the presence tokens $\langle b^3_{l'}\rangle$ and then consume them, the agents $Ag^2_{l'}$ must be involved in all the $2n$ reduction steps. Hence, $W_{2n} \equiv T|\prod_{l'\in L'}\langle l'\rangle|R$.

In the second phase the term Ag^2_a produce the tuple $\langle a\rangle$ that is consumed by S by means of the derivation $S \stackrel{a}{\longrightarrow} S'$:

$$
\begin{array}{rll}
V_{2n} & \longrightarrow S|\langle a\rangle|\langle b^3_a\rangle|in(b^3_a).Ag^2_a|\prod_{l'\in L'}\langle l'\rangle & \\
& \quad |\prod_{l\in L} Ag^1_l|\prod_{l\in L\setminus\{a\}} Ag^2_l & (\stackrel{def}{=} V_{2n+1}) \\
& \longrightarrow S|\langle a\rangle|\prod_{l'\in L'}\langle l'\rangle|R & (\stackrel{def}{=} V_{2n+2}) \\
& \longrightarrow S'|\prod_{l'\in L'}\langle l'\rangle|R & (\stackrel{def}{=} V_{2n+3})
\end{array}
$$

The agent W_{2n} must allow the reductions:

$$W_{2n} \longrightarrow W_{2n+1} \longrightarrow W_{2n+2} \longrightarrow W_{2n+3}$$

with $V_i \stackrel{\bullet}{\sim} W_i$. The first two steps are surely induced by the agent Ag^2_a because the presence token b^3_a is rendered in the first step and consumed in the second one. This ensures that $W_{2n+2} \equiv T|\langle a\rangle|\prod_{l'\in L'}\langle l'\rangle|R$.

The third step is the crucial one. The term R cannot be involved because it surely changes the available presence tokens. Thus the term T is involved. We proceed by case analysis on the possible ways the agent T can infer the reduction step.

· $T \xrightarrow{a} T'$ or $T \xrightarrow{\tau} T'$:
In this case $W_{2n+3} \equiv Z | \prod_{l' \in L'} \langle l' \rangle | R$ with $Z \equiv T'$ in the first case or $Z \equiv T' | \langle a \rangle$ in the second one.

All the tuples $\langle l' \rangle$ for each $l' \in L'$ present in V_{2n+3} can be now consumed by the corresponding agents $Ag^1_{l'}$:

$$
\begin{array}{rll}
V_{2n+3} & \longrightarrow & S' | \prod_{l' \in L' \setminus \{l_1\}} \langle l' \rangle | \langle b^1_{l_1} \rangle | in(b^1_{l_1}).Ag^1_{l_1} \\
 & & | \prod_{l \in L \setminus \{l_1\}} Ag^1_l | \prod_{l \in L} Ag^2_l \qquad (\stackrel{def}{=} V_{2n+4}) \\
\ldots & & \\
 & \longrightarrow & S' | \prod_{l' \in L'} \langle b^1_{l'} \rangle | \prod_{l' \in L'} in(b^1_{l'}).Ag^1_{l'} \\
 & & | \prod_{l \in L \setminus L'} Ag^1_l | \prod_{l \in L} Ag^2_l \qquad (\stackrel{def}{=} V_{3n+3}) \\
 & \longrightarrow & S' | \prod_{l' \in L' \setminus \{l_1\}} \langle b^1_{l'} \rangle | \prod_{l' \in L' \setminus \{l_1\}} in(b^1_{l'}).Ag^1_{l'} \\
 & & | \prod_{l \in L \setminus L' \cup \{l_1\}} Ag^1_l | \prod_{l \in L} Ag^2_l \qquad (\stackrel{def}{=} V_{3n+4}) \\
\ldots & & \\
 & \longrightarrow & S' | R \qquad (\stackrel{def}{=} V_{4n+3})
\end{array}
$$

The barbed bisimilar agent W_{2n+3} must allow the sequence of reductions:

$$W_{2n+3} \longrightarrow W_{2n+4} \longrightarrow \ldots \longrightarrow W_{4n+3}$$

where $V_i \stackrel{\bullet}{\sim} W_i$; hence $W_{3n+3} \downarrow \overline{b^1_{l'}}$ and $W_{4n+3} \not\downarrow \overline{b^1_{l'}}$ for every $l' \in L'$. In order to produce all the presence tokens $\langle b^1_{l'} \rangle$ and then consume them, the agents $Ag^1_{l'}$ must be involved in all the $2n$ reduction steps. Hence, $W_{4n+3} \equiv Z|R$. This ensures that $S'|R \stackrel{\bullet}{\sim} Z|R$. Then, if $T \xrightarrow{a} T'$ it is $(S', T') \in \mathcal{R}$ otherwise if $T \xrightarrow{\tau} T'$ then $(S', T'|\langle a \rangle) \in \mathcal{R}$.

· $T \xrightarrow{\neg b} T'$ with $b \notin L' \cup \{a\}$:
In this case $W_{2n+3} \equiv T' | \langle a \rangle | \prod_{l' \in L'} \langle l' \rangle | R$. Proceeding in the same way as in the previous case it is possible to prove that $S'|R \stackrel{\bullet}{\sim} T'|\langle a \rangle|R$. By definition of L' this implies the contradiction $b \in L'$ or $b = a$. Hence, the reduction $W_{2n+2} \longrightarrow W_{2n+3}$ cannot be inferred by a derivation $T \xrightarrow{\neg b} T'$.

· $T \xrightarrow{b} T'$ with $b \in L'$:
In this case $W_{2n+3} \equiv T' | \langle a \rangle | \prod_{l' \in L' \setminus \{b\}} \langle l' \rangle | R$. Proceeding as in the first case, with the difference that the tuple $\langle b \rangle$ is not removed from the term $V_{2n+3} \equiv S' | \prod_{l' \in L'} \langle l' \rangle | R$, it is easy to prove that $S'|\langle b \rangle|R \stackrel{\bullet}{\sim} T'|\langle a \rangle|R$. Moreover, by definition of L', it is also true that $T \xrightarrow{\neg b} T''$ with $S'|R \stackrel{\bullet}{\sim} T''|\langle a \rangle|R$.

- $S \xrightarrow{\tau} S'$:

The proof is essentially the same as the previous case with two only differences. The first is that the definition of the set L' is changed:

$$L' \stackrel{def}{=} \{b \mid T \stackrel{\neg b}{\longrightarrow} T'' \text{ with } S'|R \stackrel{\bullet}{\sim} T''|R\}$$

The second one consists of changing the intermediary phase in which the term S is involved. That phase is reduced to a single step in which S performs its derivation $S \stackrel{\tau}{\longrightarrow} S'$. In this way the term T is forced to perform two kinds of derivation, either $T \stackrel{\tau}{\longrightarrow} T'$ or $T \stackrel{b}{\longrightarrow} T'$ with $b \in L'$. In the first case it is proved that $S'|R \stackrel{\bullet}{\sim} T'|R$. In the second case $S'|\langle b\rangle|R \stackrel{\bullet}{\sim} T'|R$ and, by definition of L', also $T \stackrel{\neg b}{\longrightarrow} T''$ such that $S'|R \stackrel{\bullet}{\sim} T''|R$.

- $S \stackrel{\neg a}{\longrightarrow} S'$:
  The proof is the same as the previous case with the only difference that the set L' is defined as:

$$L' \stackrel{def}{=} \{b \mid T \stackrel{\neg b}{\longrightarrow} T'' \text{ with } S'|R \stackrel{\bullet}{\sim} T''|R \textit{ and } b \neq a\}$$

In this case the message name a is not an element of L' because T can simulate the step of S also with a derivation labeled with $\neg a$.

Acknowledgements

We would like to thank the anonymous referees for their helpful comments.

References

[ACS98] R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. *Theoretical Computer Science*, 195(2):291-324, 1998.

[Bou92] G. Boudol. Asynchrony and the π-calculus. Technical Report 1702, INRIA, Sophia–Antipolis, 1992.

[BG95] N. Busi and R. Gorrieri. A Petri Net Semantics for π–calculus. In Proc. *CONCUR'95*, volume 962 of *LNCS*, pages 145–159, Springer Verlag, 1995.

[BGZ97a] N. Busi, R. Gorrieri, and G. Zavattaro. A Truly Concurrent view of Linda Interprocess Communication. Technical Report UBLCS-97-02, Department of Computer Science, University of Bologna, 1997.

[BGZ97b] N. Busi, R. Gorrieri, and G. Zavattaro. Three Semantics of the Output Operation for Generative Communication. In Proc. *Coordination'97*, volume 1282 of *LNCS*, 205–219, 1997.

[BGZ98] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2): 167–199, 1998.

[BGZ99] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, to appear.

[BP95] N. Busi and G.M. Pinna. A Causal Semantics for Contextual P/T nets. In Proc. *ICTCS'95*, 311–325, World Scientific, 1995.

[CGZ96] P. Ciancarini, R. Gorrieri, and G. Zavattaro. Towards a Calculus for Generative Communication. In Proc. *1st IFIP Conf. on FMOODS'96*, pages 283–297. Chapman & Hall, 1996.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GC92] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, 1992.

[Gro93] J.F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263-299, 1993.

[HKH95] M. Hansen, J. Kleist, and H. Hüttel. Bisimulations for Asynchronous Mobile Processes. Tech. Report, Basic Research in Computer Science, BRICS RS-96-8, 1996.

[HT91] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In Proc. *ECOOP '91*, volume 512 of *LNCS*, pages 133–147. Springer Verlag, 1991.

[JS98] Sun Microsystem, Inc. JavaSpaces Specification. Available at `http://java.sun.com/products/javaspaces/`, July 1998.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MS92] R. Milner and D. Sangiorgi. Barbed Bisimulation. In Proc. *ICALP'92*, volume 623 of *LNCS*, pages 685–695, Springer Verlag, 1992.

[Min67] M. L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, Englewood Cliffs, 1967.

[MR95] U. Montanari and F. Rossi. Contextual Nets. *Acta Informatica*, 32(6):545–596, 1995.

[Nar90] J. E. Narem. An Informal Operational Semantics of C-Linda V2.3.5. Technical Report YALEU/DCS/TR-839, Department of Computer Science, Yale University, 1990.

[SCA95] Scientific Computing Associates. *Linda: User's guide and reference manual.* Scientific Computing Associates, 1995.

[SS63] J. C. Shepherdson and J. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.

[WLLF98] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998. Available at `http://www.almaden.ibm.com/cs/TSpaces/`, IBM Corp. Almaden Research Center, IBM Corp.