A Process Algebraic View of Linda Coordination Primitives

Nadia Busi  Roberto Gorrieri  Gianluigi Zavattaro

Technical Report UBLCS-97-6
May 1997

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)
The University of Bologna Department of Computer Science Research Technical Reports are available in gzipped PostScript format via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS or via WWW at URL http://www.cs.unibo.it/. Plain-text abstracts organized by year are available in the directory ABSTRACTS. All local authors can be reached via e-mail at the address last-name@cs.unibo.it. Questions and comments should be addressed to tr-admin@cs.unibo.it.

Recent Titles from the UBLCS Technical Report Series

95-13  Optimal Multi-Block Read Schedules for Partitioned Signature Files, P. Ciaccia, August 1995.
96-5   The Shape of Shade: a Coordination System, S. Castellani, P. Ciancarini, D. Rossi, March 1996.
96-7   Using Bayesian Belief Networks for the Automated Assessment of Students’ Knowledge of Geometry Problem Solving Procedures, M. Roccetti, P. Salomoni, March 1996 (Revised March 1997).
96-9   Towards an Algebra of Actors, M. Gaspari, April 1996.
96-10  Mobile Petri Nets, A. Asperti, N. Busi, May 1996.
96-12  A Logic Coordination Language Based on the Chemical Metaphor, P. Ciancarini, D. Fogli, M. Gaspari, July 1996.
A Process Algebraic View of Linda Coordination Primitives

Nadia Busi    Roberto Gorrieri    Gianluigi Zavattaro

Technical Report UBLCS-97-6

May 1997

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.
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 differences 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.

- **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.

---

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 $P$. 
Conceptually, the execution of the Linda output primitive \( \text{out}(a) \) 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: \( \text{out}(a).P \) 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 \( \text{in}(a) \), but it can also be tested/read by means of the read primitive \( \text{rd}(a) \). Concerning the conditional input/read predicates, we decide to reformulate them as operators \( \text{inp}(a)? P \cdot Q \) and \( \text{rdp}(a)? P \cdot Q \), 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 \( L \) is essentially CCS, 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 \( L \) obtained by adding some of the other Linda coordination primitives. The top language \( L[^{rd}, \text{inp}, \text{rdp}] \) 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, 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 \( \tau \) labeled derivations. In our case, we consider only \( \tau \) labeled steps in those languages that do 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 \( \tau \) 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 \( \pi \)-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 \( \text{out} \) operation (see the concluding section for a comparison).

For \( L[^{rd}] \) the internal transitions are left untouched, but the resulting coarsest congruence, called \( \text{rd–bisimulation} \), 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 \( \text{rd–bisimulation} \):

\[
\text{rd}(a).P + \tau.P \sim_{\text{rd}} \tau.P
\]

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

2. To be precise, we have omitted the relabeling operator and changed the names of the prefixes.
L[inp] includes a conditional input operation – of the form $\text{inp}(a) \cdot P \cdot Q$ – 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 $\text{inp}(a) \cdot P \cdot Q$: if $\langle a \rangle$ is absent, then $Q$ is started with an internal step. The coarsest congruence is called $\text{inp}$–bisimulation (firstly presented in [BGZ97b]) and allows additional forms of matching. A typical law that holds for this bisimulation is:

$$\text{inp}(a) \cdot P \cdot Q + \tau \cdot Q \sim_{\text{inp}} \text{in}(a) \cdot P + \tau \cdot Q$$

Intuitively, this law holds because if the tuple $\langle a \rangle$ is present, then $\text{inp}(a) \cdot P \cdot Q$ behaves like $\text{in}(a) \cdot P$; otherwise, $\text{inp}(a) \cdot P \cdot Q$ behaves like $\tau \cdot Q$. However, the $\tau$ 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 $\tau \cdot Q$ summand.

$L[rdp]$ includes a conditional read operation $\text{rdp}(a) \cdot P \cdot Q$, differing from $\text{inp}(a) \cdot P \cdot Q$ 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 $\text{rdp}$–bisimulation). It offers further new matchings, well exemplified by the following typical law:

$$\text{rdp}(a) \cdot P \cdot P \sim_{\text{rdp}} \tau \cdot P$$

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

We also show that the right semantics for $L[rd, inp]$, as well as for all the other languages extending $L[rdp]$, is the $\text{rdp}$–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 $L$, $L[rd]$, $L[inp]$, and $L[rdp]$, respectively. Hence, Section 2 is mainly devoted to the issue of asynchronous communication, Section 3 to the read operation, Section 4 to the 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 $L[inp]$, 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 $\text{in}(a)$ and $\text{out}(a)$, respectively.

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

$$C ::= 0 \mid \mu.C \mid C|C \mid C + C \mid C\langle a \rangle \mid X \mid \text{rec}X.C$$

where the possible prefixes $\mu$ are:

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

The term 0 denotes one inactive process, and it is usually omitted for the sake of simplicity. The possible prefixes are $\tau$ (standing for an internal action), $\text{in}(a)$, and $\text{out}(a)$ (standing for the input and output primitives of Linda, respectively). We consider the usual parallel ($\|$), choice ($+$),
(i) \( P \circ \emptyset \equiv P \)
(ii) \( P \circ Q \equiv Q \circ P \)
(iii) \( (P \circ Q) \circ R \equiv P \circ (Q \circ R) \)
(iv) \( P + 0 \equiv P \)
(v) \( P + P \equiv P \)
(vi) \( P + Q \equiv Q + P \)
(vii) \( (P + Q) + R \equiv P + (Q + R) \)
(viii) \( 0 \circ a \equiv 0 \)
(ix) \( (P \setminus a) \setminus b \equiv (P \setminus b) \setminus a \)
(x) \( (P \circ Q) \setminus a \equiv P \circ (Q \setminus a) \quad a \notin fn(P) \)
(xi) \( P \setminus a \equiv P \setminus [b/a] \setminus b \quad b \) fresh
(xii) \( rec X.P \equiv P[rec.X.P/X] \)

Table 1. Structural congruence.

restriction (\( \setminus \)) and recursion (\( rec.X.C \)) operators. In the following, if \( A = \{a_1, \ldots, a_n\} \), then \( P \setminus A \) is a shorthand for \( P \setminus a_1 \ldots \setminus a_n \); 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:

\[
P ::= \langle a \rangle \mid C \mid P[P/\] P \langle a \rangle
\]

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 \( P, Q, \ldots \) 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 \( P \):

\[
fn(\emptyset) = \emptyset \quad fn(PQ) = fn(P) + fn(Q) \quad fn(P) = fn(P) \cup \{a\}
fn(\langle a \rangle) = \{a\} \quad fn(P[a/a]) = fn(P) \setminus \{a\}
fn(recX.P) = fn(\tau.P) = fn(P)
fn(out(a).P) = fn(in(a).P) = \{a\} \cup fn(P)
\]

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 \( a \)–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 \( (Agent, Label, \rightarrow) \) specifying how states (called also agents) evolve. \( Label \) \( \triangleq \{\tau, a, \pi\} \) (ranged over by \( \alpha, \beta, \ldots \) ) is the set of the possible labels. The labeled transition relation \( \rightarrow \) 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 \( \pi \) 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 \(\pi\). The other axiom and rules are the usual ones for the prefix \(\tau\) (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 (xi) which applies one unfolding step to a recursively defined process.

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

### 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 lose 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:

- \(\text{Succ}(r_j)\): add 1 to the contents of register \(r_j\);
- \(\text{DecJump}(r_j, s)\): 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):

\[
\begin{align*}
1. & \quad \text{out}(a).P \xrightarrow{\alpha} \langle a \rangle P \\
2. & \quad \langle a \rangle \xrightarrow{\pi} 0 \\
3. & \quad \text{in}(a).P \xrightarrow{\alpha} P \\
4. & \quad \tau.P \xrightarrow{\alpha} P \\
5. & \quad P \xrightarrow{\alpha} P' \\
6. & \quad P + Q \xrightarrow{\alpha} P' \\
7. & \quad P \xrightarrow{\alpha} P', Q \xrightarrow{\pi} Q' \\
8. & \quad P\{Q \xrightarrow{\alpha} P'\} | Q' \xrightarrow{\alpha} P\{Q\} | Q' \\
9. & \quad P \equiv Q \quad Q \xrightarrow{\alpha} Q', P' \equiv Q' \\
10. & \quad P \xrightarrow{\alpha} P'
\end{align*}
\]

Table 2. Operational semantics of \(L\).
The above program is then modeled by:

\[ \text{DecJump}(r_2, A) \]
\[ \text{Succ}(r_1) \]
\[ \text{DecJump}(r_3, 1) \]

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:

\[
\begin{align*}
[i : \text{Succ}(r_j)] & \quad \overset{\text{def}}{=} \text{rec } X.\text{in}(p_i).\text{out}(\text{inc}_j).\text{in}(\text{ack}_j).\text{out}(p_{i+1}).X \\
[i : \text{DecJump}(r_j, s)] & \quad \overset{\text{def}}{=} \text{rec } X.\text{in}(p_i).\text{out}(\text{test}_j).(\text{in}(\text{zero}_j).\text{out}(p_i).X + \text{in}(\text{dec}_j).\text{out}(p_{i+1}).X)
\end{align*}
\]

A \text{Succ} instruction on register \( r_j \) at position \( i \) is represented by an agent that consumes the "program counter tuple", produces a tuple \( \langle \text{inc}_j \rangle \), representing a request for increment of register \( r_j \), waits for an acknowledgment that the increment has occurred, i.e. consumes a tuple \( \langle \text{ack}_j \rangle \), and finally updates the program counter by adding a tuple \( \langle p_{i+1} \rangle \).

An instruction \text{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 \( \langle \text{test}_j \rangle \), 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 \( \langle \text{zero}_j \rangle \) is produced, the agent consumes that tuple and updates the program counter in order to perform a jump to the \( j^{th} \) instruction by adding \( \langle p_i \rangle \); otherwise, after the decrement is performed, a tuple \( \langle \text{dec}_j \rangle \) 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:

\[
\begin{align*}
Z_j & \quad \overset{\text{def}}{=} \text{rec } X.\text{in}(\text{test}_j).\text{out}(\text{zero}_j).X + \text{in}(\text{inc}_j).\text{out}(\text{ack}_j).(O_j[\text{in}(a).X]\langle a \rangle).
\\
O_j & \quad \overset{\text{def}}{=} \text{rec } X.\text{in}(\text{test}_j).\text{out}(\text{dec}_j).\text{out}(a) + \text{in}(\text{inc}_j).\text{out}(\text{ack}_j).(E_j[\text{in}(b).X]\langle b \rangle).
\\
E_j & \quad \overset{\text{def}}{=} \text{rec } Y.\text{in}(\text{test}_j).\text{out}(\text{dec}_j).\text{out}(b) + \text{in}(\text{inc}_j).\text{out}(\text{ack}_j).(X[\text{in}(a).Y]\langle a \rangle).
\end{align*}
\]

If the agent \( Z_j \) receives a \( \text{test}_j \) message, then the tuple \( \langle \text{zero}_j \rangle \) is inserted in TS in order to state that the register \( r_j \) contains the value 0. If an \( \text{inc}_j \) message is received, then \( Z_j \) sends the corresponding \( \text{ack}_j \) and it becomes the term \( (O_j[\text{in}(a).Z_j]\langle a \rangle) \). The term \( \text{in}(a).Z_j \) is blocked by its guard \( \text{in}(a) \) 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 \( \langle \text{test}_j \rangle \) is inserted in TS by some instruction, then the term \( O_j \) generates the tuple \( \langle \text{dec}_j \rangle \) and activates the term \( Z_j \) by sending the message \( a \). Otherwise, if a tuple \( \langle \text{inc}_j \rangle \) is inserted in TS, then \( O_j \) sends the corresponding \( \text{ack}_j \) and becomes \( (E_j[\text{in}(b).O_j]\langle b \rangle) \). In this case the term \( \text{in}(b).O_j \) is guarded by the prefix \( \text{in}(b) \), 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 \( \text{in}(a).Z_j \). 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 \( \langle p_1 \rangle \):

\[
B \overset{\text{def}}{=} \text{out}(\text{inc}_1) \ldots \text{out}(\text{inc}_1).\text{in}(\text{ack}_1) \ldots \text{in}(\text{ack}_1) \ldots \text{out}(\text{inc}_m) \ldots \text{out}(\text{inc}_m).\text{in}(\text{ack}_m) \ldots \text{in}(\text{ack}_m) \ldots \text{out}(p_1)
\]

The above program is then modeled by:

\[
Z_1 \ldots Z_l | B [\langle I_1 \rangle] \ldots [\langle I_k \rangle]
\]
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 \( \overline{a} \)) are those labeled with \( / \):

\[
P \xrightarrow{\overline{a}} P' \text{ iff } P \xrightarrow{a} P' \text{ for some } P'
\]

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:

\[
P \xrightarrow{a} \text{ iff } P \xrightarrow{\overline{a}} P' \text{ for some } P'
\]

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 \( P \xrightarrow{a} P' \) then there exists \( Q' \) such that \( Q \xrightarrow{a} Q' \) 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 \( P \sim Q \), 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 \( \pi \)-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 \( P \xrightarrow{a} P' \) then there exists \( Q' \) such that \( Q \xrightarrow{a} Q' \) 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 \( \pi \)-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 \( \overline{a} \) 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 \( P \xrightarrow{\overline{a}} P' \). If \( P \xrightarrow{b} P' \) then also \( P' \xrightarrow{\overline{a}} P'' \).

**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 \( P \xrightarrow{b} P' \) 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.
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, $\prod_i P_i$ is used as a shorthand for 0, if $n = 0$, or for $n$ copies of the agent $Q$ composed in parallel, while $\prod_{i \in I} P_i$ stands for 0 if $I = \emptyset$, or for $P_{a_1} \ldots P_{a_n}$ if $L = \{a_1, \ldots, a_n \}$.

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

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

**Theorem 2.6** Let $P$ and $Q$ be agents of $L$. If $P \xrightarrow{R} Q \xrightarrow{R}$ 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 $R$:

$$
R \overset{def}{=} \prod_{i \in L} Ag_i^1 \prod_{i \in L} Ag_i^2 \prod_{i \in L} Ag_i^3 \prod_{i \in L} Ag_i^4
$$

such that the relation:

$$
\mathcal{R} = \{(S, T) \mid S \xrightarrow{R} T \text{ and } fn(S), fn(T) \subseteq L \}
$$

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

The agents $Ag_i^j$ are defined as follows:

- $Ag_i^1 \overset{def}{=} \mathsf{rec} X.\mathsf{in}(l).\mathsf{out}(b_i^1).\mathsf{in}(b_i^1).X$
- $Ag_i^2 \overset{def}{=} \mathsf{rec} X.\mathsf{in}(l).\mathsf{out}(b_i^2).\mathsf{out}(c_i).\mathsf{in}(b_i^2).X$
- $Ag_i^3 \overset{def}{=} \mathsf{rec} X.\mathsf{out}(l).\mathsf{out}(b_i^3).\mathsf{in}(b_i^3).X$
- $Ag_i^4 \overset{def}{=} \mathsf{rec} X.\mathsf{in}(c_i).\mathsf{out}(b_i^4).\mathsf{out}(l).\mathsf{in}(b_i^4).X$

where $b_i^1$ and $c_i$ are all fresh and distinct names for every $i$ and $l$. The tuples $(b_i^j)$ are called presence tokens: each agent $Ag_i^j$ (and only it) is able to generate and consume the corresponding presence token $(b_i^j)$. Moreover, for every agent $Ag_i^j$, if $Ag_i^j \xrightarrow{a} R'$, then $R' \xrightarrow{a} R' \downarrow b_i^j$, 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.

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

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

  $$
  \prod_{i \in L \setminus \{a\}} Ag_i^1 \prod_{i \in L} Ag_i^2 \prod_{i \in L} Ag_i^3 \prod_{i \in L} Ag_i^4
  $$

  then:

  $$
  S \xrightarrow{\pi} S'[\mathsf{out}(b_a^1).\mathsf{in}(b_a^1).Ag_a^1][R'] \overset{def}{=} V_1
  $$

  $$
  \xrightarrow{\pi} S'[\mathsf{in}(b_a^1).Ag_a^1][R'] \overset{def}{=} V_2
  $$

  $$
  \xrightarrow{\pi} S'[R] \overset{def}{=} V_3
  $$

UBLCS-97-6 9
Observe that \( V_2 \Downarrow \overline{b^3_0} \) while \( V_3 \uparrow \overline{b^3_0} \). The agent \( T|R \) is barbed bisimilar to \( S|R \), hence:

\[
T|R \longrightarrow W_1 \longrightarrow W_2 \longrightarrow W_3
\]

where \( V_i \sim W_i \). Then, also \( T|R \) must generate and then consume the presence token \( \langle b^3_0 \rangle \) (because \( W_2 \Downarrow \overline{b^3_0} \) and \( W_3 \Downarrow \overline{b^3_0} \)). The previous observations on the presence tokens, ensures that the agent \( A_{g_0}^3 \) 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 \( A_{g_0}^3 \). The consumed tuple \( \langle a \rangle \), must be present in \( T \), because \( R \) does not contain any tuple. Hence, \( T \rightarrow \overline{T} \) 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 \( \langle b^3_0 \rangle \), respectively. Observe that \( S'|R \sim T'|R \) and \( fn(S'), fn(T') \subseteq L \), then also \( (S', R') \in R \).

\( S \xrightarrow{a} S' \):

It is not restrictive to suppose \( S \equiv S_1 \prod_n \langle a \rangle \) (with \( n \geq 0 \)) where \( S_1 \xrightarrow{\overline{a} \overline{b^3_0}} \). 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 \( A_{g_0}^3 \). Let \( R'' \) be the term:

\[
\prod_{l \in L} A_{g_0}^1 \mid \prod_{l \in L \setminus \{a\}} A_{g_0}^2 \mid \prod_{l \in L} A_{g_0}^3 \mid \prod_{l \in L} A_{g_0}^4
\]

then:

\[
S|R \quad \longrightarrow \quad S_1 \mid \prod_{n-1} \langle a \rangle \mid out(\overline{b^3_n}) \cdot out(c_a) \cdot in(\overline{b^3_0}) \cdot A_{g_0}^3 \cdot R'' \quad \overset{def}{=} \quad V_1
\]

\[
\longrightarrow \quad S_1 \mid \prod_{n-1} \langle a \rangle \mid \langle b^3_0 \rangle \mid out(c_a) \cdot in(\overline{b^3_0}) \cdot A_{g_0}^3 \cdot R'' \quad \overset{def}{=} \quad V_2
\]

\[
\longrightarrow \quad S_1 \mid \prod_{n-1} \langle a \rangle \mid \langle b^3_0 \rangle \mid \langle c_a \rangle \mid in(\overline{b^3_0}) \cdot A_{g_0}^3 \cdot R'' \quad \overset{def}{=} \quad V_3
\]

\[
\longrightarrow \quad S_1 \mid \prod_{n-1} \langle a \rangle \mid \langle c_a \rangle \mid R \quad \overset{def}{=} \quad V_4
\]

\[ \ldots \]

\[
\longrightarrow \quad S_1 \mid \prod_n \langle c_a \rangle \mid R \quad \overset{def}{=} \quad V_{4n}
\]

Observe that \( V_{4n} \not\sim \overline{a} \) and that alternatively, every two steps, the presence token \( \langle b^3_0 \rangle \) 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 \( V_i \sim W_i \). Also in the sequence of reductions performed by \( T|R \), the presence token \( \langle b^3_0 \rangle \) must be alternatively generated and consumed, every two steps, for \( n \) times. Only the agent \( A_{g_0}^3 \) 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 \( T \rightarrow \overline{T} \rightarrow \overline{T} \rightarrow \ldots \rightarrow \overline{T} \) because the agent \( R \) does not contain any tuple. If \( T_1 \overset{def}{=} \overline{T}^n \), then by Proposition 2.5, \( T \equiv T_1 \mid \prod_n \langle a \rangle \). Hence \( W_{4n} \equiv T_1 \mid \prod_n \langle c_a \rangle \mid R \) where \( W_{4n} \not\sim \overline{a} \).

The agent \( V_{4n} \) is now able to generate a new tuple \( \langle a \rangle \) (because of its subagent \( A_{g_0}^3 \)) which is then consumed by \( S_1 \) performing the derivation \( S_1 \overset{a}{\longrightarrow} S'_1 \). Let \( R''' \) be the term:

\[
\prod_{l \in L} A_{g_0}^1 \mid \prod_{l \in L} A_{g_0}^2 \mid \prod_{l \in L \setminus \{a\}} A_{g_0}^3 \mid \prod_{l \in L} A_{g_0}^4
\]

then:

\[
V_{4n} \quad \longrightarrow \quad S_1 \mid \prod_n \langle c_a \rangle \mid \langle a \rangle \mid out(\overline{b^3_0}) \cdot in(\overline{b^3_0}) \cdot A_{g_0}^3 \cdot R''' \quad \overset{def}{=} \quad V_{4n+1}
\]

\[
\longrightarrow \quad S_1 \mid \prod_n \langle c_a \rangle \mid \langle a \rangle \mid in(\overline{b^3_0}) \cdot A_{g_0}^3 \cdot R''' \quad \overset{def}{=} \quad V_{4n+2}
\]

\[
\longrightarrow \quad S'_1 \mid \prod_n \langle c_a \rangle \mid \langle b^3_0 \rangle \mid \langle a \rangle \cdot A_{g_0}^3 \cdot R''' \quad \overset{def}{=} \quad V_{4n+3}
\]

\[
\longrightarrow \quad S'_1 \mid \prod_n \langle c_a \rangle \mid R \quad \overset{def}{=} \quad V_{4n+4}
\]
Observe that $V_{4n+3} \not\rightarrow \bar{a}$ while $V_{4n+3} \not\parallel \bar{a}$. This follows from Proposition 2.3 because $S_1 \overset{\rightarrow}{\rightarrow} T_{\pi}$ and $S_1 \overset{\rightarrow}{\rightarrow} S'_1$. Also $W_{4n}$ must offer equivalent reduction steps:

$$W_{4n} ightarrow W_{4n+1} \rightarrow \ldots \rightarrow W_{4n+4}$$

where $V_i \not\overset{a}{\overset{=}{}\parallel} W_i$. The fact that the presence token $\langle b_i^0 \rangle$ appears after two reduction steps implies that the first two steps are performed by $Ag_i^0$. Moreover, $W_{4n+3} \not\parallel \bar{a}$ implies that the tuple $\langle a \rangle$, generated by $Ag_i^3$, 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 $Ag_i^j$ with $i = 1, 2$ that performs the operation $in(a)$. This implies the contradiction $V_{4n+3} \not\parallel W_{4n+3}$ because (see the observations about the presence tokens at the beginning of this proof) $W_{4n+3} \rightarrow W \not\rightarrow \bar{b}_i^0$, while $V_{4n+3}$ requires at least two steps in order to generate a new presence token $\langle b_i^0 \rangle$. Then, the tuple $\langle a \rangle$ must be removed by $T_1$, hence $T_1 \overset{a}{\rightarrow} T'_1$. We can conclude that $W_{4n+4} \equiv T'_1[\prod _i (c_i)]R$. The $n$ tuples $(c_i)$ in $V_{4n+4}$ can be now renamed in $(a)$ by the agent $Ag_i^4$. Let $R'''$ be the term:

$$\prod _{i \in L} Ag_i^1 \prod _{i \in L} Ag_i^2 \prod _{i \in L \setminus \{a\}} Ag_i^3 \prod _{i \in L \setminus \{a\}} Ag_i^4$$

then:

$$V_{4n+4} \rightarrow S'_1[\prod _{i \in L} (c_i)]out(b_i^1), out(a), in(b_i^2), Ag_i^4|R'''$$

implies that the first two steps are performed by $Ag_i^4$. This ensures $\prod _{i \in L} (c_i)]out(b_i^1), out(a), in(b_i^2), Ag_i^4|R'''$ (def $V_{4n+5}$)

Moreover, $T_1 \overset{a}{\rightarrow} T'_1$ ensures $T \overset{a}{\rightarrow} T'_1$. Finally, observe that $S'_1 \overset{\rightarrow}{\rightarrow} T'_1[\prod _i (a)]r$. This implies $S| R \rightarrow S'| R$. The barbed equivalent agent $T | R$ must be able to perform the reduction step $T | R \rightarrow W$ where $S'_1 | R \not\rightarrow W$. 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 $S'_1 | R$ cannot. Hence, $W \equiv T' | R$ with $T \overset{r}{\rightarrow} T', T' \equiv T'_1[\prod _i (a)]$ because $T \equiv T_1[\prod _i (a)]$ and $T_1 \overset{a}{\rightarrow} T'_1$. Finally, observe that $S'_1 | R \not\rightarrow T' | R$ and $fn(S'), fn(T') \subseteq L$, then also $(S'_1, R') \in \mathcal{R}$.}

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

**Proof.** Let $\overset{\sim}{=}$ be a congruence contained in $\overset{\Rightarrow}{\Rightarrow}$. We show that $\overset{\sim}{=}$ is a congruence. In fact, if $P \overset{\sim}{=} Q$ then $P | R \overset{\sim}{=} Q| R$ for every agent $R$ because $\overset{\sim}{=}$ is a congruence. By $\overset{\sim}{=}$ follows $P | R \overset{\Rightarrow}{\Rightarrow} Q| R$. By Theorem 2.6 also $P \overset{\sim}{=} Q$ holds.
3 The Language $L[r\!d]$ 

In this section we present the language $L[r\!d]$. The syntax of this language is obtained by extending the possible prefixes of the language $L$ with the new prefix:

$$\mu ::= r\!d(a)$$

Also the set $Label$ of the possible labels is extended with a new label $a$ standing for the execution of a prefix $r\!d(a)$. The operational semantics of the language $L[r\!d]$ 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 $a \neq a$ must be added to the rule (8) of Table 2, because also the new label $a$ cannot pass through a restriction on the name $a$.

In rule (11) the execution of the $r\!d$ 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 $L[r\!d]$ we investigate the coarsest congruence contained in the barbed bisimulation. The obtained equivalence is called $r\!d$–bisimulation and it is defined as follows.

**Definition 3.1** A binary, symmetric relation $R$ on $Agent$ is a $r\!d$–bisimulation if $(P, Q) \in R$ implies:

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

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

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

$$r\!d(a).P + \tau.P \sim_{r\!d} \tau.P$$

In order to prove that the $r\!d$–bisimulation is the coarsest congruence contained in the barbed bisimulation, we follow the approach presented in the previous section for $L$.

**Proposition 3.2** $r\!d$–bisimulation is a congruence for the operators of $L[r\!d]$.

**Theorem 3.3** Let $P$ and $Q$ be agents of $L[r\!d]$. If $P \mid R \sim Q \mid R$ for every agent $R$, then $P \sim_{r\!d} Q$.

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

$$R \overset{def}{=} \prod_{i \in L} A_i \prod_{i \in L} A_i^2 \prod_{i \in L} A_i^3 \prod_{i \in L} A_i^4 \prod_{i \in L} A_i^5$$

### Table 3. Additional rules for $L[r\!d]$. 

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10)</td>
<td>$r!d(a).P \xrightarrow{\mu} P$</td>
</tr>
<tr>
<td>(11)</td>
<td>$P \xrightarrow{a} P', Q \xrightarrow{a} Q' \quad P</td>
</tr>
</tbody>
</table>
A new set of agents $A_{g_2}^i$ is introduced, and a new set of fresh and distinct presence tokens $b_{a}^i$ is considered. Also for $A_{g_2}^i$ it is ensured that for every $R'$ such that $A_{g_2}^i \vdash R'$, then $R' \rightarrow R'' \downarrow b_{a}^i$.

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 $S \rightarrow S'$ and $S' \rightarrow a S'$. In the case of $S \rightarrow S'$, the agent $T|R$ could respond with the new kind of synchronization due to the labels $a$ and $\pi$. But this is not possible because the agent $R$ does not contain tuples (i.e. $T \not\vdash \pi$). Only the new case $S \rightarrow S'$ is presented.

- $S \rightarrow S'$:
  
  Also in this case it is not restrictive to suppose $S \equiv S_1 | \prod_n \langle a \rangle$ where $S_1 \not\vdash \pi$. Then also $S_1 \rightarrow S'_1$ 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 $S \rightarrow S'$ 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 $\langle c \rangle$. Let $R'$ be the term:

$$\prod_{i \in L} A_{g_1}^i | \prod_{i \in L \setminus \{a\}} A_{g_2}^i | \prod_{i \in L} A_{g_3}^i | \prod_{i \in L} A_{g_4}^i \prod_{i \in L} A_{g_5}^i$$

then:

$$S|R \rightarrow S_1 | \prod_{n-1} \langle a \rangle | out(b_{a}^1), out(c_{a}) \cdot in(b_{a}^1), A_{g_2}^1 | R' \quad (\overset{df}{=} V_1)$$

$$\cdots \rightarrow S_1 | \prod_n \langle c \rangle|R \quad (\overset{df}{=} V_n)$$

We have already proved that the generation and consumption of the presence token $\langle b_{a}^1 \rangle$ implies that the agent $T|R$ has to reply with the reductions:

$$T|R \rightarrow W_1 \rightarrow \ldots \rightarrow W_n$$

where $V_i \not\sim W_i$ and $W_n \equiv T_1 | \prod_n \langle c \rangle|R$ with $T \equiv T_1 | \prod_n \langle a \rangle$ and $W_n \not\vdash \pi$. This ensures $T_1 \not\vdash \pi$.

The agent $V_n$ is now able to perform the following reductions. Let $R''$ be the term:

$$\prod_{i \in L} A_{g_1}^i | \prod_{i \in L} A_{g_2}^i | \prod_{i \in L} A_{g_3}^i | \prod_{i \in L} A_{g_4}^i | \prod_{i \in L \setminus \{a\}} A_{g_5}^i$$

then:

$$V_n \rightarrow S_1 | \prod_n \langle c \rangle | out(b_{a}^1).in(a).in(b_{a}^1).A_{g_2}^1 | R'' \quad (\overset{df}{=} V_{n+1})$$

$$\rightarrow S_1 | \prod_n \langle c \rangle | \langle \langle b_{a}^5 \rangle \rangle | in(a).in(b_{a}^1).A_{g_2}^1 | R'' \quad (\overset{df}{=} V_{n+2})$$

$$\rightarrow S'_1 | \prod_n \langle c \rangle | \langle \langle b_{a}^5 \rangle \rangle | in(a).in(b_{a}^1).A_{g_2}^1 | R'' \quad (\overset{df}{=} V_{n+3})$$

$$\rightarrow S'_1 | \prod_n \langle c \rangle | \langle \langle b_{a}^5 \rangle \rangle | in(b_{a}^1).A_{g_2}^1 | R'' \quad (\overset{df}{=} V_{n+4})$$

$$\rightarrow S'_1 | \prod_n \langle c \rangle|R \quad (\overset{df}{=} V_{n+5})$$

Observe that $V_{n+3} \not\vdash \pi$ as also $V_{n+3} \not\vdash \pi$. The agent $W_n$ must offer equivalent reduction steps:

$$W_n \rightarrow W_{n+1} \rightarrow \ldots \rightarrow W_{n+5}$$

where $V_i \not\sim W_i$. The presence token $\langle b_{a}^5 \rangle$ appears after two reduction steps, then:

$$W_{n+2} \equiv T_1 | \prod_n \langle c \rangle | \langle \langle b_{a}^5 \rangle \rangle | in(a).in(b_{a}^1).A_{g_2}^1 | R''$$
We now have to analyze the reduction step $W_{4n+2} \rightarrow W_{4n+3}$ where $W_{4n+3} \downarrow \pi$. 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 $\text{in}(a)$.\text{in}(b^5)_{a}$.\text{Ag}_{a}^{4}$ cannot be involved, otherwise $W_{4n+3} \downarrow \pi$ (remember that $T_1 \not\rightarrow T_1'$). Then, the step must be inferred by $T_1$. As the first case we consider $T_1 \rightarrow T_1'$ for a generic name $b$, but we observe that this cannot be because no agent can synchronize with this action. The same holds for $T_1 \rightarrow T_1'$ and $T_1 \rightarrow T_1'$ if $b \neq a$. In the case $T_1 \rightarrow T_1'$ 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 $W_{4n+3} \downarrow \pi$ (remember that by Proposition 2.3 also $T_1' \not\rightarrow T_1'$). The only possible cases are $T_1 \rightarrow T_1'$ or $T_1 \rightarrow T_1'$. The following reductions must be performed in order to remove the presence token $b^5_{a}$. hence $W_{4n+5} \equiv T_1' | \prod_{a} \langle \alpha \rangle | R$. The $n$ occurrences of the tuple $\langle \alpha \rangle$ in $V_{a_{n+5}}$ can be now renamed as $\langle \alpha \rangle$ by the agent $\text{Ag}_{a}^{4}$. Let $R''$ be the term:

$$
\prod_{i \in L} \text{Ag}_{i}^{1} | \prod_{i \in L} \text{Ag}_{i}^{2} | \prod_{i \in L} \text{Ag}_{i}^{4} | \prod_{i \in L} \text{Ag}_{i}^{5}
$$

then:

$$
V_{4n+5} \rightarrow S_{i} | \prod_{a} \langle \alpha \rangle | \text{out}(b^4), \text{out}(a), \text{in}(b^4)_{a}: \text{Ag}_{a}^{4} | R'' \quad (\overset{d_{e}f}{\equiv} V_{4n+6})
$$

$$
\ldots \rightarrow S_{i} | \prod_{a} \langle \alpha \rangle | R \quad (\overset{d_{e}f}{\equiv} V_{8n+5})
$$

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

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

where $V_i \not\sim W_i$ and $W_{8n+5} \equiv T_1' | \prod_{a} \langle \alpha \rangle | R$. This ensures $(S_{i} | \prod_{a} \langle \alpha \rangle, T_1' | \prod_{a} \langle \alpha \rangle) \in R$. We have already observed that $S_{i} \equiv S_{i} | \prod_{a} \langle \alpha \rangle$. Moreover, $T_1 \rightarrow T_1'$ (or $T_1 \rightarrow T_1'$) ensures $T_1 \rightarrow T_1' | \prod_{a} \langle \alpha \rangle$ (or $T \rightarrow T_1' | \prod_{a} \langle \alpha \rangle$).

**Corollary 3.4** $rd$–bisimulation is the coarsest congruence contained in the barbed bisimulation for $L[rd]$. 

**4** The Language $L[inp]$ 

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

$$
C ::= \text{inp}(a)^? \cdot C - C
$$
The process \( inp(a) \vdash P \cdot Q \) 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 \( \text{Label} \), and the rules of Table 4 to the ones of Table 2. In this case the side condition for the new label (i.e. \( a \neq \neg a \)) 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 \( inp(a) \vdash P \cdot Q \) 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 \( P \cdot Q \) 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 \( \tau \)), 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 \( Dec.Jump \) 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.

\[
\begin{align*}
[i : Succ(r_j)] & \overset{\text{def}}{=} rec X. (inp(r_j) \cdot out(p_{i+1}) \cdot X) \\
[i : Dec.Jump(r_j, s)] & \overset{\text{def}}{=} rec X. (inp(r_j) \cdot out(p_{i+1}) \cdot X) \cdot (out(p_s) \cdot X)
\end{align*}
\]

Then, the agent modeling the program \( l_1, \ldots, l_k \) with inputs \( n_1, \ldots, n_m \) can be defined as:

\[
\langle p_1 \rangle \langle r_1 \rangle \ldots \langle r_m \rangle \langle r_{m+1} \rangle \ldots \langle r_{m+n} \rangle \mid [l_1] \mid \ldots \mid [l_k]
\]

\( n_1 \) times \( n_m \) times

### 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 \( \tau \), 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 \( \tau \) step is. Then, we redefine reductions:

\[
P \rightarrow P' \text{ if } P \xrightarrow{\neg a} P' \text{ or } P \xrightarrow{\neg a} P' \text{ for some } P'
\]

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 \( R \) on \( Agent \) is an \( inp \)-bisimulation if \((P, Q) \in R \) implies:

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

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 \( \tau \) labeled action. Hence the following law holds:

\[
inp(a) \equiv P \cdot Q + \tau.Q \sim_{inp} \inp(a).P + \tau.Q
\]

Intuitively, this holds because if the tuple \( \langle a \rangle \) is present, then \( inp(a) \equiv P \cdot Q \) behaves like \( \inp(a).P \); otherwise, \( inp(a) \equiv P \cdot Q \) behaves like \( \tau.Q \). However, the \( \tau \) 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 \( \exists \)).

**Proposition 4.2** Given the agent \( P \), if \( P \xrightarrow{\neg a} P' \) then \( P \xrightarrow{\neg a} P' \).

**Proof.** By induction on the proof of the transition \( P \xrightarrow{\neg a} P' \). \[ \square \]

**Proposition 4.3** \( inp\)-bisimulation is a congruence for the operators of \( L[\inp] \).

**Theorem 4.4** Let \( P \) and \( Q \) be agents of \( L[\inp] \). If \( P \models R \equiv Q \models R \) 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 \( S \xrightarrow{\exists} S' \) and \( S \xrightarrow{\neg a} S' \). The cases \( S \xrightarrow{\tau} S' \) and the new \( S \xrightarrow{\neg a} S' \) are presented.

- \( S \xrightarrow{\tau} S' \)
  - Let \( L' = \{ a \mid T \xrightarrow{\neg a} T' \} \) 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 \( T \downarrow L' \) for each \( l' \in L' \).
  - The agent \( S|R \) can execute the following sequence of reduction steps in which the agents \( A_{g_i}^j \) (with \( l' \in L' \)) generate the tuples \( \langle l' \rangle \). Let \( R' \) be the term:

\[
\prod_{l \in L} A_g^1 \big| \prod_{l \in L} A_g^2 \big| \prod_{l \in L} A_g^3 \big| \prod_{l \in L} A_g^4 \big| \prod_{l \in L} A_g^5 \big| \prod_{l \in L \setminus \{l_i \}} A_g^5
\]

and \( R'' \) be the agent:

\[
\prod_{l \in L} A_g^1 \big| \prod_{l \in L} A_g^2 \big| \prod_{l \in L} A_g^3 \big| \prod_{l \in L} A_g^4 \big| \prod_{l \in L \setminus L'} A_g^5
\]

then:

\[
S|R \quad \rightarrow \quad S|\langle l_1 \rangle |out(b_{l_1}^5), in(l_1), in(b_{l_1}^5), A_g^5 | R' \quad (\text{def} \ V_1)
\]

\[
S|R \quad \rightarrow \quad S|\langle l_1 \rangle |in(l_1), in(b_{l_1}^5), A_g^5 | R' \quad (\text{def} \ V_2)
\]

\[
\quad \ldots
\]

\[
S|R \quad \rightarrow \quad S|\prod_{l' \in L'} \langle l' \rangle | \prod_{l' \in L'} | in(l'), in(b_{l_1}^5), A_g^5 | R'' \quad (\text{def} \ V_{2n})
\]
The barbed bisimilar agent $T|R$ must allow the sequence of reductions:

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

where $V_i \models W_i$ and $W_{2n} \uparrow R$ for every $l' \in L'$. In order to generate the presence tokens $\langle b_{i'}^5 \rangle$ the agents $Ag_{i'}^5$ must be involved, then:

$$W_{2n} \equiv T|\prod_{l' \in L'}\langle l' \rangle|\prod_{l' \in L'}\langle b_{i'}^5 \rangle|\prod_{l' \in L'}in(l').in(b_{i'}^5).Ag_{i'}^5|R',$$

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

$$V_{2n} \rightarrow S'|\prod_{l' \in L'}\langle l' \rangle|\prod_{l' \in L'}\langle b_{i'}^5 \rangle|\prod_{l' \in L'}in(l').in(b_{i'}^5).Ag_{i'}^5|R' \quad (\overset{\text{def}}{=} V_{2n+1})$$

The agent $W_{2n}$ must allow the reduction step $W_{2n} \rightarrow W_{2n+1}$ with $V_i \models W_i$. 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 $l' \in L'$, $T \uparrow R$); thus, if $\langle l' \rangle$ is consumed, then neither the agents $in(l').in(b_{i'}^5).Ag_{i'}^5$ 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 $L'$, $T \rightarrow a \rightarrow T'$ implies $a \in L'$). Hence, the derivation must be labeled with $\tau$ and $W_{2n+1} \equiv T'|\prod_{l' \in L'}\langle l' \rangle|\prod_{l' \in L'}\langle b_{i'}^5 \rangle|\prod_{l' \in L'}in(l').in(b_{i'}^5).Ag_{i'}^5|R''$ with $T \rightarrow T'$. The tuples $\langle l' \rangle$ and the presence tokens $\langle b_{i'}^5 \rangle$ can be now removed. Let $R'''$ be the term:

$$\prod_{l' \in L \setminus \{l_i\}}in(l').in(b_{i'}^5).Ag_{i'}^5|R''$$

then:

$$V_{2n+1} \rightarrow S'|\prod_{l' \in L \setminus \{l_i\}}\langle l' \rangle|\prod_{l' \in L'}\langle b_{i'}^5 \rangle|\prod_{l' \in L'}in(l').in(b_{i'}^5).Ag_{i'}^5|R''' \quad (\overset{\text{def}}{=} V_{2n+2})$$

$$\rightarrow S'|\prod_{l' \in L \setminus \{l_i\}}\langle l' \rangle|\prod_{l' \in L'}\langle b_{i'}^5 \rangle|\prod_{l' \in L'}in(l').in(b_{i'}^5).Ag_{i'}^5|R''' \quad (\overset{\text{def}}{=} V_{2n+3})$$

$$\ldots$$

$$\rightarrow S'|R \quad (\overset{\text{def}}{=} V_{2n+4})$$

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

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

where $V_i \models W_i$. The consumption of the presence tokens in $2n$ steps ensures that $W_{4n+1} \equiv T'[R].$ Finally, observe that $S'|R \models T'[R]$ ensures $(S',T') \in \mathcal{R}$.

- $S \models a$, $S'$:
  - The proof is the same as the previous case with the only difference that the set $L'$ is defined as $L' \overset{\text{def}}{=} \{ b \mid T \rightarrow a \rightarrow T' \text{ and } b \neq a \}$. 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 $T \rightarrow \rightarrow T'$ or $T \rightarrow a \rightarrow T'$, where $(S',T') \in \mathcal{R}$.

\textbf{Corollary 4.5} $inp$-bisimulation is the coarsest congruence contained in the barbed bisimulation for $L[inp]$. 

UBLCS-97-6 17
\[
\begin{align*}
(16) \quad & \text{rdp}(a)? P \quad \xrightarrow{a} \quad P \\
(17) \quad & \text{rdp}(a)? P \quad \xrightarrow{\neg a} \quad Q
\end{align*}
\]

Table 5. Additional rules for L[rdp].

5 The Language L[rdp]

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

\[
C ::= \text{rdp}(a)? C \cdot C
\]

The process rdp(a)? P Q tests if the tuple (a) 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 defined: the action of reading the tuple (a) (and the following choice of the process P) is labeled with a (axiom (16)), while when the process Q is chosen the label is \(\neg a\) (axiom (17)). The semantics of the language L[rdp] is defined by the axioms and rules of Table 2 (plus the the side condition \(a \neq \bar{a} \rightarrow \neg a\) for the rule (8) and \(a \neq \neg a\) for the rule (6)), the rule (11) of Table 3 (defining the behavior of the synchronization due to the the labels \(\bar{a}\) and \(\overline{\text{p}}\), 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 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 \(\alpha \neq \bar{a} \rightarrow \neg a\), \(\tau\), 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 \(Q \xrightarrow{\bar{a}} Q', Q' \xrightarrow{\neg a} Q'', (P', Q') \in \mathcal{R}\) and \((P', Q'') \in \mathcal{R}\);
- if \(P \xrightarrow{\alpha} P'\), with \(\alpha = \bar{a} \rightarrow \neg a\), 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 \(Q \xrightarrow{\bar{a}} Q', Q' \xrightarrow{\neg a} Q'', (P', Q') \in \mathcal{R}\) and \((P', Q'') \in \mathcal{R}\).

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

A peculiar feature of rdp–bisimulation, is that a \(\tau\) can be matched with a \(a\) or a \(\neg a\) labeled derivation:

\[
\text{rdp}(a)? P \sim_{\text{rdp}} \tau.P
\]

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

UBLCS-97-6
Proposition 5.2. \textit{rdp}-bisimulation is a congruence for the operators of $L[rdp]$.

\begin{proof}
Let $P$ and $Q$ be agents of $L[rdp]$. If $P|\bar{R} \sim Q|\bar{R}$ for every agent $R$, then $P \sim_{rdp} Q$.

The relation $\sim_{rdp}$ is defined as in the proof of Proposition 2.6. The agent $R$ is the same used in the proof of Proposition 3.3.

The relation $\bar{R}$ is defined as in the proof of Proposition 2.6. Given the pair $(S,T) \in \bar{R}$ the analysis of the possible derivations of $S$ is the same for $\bar{S} \bar{\rightarrow} S'$ and $S \bar{\rightarrow} S'$. The cases $S \bar{\rightarrow} S'$, $S \bar{\rightarrow} S'$, and $S \bar{\rightarrow} S'$ are presented.

- $S \bar{\rightarrow} S'$:
  The proof is similar to the corresponding case in Section 4, but the set $S'$ must be defined as
  \[ S' \setminus \bar{R} = \{ b \mid T \bar{\rightarrow} T' \land S'R \sim T'|R \}. \]
  If $T \bar{\rightarrow} T'$ and also $T \bar{\rightarrow} T''$ with $(S'|R,T''|R) \in \bar{R}$, then if the tuples corresponding to the names in $S'$ are in $TS$, then the derivation labeled with $b$ 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):
  \[
  S|R \rightarrow \ldots \rightarrow S|\prod_{\ell \in L'} \langle l' \rangle | \prod_{\ell \in L'} \langle \ell \rangle | \prod_{\ell \in L} \text{in}(l'). \text{in}(\ell). A_g \bar{g}\bar{c} \ | R'' (\equiv V_{2n})
  \]
  and that the barbed equivalent agent $T|R$ must offer:
  \[
  T|R \rightarrow \ldots \rightarrow T|\prod_{\ell \in L'} \langle l' \rangle | \prod_{\ell \in L'} \langle \ell \rangle | \prod_{\ell \in L} \text{in}(l'). \text{in}(\ell). A_g \bar{g}\bar{c} \ | R'' (\equiv W_{2n})
  \]
  where $V_i \top W_i$.

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

\[
V_{2n} \rightarrow S'|\prod_{\ell \in L'} \langle l' \rangle | \prod_{\ell \in L'} \langle \ell \rangle | \prod_{\ell \in L} \text{in}(l'). \text{in}(\ell). A_g \bar{g}\bar{c} \ | R'' (\equiv V_{2n+1})
\]

The agent $W_{2n}$ must allow the reduction step $W_{2n} \rightarrow W_{2n+1}$ with $V_i \top W_i$. We have already shown that the subagent $R''$ and the agents $\text{in}(l'). \text{in}(\ell). A_g \bar{g}\bar{c}$ cannot be involved in this reduction step, and that the tuples $\langle l' \rangle$ cannot be consumed. Then derivations labeled with $b$ or $\bar{b}$ cannot be performed by $T$. Also derivations labeled with $\bar{c}$ cannot be performed. In fact, if $V_{2n} \bar{\rightarrow} W_{2n+1}$, then this derivation is inferred by $T \bar{\rightarrow} T'$. Reasoning as below, we can prove that $(S'|R,T'|R) \in \bar{R}$, hence $c \notin L'$ by definition of $L'$. Thus the tuple $\langle c \rangle$ is present in $V_{2n}$, hence the derivation $V_{2n} \bar{\rightarrow} W_{2n+1}$ is not possible.

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

As shown in Section 4 the tuples $\langle l' \rangle$ and the presence tokens $\langle \ell \rangle$ in $V_{2n+1}$ can be removed:

\[
V_{2n+1} \rightarrow \ldots \rightarrow S'|R (\equiv V_{2n+1})
\]

and also:

\[
W_{2n+1} \rightarrow \ldots \rightarrow W_{2n+1}
\]

with $V_i \top W_i$ and $W_{2n+1} \equiv T'|R$.
\end{proof}
• $S \xrightarrow{\alpha} S'$:
The proof is the same as the previous case with the only difference that the set $L'$ is defined as
$\{ c \mid T \xrightarrow{\alpha} T' \land c \neq a \land S' \cup T' \cup R \}$. 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 $T \xrightarrow{\alpha} T'$, and
$T \xrightarrow{\neg a} T'$ (with also $T \xrightarrow{\neg a} T''$ such as $(S', T'') \in R$), but also the derivation $T \xrightarrow{\neg a} T'$.
• $S \xrightarrow{\xi} S'$:
We consider $S \equiv S_1 \big| \bigoplus_{n \in \mathbb{N}} \langle a \rangle$ with $S_1 \xrightarrow{\xi}$. In this case we combine the techniques used in the
Sections 3 and 4.
Let $L' \overset{def}{=} \{ c \mid T \xrightarrow{\xi} T' \land c \neq a \land S' \cup T' \cup R \}$. 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 $R^\mu$ is the same as the one defined in Section 4):

$$S|R \rightarrow \ldots \rightarrow S_1 \big| \bigoplus_{n \in \mathbb{N}} \langle c, a \rangle| R \quad (\overset{def}{=} V_{4n})$$
$$\quad \rightarrow \ldots \rightarrow S_1 \big| \bigoplus_{n \in \mathbb{N}} \langle c, a \rangle \cup \bigoplus_{l \in L} \langle l', ij \rangle \big| \bigoplus_{l \in L} \langle l', ij \rangle \cdot \text{in}(l'), \text{in}(b_{ij}^5) \cdot \text{Ag}_5^a | R^\mu \quad (\overset{def}{=} V_{6n})$$

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

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

with $V_i \approx W_i$ and $W_{6n} \equiv T_1 \big| \bigoplus_{n \in \mathbb{N}} \langle c, a \rangle \cup \bigoplus_{l \in L} \langle l', ij \rangle \big| \bigoplus_{l \in L} \langle l', ij \rangle \cdot \text{in}(l'), \text{in}(b_{ij}^5) \cdot \text{Ag}_5^a | R^\mu$
where $T \equiv T_1 \big| \bigoplus_{n \in \mathbb{N}} \langle a \rangle$ and $T_1 \xrightarrow{\xi}$. As already observed in Section 3 the derivation $S_1 \xrightarrow{\alpha} S_1'$ holds, with $S' \equiv S_1 \big| \bigoplus_{n \in \mathbb{N}} \langle a \rangle$. The
agent $V_{6n}$ is now able to perform the following reductions. Let $R^\mu$ be the term:

$$\prod_{n \in \mathbb{N}} \langle c, a \rangle \big| \bigoplus_{l \in L} \text{in}(l'), \text{in}(b_{ij}^5) \cdot \text{Ag}_5^a \big| \bigoplus_{l \in L} \text{Ag}_5^a \big| \bigoplus_{l \in L} \text{Ag}_5^a \big| \bigoplus_{l \in L} \text{Ag}_5^a \big| \bigoplus_{l \in L} \text{Ag}_5^a$$

then:

$$V_{6n} \rightarrow S_1 \big| \bigoplus_{l \in L} \langle l', ij \rangle \big| \langle c, a \rangle \cdot \text{out}(b_{ij}^5) \cdot \text{in}(l'), \text{in}(b_{ij}^5) \cdot \text{Ag}_5^a | R^\mu \quad (\overset{def}{=} V_{6n+1})$$
$$\rightarrow S_1 \big| \bigoplus_{l \in L} \langle l', ij \rangle \big| \langle c, a \rangle \cdot \text{out}(b_{ij}^5) \cdot \text{in}(l'), \text{in}(b_{ij}^5) \cdot \text{Ag}_5^a | R^\mu \quad (\overset{def}{=} V_{6n+2})$$
$$\rightarrow S_1 \big| \bigoplus_{l \in L} \langle l', ij \rangle \big| \langle c, a \rangle \cdot \text{out}(b_{ij}^5) \cdot \text{in}(l'), \text{in}(b_{ij}^5) \cdot \text{Ag}_5^a | R^\mu \quad (\overset{def}{=} V_{6n+3})$$
$$\rightarrow S_1 \big| \bigoplus_{l \in L} \langle l', ij \rangle \big| \langle c, a \rangle \cdot \text{out}(b_{ij}^5) \cdot \text{in}(l'), \text{in}(b_{ij}^5) \cdot \text{Ag}_5^a | R^\mu \quad (\overset{def}{=} V_{6n+4})$$
$$\rightarrow S_1 \big| \bigoplus_{l \in L} \langle l', ij \rangle \big| \langle c, a \rangle \cdot \text{out}(b_{ij}^5) \cdot \text{in}(l'), \text{in}(b_{ij}^5) \cdot \text{Ag}_5^a | R^\mu \quad (\overset{def}{=} V_{6n+5})$$

This derivations are possible because the agent $Ag_5^a$ is not involved in the creation of the
tuples $\langle l' \rangle$, in fact $a \not\in L'$. Because of $V_{6n} \approx W_{6n}$, then also:

$$W_{6n} \rightarrow W_{6n+5}$$

with $V_i \approx W_i$. The presence token $\langle b_{ij}^5 \rangle$ ensures that (excluding the derivation $W_{6n+2} \rightarrow W_{6n+5}$) the agent $W_{6n}$ must reply with the same reductions performed by $Ag_5^a$. We now
analyze the derivation $W_{6n+2} \rightarrow W_{6n+5}$ which (as we have shown in Section 4) must be
inferred by the agent $T_1$. As above, the labels of the kind $b, b, c$ or $\neg c$ are not allowed. Hence,
the possible cases are $T \xrightarrow{\alpha} T'$ and $T \xrightarrow{\neg a} T'$. 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 $T \xrightarrow{\neg a} T''$
and $(S', T'') \in R$ (following from the definition of $L'$).
As shown in the Section 4 the presence tokens \( \langle b_i^5 \rangle \) in \( V_{2n+1} \) can be removed, and then (as shown in Section 3) the tuples \( \langle c_i \rangle \) can be renamed in \( \langle a \rangle \):

\[
V_{6n+5} \rightarrow \ldots \\
\rightarrow S'_i \prod a \langle c_i \rangle | R \quad (\overset{df}{=} V_{8n+5}) \\
\rightarrow \ldots \\
\rightarrow S' | R \quad (\overset{df}{=} V_{12n+5})
\]

But also:

\[
W_{6n+5} \rightarrow \ldots \rightarrow W_{12n+5}
\]

with \( V_i \sim W_i \) and \( W_{12n+5} \equiv T' | R \).

**Corollary 5.4** \( rdp \)-bisimulation is the coarsest congruence contained in the barbed bisimulation for \( L[rdp] \).

### 6 The Other Languages

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

---

**Table 6. Operational semantics of LINPA.**
6.1 Example

We present an example that shows a possible use of the \textit{rd} and \textit{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 $\langle \text{req}_i \rangle$. 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 $\text{yes}_i$ is sent to the client; if the client is not enabled, the answer $\text{no}_i$ is sent. When Client $i$ receives a $\text{no}_i$ 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 $\text{enable}_i$. If the index of the client is not in the set $V$, then it blocks waiting for the (never coming) answer of the Controller.

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 \textit{rdp}–bisimulation. This can be proved following the proofs presented in Section 5.

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

\begin{align*}
\text{Obs}(L) & = \sim \\
\text{Obs}(L[rd]) & = \sim_{rd} \\
\text{Obs}(L[inp]) & = \sim_{inp} \\
\text{Obs}(L[rd]) & = \text{Obs}(L[rd, inp]) = \text{Obs}(L[rd, rdp]) = \text{Obs}(L[rd, inp, rdp]) = \text{Obs}(\text{LINPA}) = \sim_{rdp}
\end{align*}

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

The following proposition compares the four bisimulations.

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

\begin{align*}
\sim & \subseteq \sim_{rd} \\
\sim & \subseteq \sim_{inp} \\
\sim_{rd} & \subseteq \sim_{rdp} \\
\sim_{inp} & \subseteq \sim_{rdp}
\end{align*}

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

\[ \text{rd}(a).P + \text{inp}(a)?Q \sim \text{rdp}(a)?P \sim \text{inp}(a).Q \]

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

\[ \text{rd}(a).P + \text{inp}(a)?Q \sim_{\text{rdp}} \tau.P + \text{inp}(a).Q \]

where both the terms do not contain the \( \text{rdp} \) primitive.

Observe that the four inclusions above are strict; moreover, only the \( \text{rd} \)-bisimulation and the \( \text{inp} \)-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 \( P, Q \in L' \), \( (P, Q) \in \text{Obs}(L') \) if and only if \( (P, Q) \in \text{Obs}(L'') \).

**Proof.** If \( \text{Obs}(L') \neq \text{Obs}(L'') \), then the difference consists of additional matchings in \( \text{Obs}(L'') \) 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 \( \text{rdp} \) – appear here for the first time.

(ii) The characterization of the four observational semantics for the eight different languages; in particular, two of them (\( \text{rd} \)-bisimulation and \( \text{rdp} \)-bisimulation) are completely new; for the other two semantics (classic bisimulation [Mil89] and \( \text{inp} \)-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 \( \text{L} \), it seems reasonable to claim that such a semantics (we denote \( \simeq \)) identifies, for example, the following processes:

\[ \text{out}(a).\text{out}(b) \overset{*}{\simeq} \text{out}(b).\text{out}(a) \]

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 \( \text{in}(a).\text{inp}(b)?0.\text{out}(c) \) distinguishes the agents above:

\[ \text{out}(a).\text{out}(b)|\text{in}(a).\text{inp}(b)?0.\text{out}(c) \neq \text{out}(b).\text{out}(a)|\text{in}(a).\text{inp}(b)?0.\text{out}(c) \]

In fact, the left-hand-side agent can generate the tuple \( (c) \), 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 \( \pi \)-calculus [MPW92]. In this approach, \( \text{out}(a) \) represents the message in TS; for instance, in \( \text{out}(a) \parallel P \) (the parallel composition of process \( P \) with \( \text{out}(a) \)), \( P \) can immediately input message \( a \). In other words, \( \text{out}(a) \parallel P \) is to be considered structurally congruent to \( \langle a \rangle P \). To obtain this, in the asynchronous \( \pi \)-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 \( \text{out}(a) \) 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 \( \text{out}(a) \) 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 \( \text{LINPA} \). Further operational semantics include [CY94], where a value-passing CCS with asynchronous communication is presented. The \( \text{out} \) 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 \( \tau \) labeled one), forcing a new notion of stable state, where not only \( \tau \) 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 \( \text{LINPA} \) 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 \( \pi \)-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 \( in(a).(a) \) in our semantics). Hence the law:

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

reminds the asynchronous law in [ACS96]:

\[
in(a).(a)|P| + \tau.P \sim_a \tau.P
\]

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


