Three Semantics of the Output Operation for Generative Communication*

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

[1] Dipartimento di Matematica, Università di Siena,
Via del Capitano 15, I-53100 Siena, Italy
e-mail: busi@cs.unibo.it

[2] Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy
e-mail: {gorrieri,zavattar}@cs.unibo.it

Abstract. A simple, yet Turing powerful, calculus based on generative communication is introduced; among its primitives, it contains a conditional input operation that tests for presence (or absence) of an output, reminiscent of the *inp* predicate of Linda. We study three different operational semantics for the output operation, called *instantaneous*, *ordered* and *unordered*. The associated behavioural semantics are obtained as the coarsest congruence contained in the corresponding strong barbed semantics. We prove that when the output operation is *instantaneous*, the obtained semantics is a sort of asynchronous bisimulation; on the contrary, for the *ordered* semantics, as well as for the *unordered* one, the resulting semantics is a small variant of the classic (synchronous) bisimulation. A further result is that the language under *unordered* semantics is no more Turing powerful, hence the language becomes strictly less expressive.

1 Introduction

Generative communication, realized by means of the insertion and withdrawal of elements from a shared multiset, is the peculiar feature of a family of coordination languages [GC92], of which Linda [Gel85] is the most prominent representative. This communication mechanism is based on the following principles: a sender communicates with a receiver through a shared data space (called *tuple space*, TS for short), where emitted messages are collected; the receiver can consume the message from TS; a message generated by a process has an independent existence in the tuple space until it is explicitly withdrawn by a receiver; in fact, after its insertion in TS, a message becomes equally accessible to all processes, but it is bound to none. Hence, the communication is asynchronous because the sender may proceed just after performing the emission of a message to the TS. Similarly, the receiver can input a message present in TS at any time: a hand-shake synchronization between TS and the receiver completes the communication

* Work supported by Esprit Working Group Coordina.

between the sender and the receiver, with the side-effect of removing the message from TS.

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

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

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

- *Instantaneous*: With $out(a)$ we mean that the message is already in the TS. Hence, $out(a).P = \langle a \rangle | P$, where $|$ is the parallel composition operator. For instance, consider a process P that wants to input a and a process $out(a)$; if composed in parallel, P can immediately input message a. This approach has been adopted in the asynchronous π-calculus [HT91, Bou92]; it is obtained by means of a simple syntactic restriction to that language: outputs cannot be used as prefixes. Consider processes $P = \mu.out(a_1)$ and $Q = \mu.(out(a_1)|out(a_2)| \ldots |out(a_n))$, where μ is any non output prefix; in *one* single *atomic* step, P executes action μ and puts $\langle a_1 \rangle$ in the TS, while Q executes action μ and puts messages $\langle a_1 \rangle, \ldots, \langle a_n \rangle$ in the TS.
- *Ordered*: The emission and the rendering of one message form together one single autonomous atomic action: $out(a).P$ becomes in one (internal) step the agent $\langle a \rangle | P$. In this way, the order of emission is respected by the rendering order. The implementation of $out(a)$ is simple: the sender sends the message a and then waits for the acknowledgement from the TS; hence, the emission of a message is realized by means of a synchronous hand–shake communication between the sender and the TS. This approach has already received an operational treatment in [HKH95] and [CGZ96].
- *Unordered*: The emission and the rendering of one message are distinct autonomous actions. Hence, $out(a).P$ emits message a becoming the agent $\langle\langle a \rangle\rangle | P$ in one (internal) step, where P is free to proceed, but message a is not yet present in the TS; indeed, $\langle\langle a \rangle\rangle$ takes one further internal step to become $\langle a \rangle$. The implementation of the $out(a)$ operation is trivial: the process $out(a).P$ sends the message a to the TS, and proceeds without waiting

for the message to reach the TS; hence, the emission is realized by means of an asynchronous communication between the sender and the TS. Thus the order of emission may not to be respected by the rendering order: for instance, if a process executes the sequence $out(a).out(b)$, then a may be rendered before or after the emission of b, or even after the rendering of b. To the best of our knowledge, we do not know of any other papers studying the semantics of this approach.

In Section 2 we introduce the syntax of our process algebra and the three operational semantics for the *out* primitive, comparing their differencies with one instructive example.

Section 3 studies the behavioural semantics for the three operational semantics, following the approach of [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).

The main results of the paper are that a variant of the asynchronous bisimulation [ACS96] is the right semantics for the instantaneous semantics, while the correct semantics for the other two cases is a variant of the classic (synchronous) bisimulation [Mil89], where inputs and outputs are treated symmetrically. All the resulting three congruences are different and not included one in the others.

Section 4 discusses the expressiveness of our calculus under the three semantics. We show that, for the instantaneous and ordered semantics, it is possible to encode a Random Access Machine (RAM) [SS63], a Turing equivalent formalism. On the contrary, it is possible to prove that our calculus is not Turing powerful under the unordered semantics [BGZ97b].

Finally, Section 5 reports some conclusive remarks.

2 The Language and its Operational Semantics

The syntax of our language is defined by the following grammar:

$$P ::= \langle a \rangle \ \mid \ C \ \mid \ P|P \ \mid \ P\backslash a$$
$$C ::= \mathbf{0} \ \mid \ out(a).C \ \mid \ in(a).C \ \mid \ inp(a)?C\_C \ \mid \ C|C \ \mid \ X \ \mid \ rec\,X.C$$

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

The coordination primitives $out(a)$ and $in(a)$ can be represented as usual prefixes, while $inp(a)$ requires a sort of *if-then-else* construct. In fact, $inp(a)?C\_D$

(i)	$P\|Q \equiv Q\|P$
(ii)	$(P\|Q)\|R \equiv P\|(Q\|R)$
(iii)	$P\|\mathbf{0} \equiv P$
(iv)	$rec\,X.P \equiv P[rec\,X.P/X]$

Table 1. Structural congruence.

is a program which requires the message a to be consumed; if a is present, it is removed and the program C is executed, otherwise D is chosen. Recursively defined programs are also considered, by allowing program variables (denoted by $X, Y, \ldots$) and by introducing an operator for closed and guarded recursion $rec\,X.C$. In the following *Agent* denotes the set containing all possible agents.

We present three different operational semantics for our language, one for each kind of output prefix introduced in the previous section. The semantics are presented in two steps. First, we define *structural congruences* over agents; this relations capture the fact that, for example, the order of the terms in a parallel composition has no effects on its behaviour. Next, we define *labeled transition systems* specifying how agents evolve by means of the actions performed by some program in it.

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

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

Axiom (1) shows that the tuple $\langle a \rangle$ is able to give its contents to the environment, by performing an action labeled with $\overline{a}$. Axioms (2) and (3) define the possible input actions on the message a (action labeled with a), according to the execution of an *in* or a successful *inp* operation, respectively. If a process executing an *inp* does not find the required message a, it can guess its absence by performing an action labeled with $\neg a$ (axiom (4)). Rule (5) states that no

(1)	$\langle a \rangle \xrightarrow{\overline{a}} \mathbf{0}$	(2)	$in(a).P \xrightarrow{a} P$
(3)	$inp(a)?P\_Q \xrightarrow{a} P$	(4)	$inp(a)?P\_Q \xrightarrow{\neg a} Q$
(5)	$\dfrac{P \xrightarrow{\alpha} P'}{P \backslash a \xrightarrow{\alpha} P' \backslash a} \quad \alpha \notin \{a, \overline{a}, \neg a\}$	(6)	$\dfrac{P \xrightarrow{\neg a} P'}{P \backslash a \xrightarrow{\tau} P' \backslash a}$
(7)	$\dfrac{P \xrightarrow{\neg a} P' \quad Q \stackrel{\overline{a}}{\not\longrightarrow}}{P \vert Q \xrightarrow{\neg a} P' \vert Q}$	(8)	$\dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \vert Q \xrightarrow{\tau} P' \vert Q'}$
(9)	$\dfrac{P \xrightarrow{\alpha} P'}{P \vert Q \xrightarrow{\alpha} P' \vert Q} \quad \alpha \neq \neg a$	(10)	$\dfrac{P \equiv Q \quad Q \xrightarrow{\alpha} Q' \quad P' \equiv Q'}{P \xrightarrow{\alpha} P'}$

Table 2. Operational semantics.

actions containing the name a can be performed by the agent $P \backslash a$. When an agent P willing to perform a $\neg a$ operation is restricted on the name a, its $\neg a$ operation becomes a local step of computation (i.e. labeled with τ) 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 (6)). On the other hand, if P is composed in parallel with another agent Q, the executability of $\neg a$ by $P|Q$ depends on the inability of Q to offer message a. Otherwise, the guess of P is wrong and this $\neg a$ operation cannot be executed (rule (7)). The other rules are the usual for synchronization between complementary actions (8), local actions in parallel composed agents (9), and possibility of executing the same actions for structurally congruent agents (10). There are no rules for recursion because its semantics is defined by the congruence rule (v) which applies one unfolding step to a recursively defined program.

Rule (7) 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.

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

Table 3. Three semantics for the *out* primitive.

The rules which differentiate the three semantics are presented in Table 3. Following the *instantaneous* approach messages have to be considered already available at the moment an output operation has to be performed. This is obtained by introducing a further rule for the structural congruence stating that a program starting with the prefix $out(a)$ is the same as putting the tuple $\langle a \rangle$ in parallel with the continuation of the program.

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

In the *unordered* approach, the execution of an output operation emitting the message a does not directly generate the corresponding tuple $\langle a \rangle$, but it creates an agent which will make message a available only after a non predictable delay. This agent, denoted by $\langle\langle a \rangle\rangle$,[3] is only able to perform an internal action labelled with τ becoming $\langle a \rangle$.

Example 1. An example, inspired by [SCA95], allows us to show the differences among the three semantics. Consider P and Q below where the only difference between them is the order of emission of the messages a and b:

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

The message names a and b are considered as local names in order to be sure that the *in* and *inp* operations on these names are executed locally.

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

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

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

[3] The syntax for the agents P is extended in the case of unordered output semantics by allowing P to be also the agent $\langle\langle a \rangle\rangle$.

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

	Instantaneous	Ordered	Unordered
P	C	C / D	C / D
Q	C	C	C / D

■

3 Behavioural Semantics

In this section the problem of defining observational semantics for the three different operational semantics is considered. The approach we follow consists in investigating the coarsest congruence contained in the *barbed bisimulation* [MS92] for the three semantics. In order to define barbed bisimulation we have to introduce the notion of *reduction* and *commitments*.

In our language, we consider as reductions not only the usual derivations labeled with τ, but also those labeled with $\neg a$. In fact, a derivation $P \xrightarrow{\neg a} P'$ indicates that P can become P' if no tuples $\langle a \rangle$ are available in the external environment. Hence, if P is stand–alone (i.e. without external environment), it can be considered able to become P'. Indeed, the label $\neg a$ has been introduced only for helping an SOS formulation of the semantics, while it is conceptually an internal step. Formally:

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

As in [ACS96] the notion of commitments is defined taking care that we are dealing with a calculus having asynchronous communication. The fact that an observer has no direct way of knowing if the message he has sent has been received is reflected by imposing that only *output* commitments are taken into account.

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

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

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

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

As already stated we investigate the coarsest congruence contained in the barbed bisimulation for the three semantics. In the *instantaneous* case we have obtained the following *asynchronous* $\neg$*-bisimulation*.

Definition 2. A binary, symmetric relation $\mathcal{R}$ on *Agent* is an *asynchronous $\neg$-bisimulation* if $(P,Q) \in \mathcal{R}$ implies:

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

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

In the standard bisimulation [Mil89], a step can be matched only by a step having the same label; in the asynchronous bisimulation [ACS96] an *in* step can be matched also by a τ labelled transition under some conditions. These conditions are the same presented in the final item of the definition of our asynchronous $\neg$-bisimulation; that is why we call it asynchronous. All the other new matchings allowed by the asynchronous $\neg$-bisimulation are due to the fact that we deal also with the new kind of label $\neg a$ (that is why we have introduced the symbol $\neg$ in its name). These matchings can be understood considering P and Q as below, where a is a name not appearing in R:

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

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

The agents P and Q cannot be distinguished under *instantaneous* semantics because they both perform an internal step of computation having no influence on the environment, and then become R.

In order to equate the agents above, the asynchronous $\neg$-bisimulation must introduce several new matchings. One of them, for example, allows a τ-step to be matched by a transition labeled with b. This is possible if the tuple $\langle b\rangle$ is generated immediately after having consumed it, and if an equivalent continuation can be chosen also if the message b is not present in the environment.

Proposition 3. *For the* instantaneous *semantics* $\sim_a$ *is a congruence.* ■

Theorem 4. *For the* instantaneous *semantics* $\sim_a$ *is the coarsest congruence contained in* $\stackrel{\bullet}{\sim}$.

Proof (hint). *As* $\sim_a$ *is a congruence, it is sufficient to show that for every* P *and* Q *barbed bisimilar in every context, there exists* $\mathcal{R}$, $\neg$*-bisimulation such that* $(P,Q) \in \mathcal{R}$. *Let* $n(S)$ *denote the message names appearing in the agent* S *and* $L = n(P) \cup n(Q)$. *We use* $\prod_{i\in\{1\ldots n\}} P_i$ *as a shorthand for* $P_1|\ldots|P_n$. *The* $\neg$*-bisimulation* $\mathcal{R}$ *is defined as:*

$$\mathcal{R} \;=\; \{(P,Q) \mid P|R \stackrel{\bullet}{\sim} Q|R \text{ and } n(P), n(Q) \subseteq L\}$$

where R *is the following context:*

$$\begin{aligned} R = &\textstyle\prod_{l\in L} rec\, X.in(l).out(a_l).in(a_l).X \;\mid \\ &\textstyle\prod_{l\in L} rec\, X.out(b_l).in(b_l).out(l).out(c_l).in(c_l).X \end{aligned}$$

where the names a_l, b_l, c_l *for all* $l \in L$ *are supposed to be distinct and fresh.* ∎

We now consider the *ordered* and *unordered* semantics. The barbed congruence is no more the asynchronous $\neg$-bisimulation, but a lot of possible matchings between *equivalent* steps are lost.

Definition 5. A binary, symmetric relation $\mathcal{R}$ on *Agent* is a $\neg$*-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 ($Q \xrightarrow{\neg a} Q'$ or $Q \xrightarrow{\tau} Q'$) and $(P',Q') \in \mathcal{R}$.

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

Proposition 6. *For* ordered *and* unordered *semantics* $\sim$ *is a congruence.* ∎

Theorem 7. *For the* ordered *and* unordered *semantics* $\sim$ *is the coarsest congruence contained in* $\stackrel{\bullet}{\sim}$.

Proof (hint). *The proofs are similar to the one of Theorem 4, with the context* R *defined in a different way:*

$$\begin{aligned} R = &\textstyle\prod_{l\in L} rec\, X.in(l).out(a_l).in(a_l).X \;\mid \\ &\textstyle\prod_{l\in L} rec\, X.inp(l)?out(b_l).X\_ \\ &\qquad\qquad out(l).out(c_l).rec\, Y.inp(b_l)?out(l).Y\_in(c_l).X \end{aligned}$$

∎

It is interesting to note that the restriction operator is not used in the definition of the context R in both the theorems, hence the results presented in Theorems 4 and 7 are correct also if the restriction operator is eliminated from the language.

A comparison among the equivalences

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

$P \sim_1 Q$ iff $P \sim_a Q$ in the *instantaneous* semantics
$P \sim_2 Q$ iff $P \sim Q$ in the *ordered* semantics
$P \sim_3 Q$ iff $P \sim Q$ in the *unordered* semantics

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

Example 2. Consider the following agents:

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

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

This example shows that $\sim_1 \not\subseteq \sim_2$, $\sim_1 \not\subseteq \sim_3$, and $\sim_2 \not\subseteq \sim_3$. The following example shows that also the inverse inclusions are false.

Example 3. In this example a τ prefix and an internal choice $\tau.P + \tau.Q$ operator are used. The agent P containing τ-prefixes and Q containing internal choices, are defined as follows, where a_i are considered to be distinct and fresh names:

$P \stackrel{def}{=} (P'|\langle a_1\rangle|\ldots|\langle a_n\rangle)\backslash a_1\backslash \ldots \backslash a_n$
where P' is obtained by substituting $in(a_i)$ for the i^{th} τ in P
$Q \stackrel{def}{=} (Q'|\langle a_1\rangle|\ldots|\langle a_n\rangle)\backslash a_1\backslash \ldots \backslash a_n$
where Q' is obtained by substituting $in(a_i).R_i + in(a_i).S_i$ for the i^{th} internal choice $\tau.R_i + \tau.S_i$ in Q

Consider the following agents:

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

It is not difficult to see that under the *unordered* semantics P and Q have the same possible computations; hence $P \sim_3 Q$. Instead, they can not be equivalent in the case of *instantaneous* or *ordered* semantics because $Q \Downarrow \overline{d}$ while $P \not\Downarrow \overline{d}$ in both the cases.

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

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

In the case of *ordered* semantics P and Q have the same possible computations; hence $P \sim_2 Q$. Instead, under *instantaneous* semantics P and Q can not be considered equivalent because $Q \Downarrow \overline{d}$ while $P \not\Downarrow \overline{d}$. ∎

4 Expressiveness of the Language

In this section we analyze the expressive power of our small calculus under the three possible semantics. We show that our language is Turing powerful (under the *instantaneous* and *ordered* semantics) because it is expressive enough to model a Random Access Machine (RAM) [SS63]. In the case of *unordered* semantics, the RAM is not implementable because the language is no more Turing powerful.

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 the last instruction of the program and this instruction does not require a jump, or if the current instruction requires a jump to an instruction number not appearing in the program. If the program terminates, the result of the computation is the content of the registers specified as outputs.

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

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

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

$1 : DecJump(r_2, 4)$
$2 : Succ(r_1)$
$3 : DecJump(r_3, 1)$

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

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

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

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

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

The unordered case

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

$$\begin{array}{ll} i & : Succ(r_j) \\ i+1 & : DecJump(r_j, s) \end{array}$$

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

We not only say that the implementation we have presented is not correct under *unordered* semantics, but we also assert that the RAM is not implementable in any way. In fact, under this semantics, the language is no more Turing powerful; in [BGZ97b] we show that the problem of termination is decidable under the unordered semantics. The proof is divided in two steps: we first define a net semantics in terms of contextual P/T nets (i.e., P/T nets extended with arcs testing for presence or absence of tokens in a place; see, e.g., [MR95, BP95]). This semantics, defined following the style of [BG95, BGZ97a], preserves the interleaving behaviour, hence also the possibility of deadlock. Then, given the contextual P/T net semantics, we present a mapping on finite (standard) P/T nets that preserves deadlock. As deadlock is decidable on finite P/T nets, we conclude that the termination problem is decidable under the unordered semantics.

5 Conclusion and Future Research

A formal semantics for three different interpretations of the output operation is studied, providing each of them with an operational and a corresponding behavioural semantics. We think this is a first necessary step in order to equip languages like Linda with a formal semantics, currently non-available.

In the reference manual [SCA95] it is often unclear which is the real interpretation of the *out* primitive. As an illustrative example, consider $Q = (out(b).out(a) \mid in(a).inp(b)?C\_D)$. If we assume the ordered semantics, then the input of a is possible only if $\langle b \rangle$ is already in TS; hence, the execution of *inp* will always enable C. Differently, if we assume the unordered semantics, then no guarantee is given that $\langle b \rangle$ is in TS; hence, it is sometimes possible that D is executed instead. The choice between the two semantics is not solved in [SCA95] and other similar publications (e.g., [Nar90]). For instance, on page 2-6 of [SCA95] we can read: "*out* returns after the tuple has been added to tuple space", hence supporting the claim that the intended semantics is ordered. On the contrary, on page 2-26 of [SCA95] a comment to a program reported on page 2-25 expresses a concern very similar to the above about the possible executability of D, hence validating that the intended semantics is the unordered one. Similar contradictions can be found in [Nar90]: on lines 19-20 of page 4 it is said that, in shared memory implementations, it is not ensured that "an out followed by a predicate operation on the same tuple will succeed", hence supporting that the intended semantics is unordered. Instead, on lines 7-9 of the same page we can read: "The time at which the tuple is visible to other processes is indeterminate. In the existing shared memory implementations, the operation is completed immediately.", hence validating the ordered semantics approach, at least for shared memory implementations.

The expressiveness of the three semantics has been also studied, at least in relation to the problem of Turing equivalence. However, we plan to study further this issue in order to provide formal support to the reasoning on the implementations of Linda-like languages. One relevant question is the following: is it possible to use one semantics as an implementation for another one? In this direction, we have already one result: the instantaneous and ordered semantics cannot be implemented in the unordered one (by contradiction, if it was possible, then the language under the unordered semantics would be Turing powerful).

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

The study carried out in this paper can be extended to cope with further coordination primitives, as, e.g., the *rd* and *rdp* Linda operators, that have been already studied in [BGZ97c], for the ordered case only.

References

[ACS96] R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. In Proc. *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.

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

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

[BGZ97b] N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing Equivalence of Linda Coordination Primitives. Technical Report UBLCS-97-05, Department of Computer Science, University of Bologna, May 1997.

[BGZ97c] N. Busi, R. Gorrieri, and G. Zavattaro, A Process Algebraic View of Linda Coordination Primitives. To appear in *Theoretical Computer Science*. Available as Technical Report UBLCS-97-06, Department of Computer Science, University of Bologna, May 1997.

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

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

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

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

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

[HKH95] M. Hansen, J. Kleist, and H. Hüttel. Bisimulations for Asynchronous Mobile Processes. In Proc. *Tbilisi Symposium on Language, Logic, and Computation*, 1995.

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

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

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

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

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

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

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

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