A Process Algebraic View of Linda Coordination Primitives*

Nadia Busi[1] Roberto Gorrieri[2] Gianluigi Zavattaro[2]

[1] Dipartimento di Matematica, Università di Siena,
Via del Capitano 15, I-53100 Siena, Italy

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

e-mail: {busi,gorrieri,zavattar}@cs.unibo.it

Abstract

The main Linda coordination primitives (asynchronous communication, read operation, nonblocking in/rd predicates) are studied in a process algebraic setting. A lattice of eight languages is proposed, where its bottom element **L** is a process algebra differing from CCS only for the asynchrony of the output operation, while all the other languages in the lattice are obtained as extension of this basic language by adding some of the Linda coordination primitives. The observational semantics for these languages are all obtained as the coarsest congruences contained in the barbed semantics, where only tuples are observable. The lattice of the eight languages collapses to a smaller four-points lattice of different bisimulation-based semantics. Notably, for **L** this semantics is the standard notion of strong bisimulation, where inputs and outputs/tuples are treated symmetrically.

Keywords: Coordination languages, Semantics of Linda, Process algebra.

1 Introduction

The aim of this paper is to provide a process algebraic treatment to Linda [Gel85] coordination primitives, studying an adequate SOS operational semantics [Plo81] as well as an observational semantics based on the concept of strong barbed bisimulation [MS92]. To this aim, we first need to analyze the conceptual differencies w.r.t. traditional process algebras like CCS [Mil89], and then to propose a possible representation of these concepts in a process algebraic style.

The main features distinguishing Linda from classic process algebras are the following:

- *Asynchronous communication.* It is realized by means of a communication medium (called *Tuple Space*), that is always ready to receive messages/tuples from the senders and always ready to deliver them to the receivers. Hence, the communication is asynchronous because the sender may proceed just after performing $out(a)$ (emission of message a to the Tuple Space – TS for short). Similarly, the receiver can input a by performing $in(a)$ at any time after a is present in TS: a hand-shake synchronization between the tuple a in TS and the receiver completes the communication between the sender and the receiver, with the side-effect of removing message a from TS.

- *Read operation.* As messages/tuples are treated as resources, it is possible to read a message without removing it from TS (as the input operation does). Like the input operation, the reading of any message a – denoted by $rd(a)$ – is blocking: if the required message a is absent, then the reading process is suspended.

*Work supported by Esprit BRA Project Coordination n. 9102.

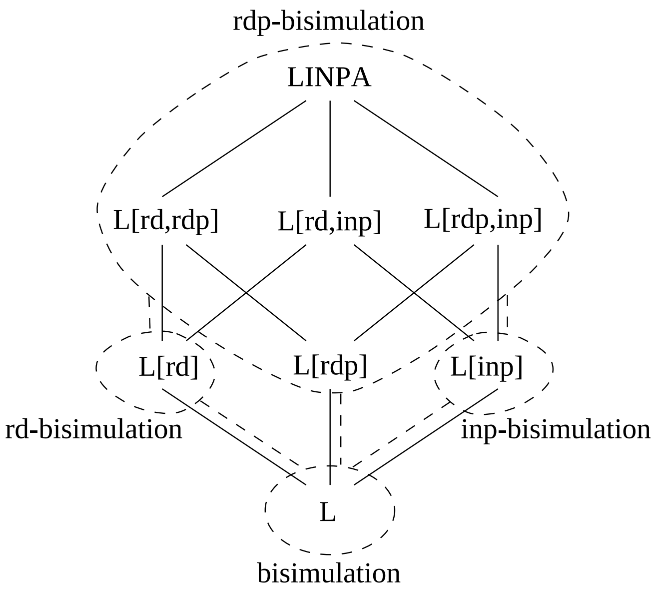

Figure 1: The lattice of the languages and their semantics.

- *Conditional input/read predicates.* The current status of TS is checked; if the required message is absent, the value $false$ is returned; on the contrary, if the message is found, their behavior is the same as the in/rd operation and the value $true$ is returned.[1]

This communication mechanism is said to be *generative* because a message generated by a process has an independent existence in TS until it is explicitly withdrawn by an input operation. In fact, after its insertion in TS, a message becomes equally accessible to all processes, and it is bound to none.

In implementing the peculiar features of Linda in a process algebra we have to take some design decisions; for instance, how to model asynchronous communication and the Tuple Space. Conceptually, the execution of the Linda 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 tuple a in the TS, we denote with $\langle a \rangle$). We decide to model the emission and the rendering of one message as forming together one single autonomous atomic action: $out(a).P$ becomes in one (internal) step the agent $\langle a \rangle | P$, where $|$ is the parallel composition operator. Hence, the representation of the Tuple Space is just given by all those tuples $\langle a \rangle$ that are in parallel with the processes.

A tuple $\langle a \rangle$ interacts with processes via hand-shake synchronization; it can be removed/consumed by each process willing to perform the input operation $in(a)$, but it can also be tested/read by means of the read primitive $rd(a)$. Concerning the conditional input/read predicates, we decide to reformulate them as operators $inp(a)?P\_Q$ and $rdp(a)?P\_Q$, respectively, that direct the flow of control to P or Q depending on the presence or absence of the required message a.

Most of the coordination primitives mentioned above are new in the world of process algebra. Hence, it is quite instructive to study each primitive in isolation first, and then their combination. Figure 1 shows a lattice of the eight languages we investigate; the bottom language $\mathbf{L}$ is essentially CCS,[2] where asynchronous communication is substituted for the synchronous hand-shake communication mechanism, by giving a semantics to the out operation as sketched above. The other seven languages are extensions of $\mathbf{L}$ obtained by adding some of the other Linda coordination primitives. The top language $\mathbf{L}[rd, inp, rdp]$ offers a process algebraic view of Linda and is called LINPA (LINda in Process Algebra).

In order to define an adequate observational semantics for the languages of the lattice, the approach we follow is inspired to [MS92]: firstly, identify the internal transitions, then define the observable actions,

[1] Linda offers also the $eval$ operation, that we do not consider because in the process algebraic approach we are going to study it would simply correspond to a spawn operation: $eval(P)$ calls for the activation of P.

[2] To be precise, we have omitted the relabeling operator and changed the names of the prefixes.

and finally find the coarsest congruence contained in the barbed semantics (a very coarse equivalence that equates processes that are bisimilar on internal actions and offer, at any pair of related states, the same observable actions).

Usually, internal transitions correspond to τ labeled derivations. In our case, we consider only τ labeled steps in those languages that does not contain predicates. For the languages with predicates we consider also a new label $\neg a$, introduced in order to give an appropriate structured operational semantics to the predicates. We show that the derivations labeled with $\neg a$ can be considered as internal as the usual τ steps are.

The second question is: what is an asynchronous observer able to detect? Following the intuition of [HT91], it is clear that what such an observer can see is just the TS: an observer can input (or read) any tuple that is present in TS, and so that has been previously emitted. On the contrary, the observer has no means for realizing if the system is performing input (as well as read) operations. Hence, there is a strong difference when considering synchronous or asynchronous communication: for the former, both inputs and outputs are relevant, while for the latter only the outputs (or better, only the tuples) are so. The barbed bisimulation that is obtained following the above observations corresponds to the one defined in [ACS96] in the setting of asynchronous π–calculus.

For language **L**, despite of the asymmetric role of inputs and tuples, the resulting congruence is the classic notion of bisimulation, where inputs and tuples are equally observable. This contrasts with [ACS96], because different is the operational modeling of the *out* operation (see the concluding section for a comparison).

For $\mathbf{L}[rd]$ the internal transitions are left untouched, but the resulting coarsest congruence, called *rd–bisimulation*, is different: it extends classic bisimulation by allowing a read operation to be matched also by an invisible transition leading to an equivalent state. Hence, the following two processes are equated by *rd–bisimulation*:

$$rd(a).P\ +\tau.P \quad \sim_{rd} \quad \tau.P$$

Intuitively, $rd(a)$ is an operation that does not modify the TS and so has the same effect as an action τ; this justifies that a $rd(a)$ operation can be matched by a τ. The reverse doesn't hold as the behavior of $rd(a)$ is more restrictive: it can be performed only if $\langle a\rangle$ is in TS.

$\mathbf{L}[inp]$ includes a conditional input operation – of the form $inp(a)?P\_Q$ – that tests for presence of tuple $\langle a\rangle$: if the tuple is present, then it is removed and P will be executed; otherwise, the execution will continue with Q. The internal transitions for the barbed semantics are enriched by those corresponding to the test for absence of the required tuple; let us consider $inp(a)?P\_Q$: if $\langle a\rangle$ is absent, then Q is started with an internal step. The coarsest congruence is called *inp–bisimulation* (firstly presented in [BGZ97b]) and allows additional forms of matching. A typical law that holds for this bisimulation is:

$$inp(a)?P\_Q\ +\tau.Q \quad \sim_{inp} \quad in(a).P\ +\tau.Q$$

Intuitively, this law holds because if the tuple $\langle a\rangle$ is present, then $inp(a)?P\_Q$ behaves like $in(a).P$; otherwise, $inp(a)?P\_Q$ behaves like $\tau.Q$. However, the τ of the right-hand-side process can always be executed, even when the tuple $\langle a\rangle$ is present; for this reason the left-hand-side process needs the $\tau.Q$ summand.

$\mathbf{L}[rdp]$ includes a conditional read operation $rdp(a)?P\_Q$, differing from $inp(a)?P\_Q$ only because the tuple $\langle a\rangle$ is tested for presence and not removed. The barbed semantics is as above, while the resulting coarsest congruence is different in this case (and called *rdp–bisimulation*). It offers further new matchings, well exemplified by the following typical law:

$$rdp(a)?P\_P \quad \sim_{rdp} \quad \tau.P$$

Intuitively, this law holds because $rdp(a)?P\_P$ performs an internal transition (either by reading tuple $\langle a\rangle$ – that has the same effect of a τ – or by checking the absence of $\langle a\rangle$) and then behaves like P.

We also show that the right semantics for $\mathbf{L}[rd, inp]$, as well as for all the other languages extending $\mathbf{L}[rdp]$, is the *rdp–bisimulation*. Hence the lattice of the eight languages collapses to a smaller four-points lattice of bisimulation semantics, as reported in Figure 1.

The paper is organized as follows. Sections 2–5 introduce the syntax, the operational semantics and the observational semantics for languages **L**, $\mathbf{L}[rd]$, $\mathbf{L}[inp]$, and $\mathbf{L}[rdp]$, respectively. Hence, Section 2 is mainly devoted to the issue of asynchronous communication, Section 3 to the read operation, Section 4 to the

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

Table 1: Structural congruence.

conditional input operation and, finally, Section 5 to the conditional read operation. We show that language **L** is Turing powerful by encoding a RAM; a much simpler definition of a RAM is also given using $\mathbf{L}[inp]$, where neither the alternative composition operator nor the restriction operator are used. Section 6 studies the various further combinations that can be obtained and gives a precise mathematical relation between the syntactic and semantic lattices. Finally, some conclusive remarks on future research and comparison with related literature are reported in Section 7.

2 The Language L

In this section we present the language **L**. It is essentially an asynchronous version of CCS [Mil89] (without relabeling), in which the standard input and output prefixes a and $\overline{a}$ are changed in $in(a)$ and $out(a)$, respectively.

Let $Mess$, ranged over by a, b, ..., be the set of message names, and let Var, ranged over by X, Y, ..., be the set of agent variables. We define *processes* the terms obtained by the following grammar:

$$C \ ::= \ \mathbf{0} \ \mid \ \mu.C \ \mid \ C|C \ \mid \ C+C \ \mid \ C\backslash a \ \mid \ X \ \mid \ rec\,X.C$$

where the possible prefixes μ are:

$$\mu \ ::= \ \tau \ \mid \ in(a) \ \mid \ out(a)$$

The term $\mathbf{0}$ denotes one inactive process, and it is usually omitted for the sake of simplicity. The possible prefixes are τ (standing for an internal action), $in(a)$, and $out(a)$ (standing for the input and output primitives of Linda, respectively). We consider the usual parallel ($|$), choice ($+$), restriction ($\backslash$) and recursion ($rec\,X.C$) operators. In the following, if $A = \{a_1, \ldots, a_n\}$, then $P\backslash A$ is a shorthand for $P\backslash a_1 \backslash \ldots \backslash a_n$; moreover, we consider only closed terms and guarded recursion [Mil89].

The prefix $out(a)$ introduces in TS a new tuple with contents a, denoted by $\langle a \rangle$. The input prefix $in(a)$ requires the presence of $\langle a \rangle$ in TS. If $\langle a \rangle$ is present, then it is removed.

Tuples are not considered in the syntax of processes. In order to represent tuples in TS, we introduce *states*, that are defined as the terms obtained by the following grammar:

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

(1)	$out(a).P \xrightarrow{\tau} \langle a\rangle \mid P$	(2)	$\langle a\rangle \xrightarrow{\overline{a}} \mathbf{0}$		
(3)	$in(a).P \xrightarrow{a} P$	(4)	$\tau.P \xrightarrow{\tau} P$		
(5)	$\dfrac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$	(6)	$\dfrac{P \xrightarrow{\alpha} P'}{P\mid Q \xrightarrow{\alpha} P'\mid Q}$		
(7)	$\dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P\mid Q \xrightarrow{\tau} P'\mid Q'}$	(8)	$\dfrac{P \xrightarrow{\alpha} P'}{P\backslash a \xrightarrow{\alpha} P'\backslash a}$	$\alpha \neq a, \overline{a}$	
(9)	$\dfrac{P \equiv Q \quad Q \xrightarrow{\alpha} Q' \quad P' \equiv Q'}{P \xrightarrow{\alpha} P'}$				

Table 2: Operational semantics of **L**.

A state is the parallel composition of processes and tuples, with the possibility to define local message names using the restriction operator. In the following P, Q, ..., are used to range over states and processes (the actual meaning will be made clear by the context), and *Agent* denotes the set of possible states. We use also $fn(P)$, defined as follows, to denote the free names in P:

$$
\begin{array}{ll}
fn(\mathbf{0}) = fn(X) = \emptyset & fn(P|Q) = fn(P+Q) = fn(P) \cup fn(Q) \\
fn(\langle a\rangle) = \{a\} & fn(P\backslash a) = fn(P) \setminus \{a\} \\
fn(in(a).P) = fn(out(a).P) = \{a\} \cup fn(P) & fn(rec\,X.P) = fn(\tau.P) = fn(P)
\end{array}
$$

In order to define the operational semantics for **L**, we first define a *structural congruence*; this relation captures the fact that, for example, the order of the terms in a parallel composition has no effect on its behavior. The structural congruence $\equiv$ is defined as the smallest congruence satisfying the axioms in Table 1. Axiom (xi) is α–conversion, where $P[b/a]$ denotes the term obtained by renaming all the free occurrences of the name a in P with the name b.

Next, we define a *labeled transition system* (*Agent*, *Label*, $\longrightarrow$) specifying how states (called also agents) evolve. $Label \stackrel{def}{=} \{\tau, a, \overline{a}\}$ (ranged over by α, β, ...) is the set of the possible labels. The labeled transition relation $\longrightarrow$ is the smallest one satisfying the axioms and rules in Table 2.

Axiom (1) shows that an output prefix $out(a)$ can generate a new tuple performing an internal action; then the tuple $\langle a\rangle$ is able to give its contents to some process in the environment, by performing an action labeled with $\overline{a}$ according to axiom (2). Axiom (3) allows an input prefix to consume a message in the environment by performing one action labeled with a, the complementary of $\overline{a}$. The other axiom and rules are the usual ones for the prefix τ (axiom (4)), for the choice operator (rule (5)), for the parallel composition operator (rules (6) and (7)), for the restriction operator (rule (8)), and for giving the possibility, to structurally congruent agents, to execute the same actions (rule (9)). There are no rules for recursion because its semantics is defined by the structural axiom (xii) which applies one unfolding step to a recursively defined process.

The set *Agent* is closed with respect to the transition relation, i.e. for every agent P, if $P \xrightarrow{\alpha} P'$, for some α and P', then also P' is a term in *Agent*.

2.1 Example

In this section we present an implementation of a Random Access Machine (RAM) [SS63] in our asynchronous version of CCS. In this way, we show that we do not loose Turing–equivalence by passing from synchronous to asynchronous communication.

A RAM is a computational model composed of a finite set of registers that can hold arbitrary large natural numbers and by 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 its last instruction 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 contents 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 contents 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 : $DecJump(r_3, 1)$

The translation of the RAM in our language is essentially borrowed from the one for CCS in [Tau89]. We model the contents of the program counter by means of tuples: if the next instruction to execute is the i^{th}, then TS contains the tuple $\langle p_i \rangle$.

To model the instructions we proceed in the following way:

$$
\begin{array}{lcl}
[\![i : Succ(r_j)]\!] & \stackrel{def}{=} & rec\,X.in(p_i).out(inc_j).in(ack_j).out(p_{i+1}).X \\
[\![i : DecJump(r_j, s)]\!] & \stackrel{def}{=} & rec\,X.in(p_i).out(test_j).(in(zero_j).out(p_s).X + in(dec_j).out(p_{i+1}).X)
\end{array}
$$

A $Succ$ instruction on register r_j at position i is represented by an agent that consumes the "program counter tuple", produces a tuple $\langle inc_j \rangle$, representing a request for increment of register r_j, waits for an acknowledgement that the increment has occurred, i.e. consumes a tuple $\langle ack_j \rangle$, and finally 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 $\langle p_i \rangle$, produces a tuple $\langle test_j \rangle$, representing a request for testing register r_j and decrementing it if its contents is greater than zero; if the contents of the agent corresponding to r_j is zero, then a tuple $\langle zero_j \rangle$ is produced, the agent consumes that tuple and updates the program counter in order to perform a jump to the s^{th} instruction by adding $\langle p_s \rangle$; otherwise, after the decrement is performed, a tuple $\langle dec_j \rangle$ is produced, the agent consumes that tuple and updates the program counter by adding $\langle p_{i+1} \rangle$. The use of the recursion operator in the representation of the instructions permits to reuse them.

The register r_j, that initially contains the value 0, is modeled by the agent Z_j defined as follows:

$$
\begin{array}{lcl}
Z_j & \stackrel{def}{=} & rec\,X.in(test_j).out(zero_j).X + in(inc_j).out(ack_j).(O_j | in(a).X)\backslash a \\
O_j & \stackrel{def}{=} & rec\,X.in(test_j).out(dec_j).out(a) + in(inc_j).out(ack_j).(E_j | in(b).X)\backslash b \\
E_j & \stackrel{def}{=} & rec\,Y.in(test_j).out(dec_j).out(b) + in(inc_j).out(ack_j).(X | in(a).Y)\backslash a
\end{array}
$$

If the agent Z_j receives a $test_j$ message, then the tuple $\langle zero_j \rangle$ is inserted in TS in order to state that the register r_j contains the value 0. If an inc_j message is received, then Z_j sends the corresponding ack_j and it becomes the term $(O_j | in(a).Z_j)\backslash a$. The term $in(a).Z_j$ is blocked by its guard $in(a)$ until the agent O_j creates a tuple $\langle a \rangle$ (observe that a is a local name). At this point of the computation, the register r_j should contain the value 1; then, if a tuple $\langle test_j \rangle$ is inserted in TS by some instruction, then the term O_j generates the tuple $\langle dec_j \rangle$ and activates the term Z_j by sending the message a. Otherwise, if a tuple $\langle inc_j \rangle$ is inserted in TS, then O_j sends the corresponding ack_j and becomes $(E_j | in(b).O_j)\backslash b$. In this case the term $in(b).O_j$ is guarded by the prefix $in(b)$, and it waits to be activated by the term E_j. We have used a different name b (instead of a) in order to avoid that the term E_j will incorrectly activate the term $in(a).Z_j$. The term E_j is defined in the same way of O_j, with the unique difference that the name a is used instead of b. In this way, the agent O_j is used to represent odd values, while E_j is used for even values.

Let consider the program $I_1, \ldots, I_k$ with inputs $n_1, \ldots, n_m$ that uses the registers $r_1, \ldots, r_l$. In order to execute it, first we have to introduce every input n_i in the corresponding register r_i. This is done by the following agent B that performs the *bootstrap* of the system by initializing the registers before emitting the program counter tuple $\langle p_1 \rangle$:

$$B \stackrel{def}{=} \underbrace{out(inc_1)\ldots out(inc_1)}_{n_1 \text{ times}}.\underbrace{in(ack_1)\ldots in(ack_1)}_{n_1 \text{ times}}\ldots\underbrace{out(inc_m)\ldots out(inc_m)}_{n_m \text{ times}}.\underbrace{in(ack_m)\ldots in(ack_m)}_{n_m \text{ times}}.out(p_1)$$

The above program is then modeled by:

$$Z_1|\ldots|Z_l|B|[\![I_1]\!]|\ldots|[\![I_k]\!]$$

2.2 Observational Semantics

The approach we follow in order to define an adequate observational semantics for our language, is inspired by [MS92]: firstly, identify the internal transitions, then define the observable actions, and finally find the coarsest congruence contained in the barbed semantics (a very coarse equivalence that equates processes that are bisimilar on internal actions and offer, at any pair of related states, the same observable actions). As usual, for the language **L** the internal actions (denoted with $P \longrightarrow P'$) are those labeled with τ:

$P \longrightarrow P'$ iff $P \xrightarrow{\tau} P'$ for some P'

More attention must be paid in order to identify what is observable or not. Following the intuition of [HT91], it is clear that what an observer can see is just the TS: an observer can input (or read) any tuple that is present in TS, and so that has been previously emitted. On the contrary, the observer has no means for realizing if the system is performing input (as well as read) operations. Then we use the following definition of committed action:

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

The resulting definition of barbed bisimulation is then the following:

Definition 2.1 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 $(P, Q) \in \mathcal{R}$. ■

This definition of barbed bisimulation is essentially the one in [ACS96] (introduced in the setting of the asynchronous π–calculus) according to which only the channel names in the output labels are considered as visible. In that paper, the asynchronous bisimulation of [HT91] is proved to be the coarsest congruence contained in the barbed bisimulation. We show that for our language **L** the coarsest congruence is instead the (standard, strong) bisimulation [Mil89], recalled below.

Definition 2.2 A binary, symmetric relation $\mathcal{R}$ on *Agent* is a *bisimulation* if $(P, Q) \in \mathcal{R}$ implies:

- if $P \xrightarrow{\alpha} P'$ then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{R}$.

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

This difference w.r.t. [ACS96] is due to the fact that in the asynchronous π–calculus, a process can receive a message and then immediately resends it. 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 in our language because the instantaneous emission is not allowed. This is formalized by the following proposition.

Proposition 2.3 Let P be an agent such that $P \stackrel{\overline{a}}{\not\longrightarrow}$. If $P \stackrel{b}{\longrightarrow} P'$ then also $P' \stackrel{\overline{a}}{\not\longrightarrow}$.

Proof. First of all we recall that the tuples $\langle a \rangle$ cannot appear in processes but only in states. Provided that only the tuples $\langle a \rangle$ are able to generate transitions labeled with $\overline{a}$, processes cannot perform transitions labeled with $\overline{a}$. The proof of the proposition is by induction on the proof of the transition $P \stackrel{b}{\longrightarrow} P'$ where the base case is P equal to $in(b).P'$, where P' is a program. ■

In order to prove that bisimulation is the coarsest congruence contained in the barbed bisimulation for the language **L**, 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 bisimilar.

Proposition 2.4 Bisimulation is a congruence for the operators of **L**. ■

The proof of the congruence result is omitted here (and in the following sections) because standard [Mil89]. 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 $L = \{a_1, \ldots, a_n\}$.

Proposition 2.5 Given $n \geq 0$, if $P \stackrel{\overline{a}}{\longrightarrow} P_1 \stackrel{\overline{a}}{\longrightarrow} \ldots \stackrel{\overline{a}}{\longrightarrow} P_n$ 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 $P_{n-1} \stackrel{\overline{a}}{\longrightarrow} P_n$. ■

Theorem 2.6 Let P and Q be agents of **L**. 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 proposition. Let $L = fn(P) \cup fn(Q)$; observe that L is finite.

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

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

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 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:

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

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 $Ag_l^i \stackrel{\alpha}{\longrightarrow} R'$, 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.

We now verify that $\mathcal{R}$ is a bisimulation.

- $S \stackrel{\overline{a}}{\longrightarrow} S'$:
  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:

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

then:

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

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 $V_i \stackrel{\bullet}{\sim} W_i$. 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 $W_3 \not\downarrow \overline{b_a^1}$). The previous observations on the presence tokens, ensures 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 $\langle a\rangle$, must be 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 must consist of the generation and the withdrawal of the presence token $\langle b_a^1\rangle$, respectively. Observe that $S'|R \stackrel{\bullet}{\sim} T'|R$ and $fn(S'), fn(T') \subseteq L$, then also $(S', R') \in \mathcal{R}$.

- $S \xrightarrow{a} S'$:
It is not restrictive to suppose $S \equiv S_1 | \prod_n \langle a\rangle$ (with $n \geq 0$) where $S_1 \stackrel{\overline{a}}{\not\longrightarrow}$. Then also $S_1 \xrightarrow{a} S_1'$ because the term $\prod_n \langle a\rangle$ cannot 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:

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

then:

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

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 $T \xrightarrow{\overline{a}} T^1 \xrightarrow{\overline{a}} \ldots \xrightarrow{\overline{a}} T^n$ because the agent R does not contain any tuple. If $T_1 \stackrel{def}{=} T^n$, then by Proposition 2.5, $T \equiv T_1 | \prod_n \langle a\rangle$. Hence $W_{4n} \equiv T_1 | \prod_n\langle c_a\rangle|R$ 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:

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

then:

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

Observe that $V_{4n+2} \downarrow \overline{a}$ while $V_{4n+3} \not\downarrow \overline{a}$. This follows from Proposition 2.3 because $S_1 \stackrel{\overline{a}}{\not\longrightarrow}$ and $S_1 \stackrel{a}{\longrightarrow} S_1'$. Also W_{4n} must offers 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 $in(a)$. 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 $\langle a \rangle$ must be removed by T_1, hence $T_1 \stackrel{a}{\longrightarrow} T_1'$. We can conclude that $W_{4n+4} \equiv T_1' | \coprod_n \langle c_a \rangle | R$.

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:

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

then:

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

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, $W_{8n+4} \equiv T_1' | \coprod_n \langle a \rangle | R$. This ensures $(S_1' | \coprod_n \langle a \rangle, T_1' | \coprod_n \langle a \rangle) \in \mathcal{R}$. We have already observed that $S' \equiv S_1' | \coprod_n \langle a \rangle$. Moreover, $T_1 \stackrel{a}{\longrightarrow} T_1'$ ensures $T \stackrel{a}{\longrightarrow} T_1' | \coprod_n \langle a \rangle$.

- $S \stackrel{\tau}{\longrightarrow} S'$:
  This implies $S|R \longrightarrow S'|R$. The barbed equivalent agent $T|R$ must be able to perform the reduction step $T|R \longrightarrow W$ where $S'|R \stackrel{\bullet}{\sim} W$. By the observations on the presence tokens, we can assert that the agent R is not involved in the reduction step of $T|R$. Otherwise, W could generate a presence token in only one reduction step while $S'|R$ cannot. Hence, $W \equiv T'|R$ with $T \stackrel{\tau}{\longrightarrow} T'$. $T' \equiv T_1' | \coprod_n \langle a \rangle$ because $T \equiv T_1 | \coprod_n \langle a \rangle$ and $T_1 \stackrel{a}{\longrightarrow} T_1'$. Finally, observe that $S'|R \stackrel{\bullet}{\sim} T'|R$ and $fn(S'), fn(T') \subseteq L$, then also $(S', R') \in \mathcal{R}$. ■

Corollary 2.7 Bisimulation is the coarsest congruence contained in the barbed bisimulation for **L**.

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 $\asymp \subseteq \stackrel{\bullet}{\sim}$ follows $P|R \stackrel{\bullet}{\sim} Q|R$. By Theorem 2.6 also $P \sim Q$ holds. ■

(10)	$rd(a).P \xrightarrow{\underline{a}} P$	(11)	$\dfrac{P \xrightarrow{\underline{a}} P' \quad Q \xrightarrow{\overline{a}} Q'}{P\|Q \xrightarrow{\tau} P'\|Q}$

Table 3: Additional rules for $\mathbf{L}[rd]$.

3 The Language $\mathbf{L}[rd]$

In this section we present the language $\mathbf{L}[rd]$. The syntax of this language is obtained by extending the possible prefixes of the language $\mathbf{L}$ with the new prefix:

$$\mu \ ::= \ rd(a)$$

Also the set *Label* of the possible labels is extended with a new label $\underline{a}$ standing for the execution of a prefix $rd(a)$. The operational semantics of the language $\mathbf{L}[rd]$ is defined by the structural congruence of Table 1 and by the SOS rules of Table 2 extended with the axiom and rule of Table 3. Moreover, the side condition $\alpha \neq \underline{a}$ must be added to the rule (8) of Table 2, because also the new label $\underline{a}$ cannot pass trough a restriction on the name a.

In rule (11) the execution of the rd operation does not change the TS, hence the tuple occurring in the agent Q is not removed (i.e. Q is left unchanged).

3.1 Observational Semantics

Also for $\mathbf{L}[rd]$ we investigate the coarsest congruence contained in the barbed bisimulation. The obtained equivalence is called rd–bisimulation and it is defined as follows.

Definition 3.1 A binary, symmetric relation $\mathcal{R}$ on *Agent* is a rd–bisimulation if $(P, Q) \in \mathcal{R}$ implies:

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

Two agents P and Q are rd–bisimilar, written $P \sim_{rd} Q$, if there exists a rd–bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. ■

This new rd–bisimulation allows to simulate a transition labeled $\underline{a}$ with an internal τ labeled action. The intuition is that $rd(a)$ is an operation that does not modify the TS and so has the same effect as an action τ. Instead, a τ cannot be matched by a $rd(a)$ operation, because the behavior of $rd(a)$ is more restrictive, because it can be performed only if $\langle a \rangle$ is in TS. This implies that the term $rd(a).P$ can be absorbed when occurring in a summation context with $\tau.P$:

$$rd(a).P + \tau.P \ \sim_{rd} \ \tau.P$$

In order to prove that the rd–bisimulation is the coarsest congruence contained in the barbed bisimulation, we follow the approach presented in the previous section for $\mathbf{L}$.

Proposition 3.2 rd–bisimulation is a congruence for the operators of $\mathbf{L}[rd]$. ■

Theorem 3.3 Let P and Q be agents of $\mathbf{L}[rd]$. If $P|R \stackrel{\bullet}{\sim} Q|R$ for every agent R, then $P \sim_{rd} Q$.

Proof. The structure of the proof is the same as the proof of Theorem 2.6. The agent R must be extended:

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

where, for each $l \in L$:

$$Ag_l^5 \quad \stackrel{def}{=} \quad rec\, X.out(l).out(b_l^5).in(l).in(b_l^5).X$$

A new set of agents Ag_l^5 is introduced, and a new set of fresh and distinct presence tokens b_l^5 is considered. Also for Ag_l^5 it is ensured that for every R' such that $Ag_l^5 \xrightarrow{\alpha} R'$, then $R' \longrightarrow R'' \downarrow \overline{b_l^5}$.

The relation $\mathcal{R}$ is defined as in the proof of Theorem 2.6. Given the pair $(S, T) \in \mathcal{R}$ the analysis of the possible derivations of S is the same for $S \xrightarrow{\overline{a}} S'$ and $S \xrightarrow{a} S'$. In the case of $S \xrightarrow{\tau} S'$, the agent $T|R$ could respond with the new kind of synchronization due to the labels $\underline{a}$ and $\overline{a}$. But this is not possible because the agent R does not contains tuples (i.e. $T \stackrel{\overline{a}}{\not\longrightarrow}$) and none of its subagent Ag_l^i is guarded by a $rd(a)$ prefix (i.e. $T \stackrel{\underline{a}}{\not\longrightarrow}$). Only the new case $S \xrightarrow{\underline{a}} S'$ is presented.

- $S \xrightarrow{\underline{a}} S'$:
  Also in this case it is not restrictive to suppose $S \equiv S_1 | \prod_n \langle a \rangle$ where $S_1 \stackrel{\overline{a}}{\not\longrightarrow}$. Then also $S_1 \xrightarrow{\underline{a}} S_1'$ because the term $\prod_n \langle a \rangle$ cannot infer such a derivation. Moreover, $S' \equiv S_1' | \prod_n \langle a \rangle$.

  As in the case $S \xrightarrow{a} S'$ of the proof of Theorem 2.6, we first consider a sequence of reduction steps that renames the n tuples $\langle a \rangle$ appearing in $S|R$, in tuples $\langle c_a \rangle$. 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:

$$\begin{array}{lll} S|R & \longrightarrow & S_1 | \prod_{n-1} \langle a \rangle | out(b_a^2).out(c_a).in(b_a^2).Ag_a^2 | R' \quad (\stackrel{def}{=} V_1) \\ & \ldots & \\ & \longrightarrow & S_1 | \prod_n \langle c_a \rangle | R \qquad\qquad\qquad\qquad\qquad (\stackrel{def}{=} V_{4n}) \end{array}$$

  We have already proved that the generation and consumption of the presence token $\langle b_a^2 \rangle$ implies that the agent $T|R$ has to reply with the reductions:

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

  where $V_i \stackrel{\bullet}{\sim} W_i$ and $W_{4n} \equiv T_1 | \prod_n \langle c_a \rangle | R$ with $T \equiv T_1 | \prod_n \langle a \rangle$ and $W_{4n} \not\downarrow \overline{a}$. This ensures $T_1 \stackrel{\overline{a}}{\not\longrightarrow}$.
  The agent V_{4n} is now able to perform the following reductions. Let R'' be the term:

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

  then:

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

  Observe that $V_{4n+2} \downarrow \overline{a}$ as also $V_{4n+3} \downarrow \overline{a}$. The agent W_{4n} must offer equivalent reduction steps:

$$W_{4n} \longrightarrow W_{4n+1} \longrightarrow \ldots \longrightarrow W_{4n+5}$$

(12)	$inp(a)?P\_Q \xrightarrow{a} P$	(13)	$inp(a)?P\_Q \xrightarrow{\neg a} Q$
(14)	$\dfrac{P \xrightarrow{\neg a} P' \quad Q \not\xrightarrow{\overline{a}}}{P\|Q \xrightarrow{\neg a} P'\|Q}$	(15)	$\dfrac{P \xrightarrow{\neg a} P'}{P\backslash a \xrightarrow{\tau} P'\backslash a}$

Table 4: Additional rules for $\mathbf{L}[inp]$.

where $V_i \stackrel{\bullet}{\sim} W_i$. The presence token $\langle b_a^5\rangle$ appears after two reduction steps, then:

$$W_{4n+2} \equiv T_1 | \prod_n \langle c_a\rangle|\langle a\rangle|\langle b_a^5\rangle|in(a).in(b_a^5).Ag_a^5|R''$$

We now have to analyze the reduction step $W_{4n+2} \longrightarrow W_{4n+3}$ where $W_{4n+3} \downarrow \overline{a}$. The observations on the presence tokens allows to state that the agent R'' cannot be involved in this reduction step (otherwise a new presence token could be generated after one single step). Also the agent $in(a).in(b_a^5).Ag_a^5$ cannot be involved, otherwise $W_{4n+3} \not\downarrow \overline{a}$ (remember that $T_1 \not\xrightarrow{\overline{a}}$). Then, the step must be inferred by T_1. As the first case we consider $T_1 \xrightarrow{\overline{b}} T_1'$ for a generic name b, but we observe that this cannot be because no agent can synchronize with this action. The same holds for $T_1 \xrightarrow{b} T_1'$ and $T_1 \xrightarrow{\underline{b}} T_1'$ if $b \neq a$. In the case $T_1 \xrightarrow{a} T_1'$ the action could synchronize with the tuple $\langle a\rangle$, but the result of the synchronization would be the consumption of the tuple $\langle a\rangle$ from which follows the contradiction $W_{4n+3} \not\downarrow \overline{a}$ (remember that by Proposition 2.3 also $T_1' \not\xrightarrow{\overline{a}}$). The only possible cases are $T_1 \xrightarrow{\underline{a}} T_1'$ or $T_1 \xrightarrow{\tau} T_1'$. The following reductions must be performed in order to remove the presence token b_a^5, hence $W_{4n+5} \equiv T_1' | \prod_n \langle c_a\rangle|R$.

The n occurrences of the tuple $\langle c_a\rangle$ in V_{4n+5} can be now renamed as $\langle a\rangle$ by the agent Ag_a^4. Let R''' be the term:

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

then:

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

The alternate generation and consumption of the presence token $\langle b_a^4\rangle$ ensures:

$$W_{4n+5} \longrightarrow W_{4n+6} \longrightarrow \ldots \longrightarrow W_{8n+5}$$

where $V_i \stackrel{\bullet}{\sim} W_i$ and $W_{8n+5} \equiv T_1' | \prod_n \langle a\rangle|R$. This ensures $(S_1' | \prod_n\langle a\rangle, T_1' | \prod_n\langle a\rangle) \in \mathcal{R}$. We have already observed that $S' \equiv S_1' | \prod_n\langle a\rangle$. Moreover, $T_1 \xrightarrow{\underline{a}} T_1'$ (or $T_1 \xrightarrow{\tau} T_1'$) ensures $T \xrightarrow{a} T_1' | \prod_n\langle a\rangle$ (or $T \xrightarrow{\tau} T_1' | \prod_n\langle a\rangle$). ■

Corollary 3.4 rd–bisimulation is the coarsest congruence contained in the barbed bisimulation for $\mathbf{L}[rd]$. ■

4 The Language $\mathbf{L}[inp]$

In this section we present the language $\mathbf{L}[inp]$. Its syntax is obtained by extending the syntax of processes of the language $\mathbf{L}$ with the new construct:

$$C \quad ::= \quad inp(a)?C\_C$$

The process $inp(a)?P\_Q$ tests if the tuple $\langle a\rangle$ is present in TS: if it is available, then it is removed and the process P is chosen, otherwise Q is executed. In order to describe this behavior in an SOS style, we add the new label $\neg a$ to the set *Label*, and the rules of Table 4 to the ones of Table 2. In this case the side condition for the new label (i.e. $\alpha \neq \neg a$) must be added not only to the rule (8), but also to the rule (6) of Table 2.

Axiom (12) of Table 4 indicates that a program $inp(a)?P\_Q$ can synchronize with a tuple $\langle a\rangle$ (removing it) and become P. Instead, if the program does not find the required message a, it can guess its absence by performing an action labeled with $\neg a$ (axiom (13)). If a process P willing to perform a $\neg a$ labeled derivation 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 this $\neg a$ operation cannot be executed (rule (14)). On the other hand (rule (15)), if the agent P is restricted on the name a, its $\neg a$ operation becomes a local step of computation (i.e. labeled with τ), this because no further agents can offer message a; in other words, the search for a has finished because it has become a local name.

Rule (14) uses a negative premise; it's easy to see that our transition system specification is strictly stratifiable [Gro93], thus there exists a unique transition system agreeing with it.

4.1 Example

In Section 2 we have shown an implementation of the RAM in the language **L**. We show that the extended language **L**[*inp*] allows a much simpler implementation, in which the registers can be represented by means of tuples only, and not by active agents. In fact, the *inp* primitive allows to test the absence of a certain tuple in order to verify if a register contains the value 0 or not.

In the new version of the RAM, we model the contents of registers in the following way: if register r_j contains the number n, then n tuples $\langle r_j\rangle$ are in the tuple space. Hence, a *Succ* instruction on register r_j only adds a new tuple $\langle r_j\rangle$, while a *DecJump* on the same register, executes an *inp* operation on the message name r_j: if one tuple $\langle r_j\rangle$ is found then it is removed, otherwise the jump to the specified instruction is obtained by introducing the corresponding program counter tuple.

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

Then, the agent modeling the program $I_1,\ldots,I_k$ with inputs $n_1,\ldots,n_m$ can be defined as:

$$\langle p_1\rangle|\underbrace{\langle r_1\rangle|\ldots|\langle r_1\rangle}_{n_1\text{ times}}|\ldots|\underbrace{\langle r_m\rangle|\ldots|\langle r_m\rangle}_{n_m\text{ times}}|[\![I_1]\!]|\ldots|[\![I_k]\!]$$

4.2 Observational Semantics

Also for **L**[*inp*] we investigate the coarsest congruence contained in the barbed bisimulation. The new operator *inp* requires a further discussion about the notion of reduction for **L**[*inp*]. In fact, it is reasonable to consider a transition labeled with $\neg a$ as a reduction step, because reductions are usually defined as those transformation steps that a *self-contained* agent is able to perform independently from the context. This is true not only for transitions labeled with τ, but also for the new label $\neg a$. Indeed, the label $\neg a$ has been introduced only for helping an SOS formulation of the semantics, while it is conceptually as internal as the τ step is. Then, we redefine reductions:

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

For **L**[*inp*], according to the new definition of reduction, the coarsest congruence contained in the barbed equivalence is called *inp*–bisimulation and it is defined as follows.

Definition 4.1 A binary, symmetric relation $\mathcal{R}$ on *Agent* is an *inp*–bisimulation if $(P,Q)\in\mathcal{R}$ implies:

- if $P \xrightarrow{\alpha} P'$, with $\alpha \neq \neg a$, then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ 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 $(P', Q') \in \mathcal{R}$.

Two agents P and Q are *inp*–bisimilar, written $P \sim_{inp} Q$, if there exists an *inp*–bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. ■

This new *inp*–bisimulation allows to simulate a transition labeled $\neg a$ with an internal τ labeled action. Hence the following law holds:

$$inp(a)?P\_Q + \tau.Q \ \sim_{inp} \ in(a).P + \tau.Q$$

Intuitively, this holds because if the tuple $\langle a \rangle$ is present, then $inp(a)?P\_Q$ behaves like $in(a).P$; otherwise, $inp(a)?P\_Q$ behaves like $\tau.Q$. However, the τ of the right-hand-side process can always be executed, even when the tuple $\langle a \rangle$ is present; for this reason the left-hand-side process needs the $\tau.Q$ summand.

We present a proposition which states that if an agent has one derivation labeled with $\neg a$, then it cannot contain any tuple $\langle a \rangle$ (i.e. it has no derivation labeled with $\overline{a}$).

Proposition 4.2 Given the agent P, if $P \xrightarrow{\neg a} P'$ then $P \stackrel{\overline{a}}{\not\longrightarrow}$.

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

Proposition 4.3 *inp*–bisimulation is a congruence for the operators of $\mathbf{L}[inp]$. ■

Theorem 4.4 Let P and Q be agents of $\mathbf{L}[inp]$. If $P|R \stackrel{\bullet}{\sim} Q|R$ for every agent R, then $P \sim_{inp} Q$.

Proof. The structure of the proof is the same as the proof of Theorem 2.6. The agent R is the same used in the proof of Theorem 3.3.

The relation $\mathcal{R}$ is defined as in the proof of Theorem 2.6. Given the pair $(S, T) \in \mathcal{R}$ the analysis of the possible derivations of S is the same for $S \xrightarrow{\overline{a}} S'$ and $S \xrightarrow{a} S'$. The cases $S \xrightarrow{\tau} S'$ and the new $S \xrightarrow{\neg a} S'$ are presented.

- $S \xrightarrow{\tau} S'$:
  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 $S \xrightarrow{\tau} S'$ is matched by T with a $\neg b$: we produce all the tuples $\langle l' \rangle$, whose message names are in L', in the TS, thus disallowing T to perform $\neg l'$ derivations. By Proposition 4.2 $T \stackrel{\overline{l'}}{\not\longrightarrow}$ for each $l' \in L'$.

  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 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'' be 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:

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

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

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

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. Also none of the tuples $\langle l' \rangle$ can be consumed: if more than one occurrence of $\langle l' \rangle$ is present in W_{2n}, it's easy to see that $\langle l' \rangle$ occurs in T, but this is not possible (remember that, for every $l' \in L'$, $T \not\downarrow \overline{l'}$); thus, if $\langle l' \rangle$ is consumed, then neither the agents $in(l').in(b^5_{l'}).Ag^5_{l'}$ can be involved, because as first step they remove one of the tuples $\langle l' \rangle$. Hence, 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 this step. 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. LetR''' be the term:

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

then:

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

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

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

where $V_i \stackrel{\bullet}{\sim} W_i$. The consumption of the presence tokens in $2n$ steps ensures that $W_{4n+1} \equiv T'|R$. 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 mimic 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 $(S', T') \in \mathcal{R}$. ∎

Corollary 4.5 *inp*–bisimulation is the coarsest congruence contained in the barbed bisimulation for $\mathbf{L}[inp]$. ∎

5 The Language $\mathbf{L}[rdp]$

In this section we present the language $\mathbf{L}[rdp]$. Its syntax is obtained by extending the syntax of processes of the language $\mathbf{L}$ with the new construct:

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

The process $rdp(a)?P\_Q$ tests if the tuple $\langle a \rangle$ is present in TS: if it is available then the program P is chosen (without consuming the tuple), otherwise Q is executed. In Table 5 the semantics of the rdp primitive is

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

Table 5: Additional rules for $\mathbf{L}[rdp]$.

defined: the action of reading the tuple $\langle a \rangle$ (and the following choice of the process P) is labeled with $\underline{a}$ (axiom (16)), while when the process Q is chosen the label is $\neg a$ (axiom (17)). The semantics of the language $\mathbf{L}[rdp]$ is defined by the axioms and rules of Table 2 (plus the the side condition $\alpha \neq \underline{a}, \neg a$ for the rule (8) and $\alpha \neq \neg a$ for the rule (6)), the rule (11) of Table 3 (defining the behavior of the synchronization due to the the labels $\underline{a}$ and $\overline{a}$), the rules (14) and (15) of Table 4 (defining the behaviour of the label $\neg a$ w.r.t. parallel composition and restriction), and the new axioms in Table 5. Also adding these rules, the transition system specification remains strictly stratifiable [Gro93].

5.1 Observational Semantics

Also for $\mathbf{L}[rdp]$ we investigate the coarsest congruence contained in the barbed bisimulation. As before the definition of reduction considers as reduction steps also the derivations labeled with $\neg a$. The coarsest congruence contained in the barbed equivalence is called *rdp*–bisimulation and it is defined as follows.

Definition 5.1 A binary, symmetric relation $\mathcal{R}$ on *Agent* is an *rdp*–bisimulation if $(P, Q) \in \mathcal{R}$ implies:

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

Two agents P and Q are *rdp*–bisimilar, written $P \sim_{rdp} Q$, if there exists an *rdp*–bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. ■

A peculiar feature of *rdp*–bisimulation, is that a τ can be matched with a $\underline{a}$ or a $\neg a$ labeled derivation:

$$rdp(a)?P\_P \;\sim_{rdp}\; \tau.P$$

Intuitively, this law holds because $rdp(a)?P\_P$ performs an internal transition (either by reading tuple $\langle a \rangle$ or by checking the absence of $\langle a \rangle$ – that have the same effect of a τ) and then behaves like P. In general, if a term P is able to perform both a $\underline{a}$ and a $\neg a$ derivation becoming P', then it completely simulates the behavior of $\tau.P'$ which becomes P' independently of the context.

The other matchings allowed by the *rdp*–bisimulation are the same of the ones present in the *rd*– and *inp*–bisimulation, with the addition of a new derived one. These derived matchings follows from the fact that derivations labeled with $\underline{a}$ or $\neg a$ can be matched by τ labeled derivations, that, in turn, can be matched by a pair of derivations, labeled with $\underline{a}$ and $\neg a$, under the conditions explained above.

Proposition 5.2 *rdp*–bisimulation is a congruence for the operators of $\mathbf{L}[rdp]$. ■

Theorem 5.3 Let P and Q be agents of $\mathbf{L}[rdp]$. If $P|R \overset{\bullet}{\sim} Q|R$ for every agent R, then $P \sim_{rdp} Q$.

Proof. The structure of the proof is the same as the proof of Theorem 2.6. The agent R is the same used in the proof of Proposition 3.3.

The relation $\mathcal{R}$ is defined as in the proof of Proposition 2.6. Given the pair $(S,T) \in \mathcal{R}$ the analysis of the possible derivations of S is the same for $S \xrightarrow{\overline{a}} S'$ and $S \xrightarrow{a} S'$. The cases $S \xrightarrow{\tau} S'$, $S \xrightarrow{\neg a} S'$, and $S \xrightarrow{\underline{a}} S'$ are presented.

- $S \xrightarrow{\tau} S'$:
  The proof is similar to the corresponding case in Section 4, but the set L' must be defined as $L' \stackrel{def}{=} \{b \mid T \xrightarrow{\neg b} T' \wedge S'|R \overset{\bullet}{\sim} T'|R\}$. If $T \xrightarrow{\underline{b}} T'$ and also $T \xrightarrow{\neg a} T''$ with $(S'|R, T''|R) \in \mathcal{R}$, then if the tuples corresponding to the names in L' are in TS, then the derivation labeled with $\underline{b}$ can synchronize with the tuple $\langle b \rangle$. As in the proof of Theorem 4.4 we have that (the agent R'' is the same as the one defined in Section 4:
  $$S|R \longrightarrow \ldots \longrightarrow S| \prod_{l' \in L'} \langle l' \rangle | \prod_{l' \in L'} \langle b^5_{l'} \rangle | \prod_{l' \in L'} in(l').in(b^5_{l'}).Ag^5_{l'}|R'' \quad (\stackrel{def}{=} V_{2n})$$
  and that the barbed equivalent agent $T|R$ must offer:
  $$T|R \longrightarrow \ldots \longrightarrow T| \prod_{l' \in L'} \langle l' \rangle | \prod_{l' \in L'} \langle b^5_{l'} \rangle | \prod_{l' \in L'} in(l').in(b^5_{l'}).Ag^5_{l'}|R'' \quad (\stackrel{def}{=} W_{2n})$$
  where $V_i \overset{\bullet}{\sim} W_i$.
  The agent V_{2n} is now able to perform the reduction due to $S \xrightarrow{\tau} S'$:
  $$V_{2n} \longrightarrow S'| \prod_{l' \in L'} \langle l' \rangle | \prod_{l' \in L'} \langle b^5_{l'} \rangle | \prod_{l' \in L'} in(l').in(b^5_{l'}).Ag^5_{l'}|R'' \quad (\stackrel{def}{=} V_{2n+1})$$
  The agent W_{2n} must allow the reduction step $W_{2n} \longrightarrow W_{2n+1}$ with $V_i \overset{\bullet}{\sim} W_i$. We have already shown that the subagent R'' and the agents $in(l').in(b^5_{l'}).Ag^5_{l'}$ cannot be involved in this reduction step, and that the tuples $\langle l' \rangle$ cannot be consumed. Then derivations labeled with b or $\overline{b}$ cannot be performed by T. Also derivations labeled with $\neg c$ cannot be performed. In fact, if $V_{2n} \xrightarrow{\neg c} V_{2n+1}$, then this derivation is inferred by $T \xrightarrow{\neg c} T'$. Reasoning as below, we can prove that $(S'|R, T'|R) \in \mathcal{R}$, hence $c \in L'$ by definition of L'. Thus the tuple $\langle c \rangle$ is present in V_{2n}, hence the derivation $V_{2n} \xrightarrow{\neg c} V_{2n+1}$ is not possible.

  Hence, the possible cases are $T \xrightarrow{\tau} T'$ and $T \xrightarrow{\underline{b}} T'$. In the second case the only tuples available for reading are the tuples $\langle l' \rangle$ (with $l' \in L'$), then $b \in L'$. This implies also $T \xrightarrow{\neg b} T''$ and $(S', T'') \in \mathcal{R}$ by definition of L'.

  As shown in Section 4 the tuples $\langle l' \rangle$ and the presence tokens $\langle b^5_{l'} \rangle$ in V_{2n+1} can be removed:
  $$V_{2n+1} \longrightarrow \ldots \longrightarrow S'|R \quad (\stackrel{def}{=} V_{4n+1})$$
  and also:
  $$W_{2n+1} \longrightarrow \ldots \longrightarrow W_{4n+1}$$
  with $V_i \overset{\bullet}{\sim} W_i$ and $W_{4n+1} \equiv T'|R$.

- $S \xrightarrow{\neg a} S'$:
  The proof is the same as the previous case with the only difference that the set L' is defined as $L' \stackrel{def}{=} \{c \mid T \xrightarrow{\neg c} T' \wedge c \neq a \wedge S'|R \overset{\bullet}{\sim} T'|R\}$. In this case the message name a is not an element of L' because T can mimic 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 not only $T \xrightarrow{\tau} T'$, and $T \xrightarrow{\underline{b}} T'$ (with also $T \xrightarrow{\neg b} T''$ such as $(S', T'') \in \mathcal{R}$), but also the derivation $T \xrightarrow{\neg a} T'$.

- $S \xrightarrow{a} S'$:
We consider $S \equiv S_1 | \prod_n \langle a \rangle$ with $S_1 \not\xrightarrow{\overline{a}}$. In this case we combine the techniques used in the Sections 3 and 4.
Let $L' \stackrel{def}{=} \{c \mid T \xrightarrow{\neg c} T' \wedge c \neq a \wedge S'|R \stackrel{\bullet}{\sim} T'|R\}$. We consider a sequence of reduction steps which first renames all the tuples $\langle a \rangle$ in $\langle c_a \rangle$, and then generates all the tuples $\langle l' \rangle$ with $l' \in L'$ (the agent R'' is the same as the one defined in Section 4):

$$
\begin{array}{rcll}
S|R & \longrightarrow & \ldots & \\
 & \longrightarrow & S_1 | \prod_n \langle c_a \rangle | R & (\stackrel{def}{=} V_{4n}) \\
 & \longrightarrow & \ldots & \\
 & \longrightarrow & S | \prod_n \langle c_a \rangle | \prod_{l' \in L'} \langle l' \rangle | \prod_{l' \in L'} \langle b^5_{l'} \rangle | \prod_{l' \in L'} in(l').in(b^5_{l'}).Ag^5_{l'} | R'' & (\stackrel{def}{=} V_{6n})
\end{array}
$$

We can combine the observations made in the Sections 3 and 4 concluding that:

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

with $V_i \stackrel{\bullet}{\sim} W_i$ and $W_{6n} \equiv T_1 | \prod_n \langle c_a \rangle | \prod_{l' \in L'} \langle l' \rangle | \prod_{l' \in L'} \langle b^5_{l'} \rangle | \prod_{l' \in L'} in(l').in(b^5_{l'}).Ag^5_{l'} | R''$ where $T \equiv T_1 | \prod_n \langle a \rangle$ and $T_1 \not\xrightarrow{\overline{a}}$.

As already observed in Section 3 the derivation $S_1 \xrightarrow{a} S'_1$ holds, with $S' \equiv S'_1 | \prod_n \langle a \rangle$. The agent V_{6n} is now able to perform the following reductions. Let R''' be the term:

$$\prod_n \langle c_a \rangle | \prod_{l' \in L'} in(l').in(b^5_{l'}).Ag^5_{l'} | \prod_{l \in L} Ag^1_l | \prod_{l \in L} Ag^2_l | \prod_{l \in L} Ag^3_l | \prod_{l \in L} Ag^4_l | \prod_{l \in L \setminus \{L' \cup \{a\}\}} Ag^5_l$$

then:

$$
\begin{array}{rcll}
V_{6n} & \longrightarrow & S_1 | \prod_{l' \in L'} \langle l' \rangle | \langle a \rangle | out(b^5_a).in(a).in(b^5_a).Ag^5_a | R''' & (\stackrel{def}{=} V_{6n+1}) \\
 & \longrightarrow & S_1 | \prod_{l' \in L'} \langle l' \rangle | \langle a \rangle | \langle b^5_a \rangle | in(a).in(b^5_a).Ag^5_a | R''' & (\stackrel{def}{=} V_{6n+2}) \\
 & \longrightarrow & S'_1 | \prod_{l' \in L'} \langle l' \rangle | \langle a \rangle | \langle b^5_a \rangle | in(a).in(b^5_a).Ag^5_a | R''' & (\stackrel{def}{=} V_{6n+3}) \\
 & \longrightarrow & S'_1 | \prod_{l' \in L'} \langle l' \rangle | \langle b^5_a \rangle | in(b^5_a).Ag^5_a | R''' & (\stackrel{def}{=} V_{6n+4}) \\
 & \longrightarrow & S'_1 | \prod_{l' \in L'} \langle l' \rangle | Ag^5_a | R''' & (\stackrel{def}{=} V_{6n+5})
\end{array}
$$

This derivations are possible because the agent Ag^5_a is not involved in the creation of the tuples $\langle l' \rangle$, in fact $a \notin L'$. Because of $V_{6n} \stackrel{\bullet}{\sim} W_{6n}$, then also:

$$W_{6n} \longrightarrow \ldots \longrightarrow W_{6n+5}$$

with $V_i \stackrel{\bullet}{\sim} W_i$. The presence token $\langle b^5_l \rangle$ ensures that (excluding the derivation $W_{6n+2} \longrightarrow W_{6n+3}$) the agent W_{6n} must reply with the same reductions performed by Ag^5_a. We now analyze the derivation $W_{6n+2} \longrightarrow W_{6n+3}$ which (as we have shown in Section 4) must be inferred by the agent T_1. As above, the labels of the kind b, $\overline{b}$ or $\neg c$ are not allowed. Hence, the possible cases are $T \xrightarrow{\tau} T'$ and $T \xrightarrow{b} T'$. In the second case the only tuples available for reading are the tuples $\langle l' \rangle$ (with $l' \in L'$), and $\langle a \rangle$. If $b \in L'$, this implies also $T \xrightarrow{\neg b} T''$ and $(S', T'') \in \mathcal{R}$ (following from the definition of L').

As shown in the Section 4 the presence tokens $\langle b^5_{l'} \rangle$ in V_{2n+1} can be removed, and then (as shown in Section 3) the tuples $\langle c_a \rangle$ can be renamed in $\langle a \rangle$:

$$
\begin{array}{rcll}
V_{6n+5} & \longrightarrow & \ldots & \\
 & \longrightarrow & S'_1 | \prod_n \langle c_a \rangle | R & (\stackrel{def}{=} V_{8n+5}) \\
 & \longrightarrow & \ldots & \\
 & \longrightarrow & S'|R & (\stackrel{def}{=} V_{12n+5})
\end{array}
$$

But also:

$$W_{6n+5} \longrightarrow \ldots \longrightarrow W_{12n+5}$$

with $V_i \stackrel{\bullet}{\sim} W_i$ and $W_{12n+5} \equiv T'|R$. ■

$$
\begin{array}{ccc}
out(a).P \xrightarrow{\tau} \langle a\rangle | P & & \langle a\rangle \xrightarrow{\overline{a}} \mathbf{0} \\[2ex]
\tau.P \xrightarrow{\tau} P & & in(a).P \xrightarrow{a} P \\[2ex]
rd(a).P \xrightarrow{\underline{a}} P & & inp(a)?P\_Q \xrightarrow{a} P \\[2ex]
inp(a)?P\_Q \xrightarrow{\neg a} Q & & rdp(a)?P\_Q \xrightarrow{\underline{a}} P \\[2ex]
rdp(a)?P\_Q \xrightarrow{\neg a} Q & & \dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \\[2ex]
\dfrac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} & \alpha \neq \neg a & \dfrac{P \xrightarrow{\neg a} P' \quad Q \stackrel{\overline{a}}{\not\longrightarrow}}{P|Q \xrightarrow{\neg a} P'|Q} \\[2ex]
\dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} & & \dfrac{P \xrightarrow{\underline{a}} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q} \\[2ex]
\dfrac{P \xrightarrow{\alpha} P'}{P\backslash a \xrightarrow{\alpha} P'\backslash a} & \alpha \neq a, \overline{a}, \underline{a}, \neg a & \dfrac{P \xrightarrow{\neg a} P'}{P\backslash a \xrightarrow{\tau} P'\backslash a} \\[2ex]
 & \dfrac{P \equiv Q \quad Q \xrightarrow{\alpha} Q' \quad P' \equiv Q'}{P \xrightarrow{\alpha} P'} &
\end{array}
$$

Table 6: Operational semantics of LINPA.

Corollary 5.4 *rdp*–bisimulation is the coarsest congruence contained in the barbed bisimulation for $\mathbf{L}[rdp]$. ■

6 The Other Languages

In this section we analyze the remaining combinations: $\mathbf{L}[rd, inp]$ and all the languages that extend $\mathbf{L}[rdp]$. We use LINPA as name for the complete language $\mathbf{L}[rd, inp, rdp]$ whose complete set of SOS rules is recalled in Table 6.

6.1 Example

We present an example that shows a possible use of the *rd* and *rdp* primitives. In Table 7 a client–server system with access control is presented. We do not specify the kind of services, but we describe only the coordination mechanisms between the clients, the server, and the access controller.

Let U be the set of possible user indexes, and let V be the set of those that can obtain the access permission. $Client_i$ can send to the $Server$ a service request by emitting the tuple $\langle req_i\rangle$. Before complying with the request of $Client_i$, the $Server$ checks if the tuple $\langle i\rangle$ is present in the local space shared by the $Server$ and the $Controller$. If the client is enabled, then the service is performed and the message yes_i is sent to the client; if the client is not enabled, the answer no_i is sent. When $Client_i$ receives a no_i answer, it asks to the agent $Controller$ to be introduced in the set of enabled users, and waits the answer of the $Controller$. If the index of the client is in the set V (i.e., $i \in V$) the $Controller$ emits the tuple $\langle i\rangle$ and sends to $Client_i$ the corresponding answer $enable_i$. If the index of the client is not in the set V, then it blocks waiting for the (never coming) answer of the $Controller$.

$System$	$\stackrel{def}{=}$	$\prod_{i\in U} Client_i \vert (Server \vert Controller) \backslash U$
$Client_i$	$\stackrel{def}{=}$	$rec\, X.out(req_i).(in(yes_i).X + in(no_i).out(perm_i).rd(enable_i).X)$
$Server$	$\stackrel{def}{=}$	$\prod_{i\in U} rec\, X.in(req_i).rdp(i)?out(yes_i).X\_out(no_i).X$
$Controller$	$\stackrel{def}{=}$	$\prod_{i\in V} rec\, X.in(perm_i).out(i).out(enable_i).X$

Table 7: A client–server system with access control.

6.2 Observational Semantics

As made in the previous cases, we have investigated the coarsest congruence contained in the barbed bisimulation also for the remaining combinations. The result is that for all these languages, such equivalence is the rdp–bisimulation. This can be proved following the proofs presented in Section 5.

We can then summarize our results by introducing the function Obs that, given one of the languages, returns its adequate observational semantics:

$Obs(\mathbf{L}) = \sim$
$Obs(\mathbf{L}[rd]) = \sim_{rd}$
$Obs(\mathbf{L}[inp]) = \sim_{inp}$
$Obs(\mathbf{L}[rdp]) = Obs(\mathbf{L}[rd, inp]) = Obs(\mathbf{L}[rd, rdp]) = Obs(\mathbf{L}[inp, rdp]) = Obs(\mathrm{LINPA}) = \sim_{rdp}$

It is trivial to verify that if $L' \subseteq L''$ then also $Obs(L') \subseteq Obs(L'')$.

The following proposition compares the four bisimulations.

Proposition 6.1 Consider the bisimulations $\sim$, $\sim_{rd}$, $\sim_{inp}$ and $\sim_{rdp}$ on the language LINPA. The following hold:

$$\begin{array}{lcl} \sim & \subseteq & \sim_{rd} \\ \sim & \subseteq & \sim_{inp} \\ \sim_{rd} & \subseteq & \sim_{rdp} \\ \sim_{inp} & \subseteq & \sim_{rdp} \end{array}$$

■

The most interesting result is that the rdp–bisimulation is the coarsest congruence contained in the barbed bisimulation also for the language $\mathbf{L}[rd, inp]$, even if such a language does not contain the rdp primitive. This is made more clear observing that:

$$rd(a).P + inp(a)?Q\_P \;\sim\; rdp(a)?P\_P + in(a).Q$$

This law holds because the two agents allow the same derivations. The bisimulation relation $\sim$ is finer than the rdp–bisimulation, then the couple of agents above is equated also by $\sim_{rdp}$. Moreover we have shown that $rdp(a)?P\_P \sim_{rdp} \tau.P$, then (by transitivity and substitutivity):

$$rd(a).P + inp(a)?Q\_P \;\sim_{rdp}\; \tau.P + in(a).Q$$

where both the terms do not contain the rdp primitive.

Observe that the four inclusions above are strict; moreover, only the rd–bisimulation and the inp–bisimulation are not comparable. The examples in Section 1 justify the observation above.

The relation between the syntactic and semantic lattices is illustrated by the following theorem.

Theorem 6.2 Let L' and L'' be two languages such that $L' \subseteq L''$. Then, for all $P, Q \in L'$, $(P, Q) \in Obs(L')$ if and only if $(P, Q) \in Obs(L'')$.

Proof. If $Obs(L') \neq Obs(L'')$, then the difference consists of additional matchings in $Obs(L'')$ involving labels that cannot occur in the transitions derivable by agents in L'. ■

7 Conclusion and Related Work

In this paper we have incrementally presented the process algebra LINPA that is able to express all the features of generative communication. Our main original contributions are:

- (i) A satisfactory interleaving operational semantics for all the Linda coordination primitives; some of these primitives have already received an operational treatment (see below for comparison), while others – like rdp – appear here for the first time.
- (ii) The characterization of the four observational semantics for the eight different languages; in particular, two of them (*rd–bisimulation* and *rdp–bisimulation*) are completely new; for the other two semantics (classic bisimulation [Mil89] and *inp-bisimulation* [BGZ97b]), only the coarsest congruence result is new.

A lot of work is left for future research. For instance, the definition of a suitable axiomatization of the four congruences for the eight languages. However, here we have provided typical laws that give us the flavour of what such axiomatizations should be. Maybe the most relevant future work is the development of a suitable semantics theory based on a weak version of the barbed semantics. For language **L**, it seems reasonable to claim that such a semantics (we denote $\simeq$) identifies, for example, the following processes:

$out(a).out(b) \;\simeq\; out(b).out(a)$

hence showing that the order of output operations is irrelevant. Unfortunately, $\simeq$ is not a congruence for parallel composition for the whole language including the non-blocking conditional operations because the agent $in(a).inp(b)?\mathbf{0}\_out(c)$ distinguishes the agents above:

$out(a).out(b)|in(a).inp(b)?\mathbf{0}\_out(c) \;\;\not\simeq\;\; out(b).out(a)|in(a).inp(b)?\mathbf{0}\_out(c)$

In fact, the left-hand-side agent can generate the tuple $\langle c\rangle$, while the right-hand-side cannot. We can then conclude that the predicates allows to observe the order of emission of the messages. We are currently investigating the problem of finding the coarsest congruences for the various languages.

One further issue is the definition of a distributed semantics for LINPA and the introduction of suitable truly concurrent observational semantics. From an operational point of view, this problem has already been solved in [BGZ97a], where LINPA receives a distributed semantics in terms of Petri Nets with inhibitor and contextual arcs. For future research we leave instead the characterization of satisfactory observational semantics in this setting.

We end this paper with a short comparison with related literature.

Other Operational Approaches for Asynchronous Communication

In the literature there are several proposals of process algebras based on asynchronous communication. Basically two different approaches have been followed: the one which considers the messages as passive entities which must be stored in buffers [BKPR91, BKP92] and the other one which represents messages as active autonomous entities [HT91, CY94, Pug95, BGZ97b].

The two papers in the first group, however, model buffers in two very different ways: in [BKPR91] the communication medium is not explicitly represented in the language but it is considered as an external data structure which is accessed by the active agents, while in [BKP92] an "encapsulation operator" is introduced to explicitly model the buffers containing the sent messages.

As far as the papers in the second group are concerned, we can see at least three different main approaches, according to [BGZ97b]: the *instantaneous* semantics, the *ordered* semantics and the *unordered* semantics. The instantaneous semantics is the one proposed by [HT91, Bou92] in order to introduce an asynchronous version of the π–calculus [MPW92]. In this approach, $out(a)$ represents the message in TS; for instance, in $out(a)|P$ (the parallel composition of process P with $out(a)$), P can immediately input message a. In other words, $out(a).P$ is to be considered structurally congruent to $\langle a\rangle|P$. To obtain this, in the asynchronous π-calculus outputs cannot be used as prefixes. However, this approach is quite demanding from an implementation point of view and we think that this approach is not very suitable for a language like Linda. Another paper following the instantaneous semantics, even if with different technicalities, is [Pug95] where the introduced asynchronous CCS uses a hybrid form of output prefix: the emission of a message is delayed until it is read or the sending agent executes an observable or a synchronization action. The ordered semantics is the one we have considered in this paper: in one internal step, the sender sends the message and makes it available for the potential partners. In this way, the order of emission is respected by the rendering order. The implementation of $out(a)$ is simpler: the sender sends the message a and then waits for the acknowledgement from the TS. This approach has already received an operational treatment in [HKH95] and [CGZ96]. Finally, according to the unordered semantics, proposed in [BGZ97b], the emission and the rendering of one message are distinct autonomous actions. The operation $out(a)$ is implemented as a send operation of message a to the TS. As in distributed systems no precise knowledge of the relative speed of processes and of their messages can be assumed, the order of emission needs not to be respected by the rendering order (i.e., the order of arrival of the messages in the TS). In [BGZ97b] we proved that the three semantics are not comparable for a simple sublanguage of $\mathbf{L}[inp]$. Further operational semantics include [CY94], where a value-passing CCS with asynchronous communication is presented. The *out* operations are modeled by an explicit emission internal step, but – unlike our proposal – the message remains related to the process from which it has been sent. This approach does not fit properly the feature of generative communication which requires that the sent messages are equally accessible to all processes and bound to none. For example, in [CY94] a message cannot be read by its sending process while in Linda this is allowed.

Other Observational Semantics for Asynchronous Communication

In [BKPR91] it is argued that a trace based semantics is enough for detecting deadlock behavior; hence, the authors said that the failure semantics "fails" in the case of asynchronous communication because in their framework refusal information is not required. In a more recent paper [BKP92], the same authors move to a "special purpose" failure semantics. In fact, in that paper the emission of a message is represented by means of an observable action (instead of a τ labeled one), forcing a new notion of stable state, where not only τ labeled actions must not be fired, but also actions representing emission of messages. On the contrary in [CGZ96], two of the authors have defined a "plain" failure semantics for the subset of LINPA not including the non-blocking conditional operations, which has been also axiomatized for the finite processes. In that paper, also rooted weak bisimulation is studied and suitably axiomatized.

All these semantics shares the fact that even if the processes are asynchronous, the observer is synchronous, i.e., the observer can equally detect the presence of inputs and outputs/tuples. In this paper, following [HT91, ACS96], it is assumed that also the observers are asynchronous: only outputs/tuples are observationally relevant. Honda and Tokoro [HT91] solves this problem by considering ordinary strong bisimulation on an enlarged transition system. The additional transitions are derived from one extra rule stating that from any state any input is possible and the reached state offers a new copy of the corresponding received tuple. Recently, [ACS96] reports a mathematically nicer version of their semantics: the transition system is not modified, but a novel bisimulation semantics, called *asynchronous* bisimulation $\sim_a$, is introduced. Asynchronous bisimulation is proved to be the coarsest congruence contained in the barbed semantics for the asynchronous π-calculus, where the instantaneous semantics is used. On the contrary, in this paper we show that when the *ordered* semantics is used, the coarsest congruence is the classic notion of bisimulation, even if the observer is asynchronous. Nonetheless, we want to stress that the kind of matchings that we have in the *rd-bisimulation* are reminiscent of the matching of the asynchronous bisimulation; indeed, a read operation $rd(a)$ can be simulated, under the instantaneous semantics, by $in(a).out(a)$ (which is the same as writing $in(a).\langle a\rangle$ in our semantics). Hence the law:

$$rd(a).P \ + \tau.P \quad \sim_{rd} \quad \tau.P$$

reminds the asynchronous law in [ACS96]:

$$in(a).(\langle a\rangle|P) \ + \tau.P \quad \sim_a \quad \tau.P$$

Other Semantics of the Read Operation

The other papers on the semantics of Linda, namely [DP95, CJY95, DP96], do not take into account the *inp* and *rdp* predicates. Here we consider the way they model the read operation.

In [DP95] observational equivalences based on testing [DH84] are applied to a language obtained by embedding the Linda primitives in a simple sequential host language. Moreover, the read operation of [DP96] is also treated as an input with a subsequent emission of the consumed message. This means that $rd(a).P$ is just a macro for $in(a).out(a).P$. Moreover, [DP96] studies a testing semantics, with a suitable axiomatic characterization, assuming a synchronous observer. Very similar to the above is the operational modeling of the read operation in [CJY95]; that paper, however, studies no semantic equivalences.

Acknowledgements

We thank the anonymous referees for suggestions that were helpful for improving the presentation.

References

[ACS96] R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. In *CONCUR'96*, volume 1119 of *LNCS*, pages 147–162, Springer Verlag, 1996.

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

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

[BGZ97b] N. Busi, R. Gorrieri, and G. Zavattaro. Three Semantics of the Output Operation for Asynchronous Communication. In *Proc. of Coordination'97*, volume of LNCS, (to appear).

[BKP92] T.S. De Boer, J.W. Klop, and C. Palamidessi. Asynchronous communication in process algebra. In *LICS '92*, pages 137–159. IEEE Computer Society Press, 1992.

[BKPR91] F.S. De Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The Failure of Failures in a Paradigm for Asynchronous Communication. In *Concur '91*, volume 527 of *LNCS*, pages 111–126. Springer Verlag, 1991.

[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.

[CJY95] P. Ciancarini, K.K. Jensen, and D. Yankelevich. On the Operational Semantics of a Coordination Language. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 77–106, Springer Verlag, 1995.

[CY94] R. Cleaveland and D. Yankelevich. An Operational Framework for Value-Passing Processes. In *Proc. of POPL'94*, 1994.

[DH84] R. De Nicola and M.C.B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83-133, 1984.

[DP95] R. De Nicola and R. Pugliese. An Observational Semantics for Linda. In *Proc. of STRICT'95*, pages 129–143, Springer Verlag, 1995.

[DP96] R. De Nicola and R. Pugliese. A Process Algebra based on Linda. In *Proc. of Coordination'96*, volume 1061 of *LNCS*, pages 160–178, Springer Verlag, 1996.

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

[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. In *Proc. of the Tbilisi Symposium on Language, Logic, and Computation*, 1995.

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

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

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed Bisimulation. In *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.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report, University of Aarhus, DAIMI FN-19, 1981.

[Pug95] R. Pugliese. A Process Calculus with Asynchronous Communication. In Proc. *ICTCS'95*, 295-310, World Scientific, 1995.

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

[Tau89] D. Taubner. Finite Representations of CCS and TCSP Programs by Automata and Petri Nets. Volume 369 of *LNCS*, Springer Verlag, 1989.