Electronic Notes in Theoretical Computer Science 7 (1997)
URL: http://www.elsevier.nl/locate/entcs/volume7.html 20 pages

# On the Turing Equivalence of Linda Coordination Primitives

Nadia Busi [a], Roberto Gorrieri [b] and Gianluigi Zavattaro [b]

[a] *Dipartimento di Matematica*
*Università di Siena*
*Siena, Italy*

[b] *Dipartimento di Scienze dell'Informazione*
*Università di Bologna*
*Bologna, Italy*

**Abstract**

We introduce a process algebra containing the coordination primitives of Linda (asynchronous communication via a shared data space, read operation, non-blocking test operators on the shared space). We compare two possible semantics for the output operation, following two different intuitions expressed in Linda reference manual [20]. The former, we call *ordered*, defines the output as an operation that returns when the message has reached the shared data space; the latter, we call *unordered*, returns just after sending the message to the tuple space. The process algebra under the ordered semantics is Turing powerful, as we are able to program a Random Access Machine. The main result of the paper is that the process algebra under the unordered semantics is not Turing powerful. This result is achieved by resorting to a net semantics in terms of contextual nets (P/T nets with inhibitor and read arcs), and showing that there exists a deadlock-preserving simulation of such nets by finite P/T nets (a formalism where termination is decidable).

## 1 Introduction

Linda [10], as well as many other coordination languages [11,2,8], is based on the principle of so-called *generative communication*: 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 read the message or even remove it from TS; a message generated by a process has an independent existence in the tuple space until it is explicitly withdrawn by a receiver; in fact, after its insertion in TS, a message becomes equally accessible to all processes, but it is bound to none.

Besides the non-blocking output operation $out(a)$ (that sends message $a$ to the tuple space), the blocking read operation $rd(a)$ (that succeeds only if message $a$ is in TS) and the blocking input operation $in(a)$ (that removes message $a$ from TS), Linda offers two further conditional input/read predicates called $inp(a)$ and $rdp(a)$, respectively. These predicates check 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, their behavior is the same as the $in/rd$ operation and the value $true$ is returned. We model the predicates of Linda by means of *if–then–else* constructs $inp(a)?P\_Q$ and $rdp(a)?P\_Q$. They are conditional instructions that direct the flow of control to $P$ or to $Q$ depending on the presence or absence of message $a$ in TS; if the message $a$ is present $inp(a)?P\_Q$ removes it before becoming $P$ while $rdp(a)?P\_Q$ does not. The test on the tuple and the removal of the message are performed by $inp()?\_$ in an atomic step.

The problem of defining a formal semantics for Linda has recently attracted some researchers (see, e.g., [6] and the references therein for an overview). One curious aspect is related to the semantics of the output operation $out(a)$, for which even the Linda reference manual [20] does not provide a unique interpretation. Conceptually, the execution of the Linda output primitive $out(a)$ can be seen as composed of two phases: the *emission* of the message $a$ (sending $a$ to the TS) and the *rendering* of $a$ (actual presence of message $a$ in TS, we denote with $\langle a\rangle$). At least the following two different semantics can be suggested:

- *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 $\langle a\rangle|P$, where $|$ is the parallel composition operator. 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 an acknowledgement from TS.
- *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 $out(a)$ in this case consists of a send operation of message $a$ to the TS, without waiting for any acknowledgement. In this way the order of emission may not 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 $b$.

As an illustrative example, consider $(out(b).out(a) \mid in(a).inp(b)?P\_Q)$. 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 $P$. 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 $Q$ is executed instead. The choice between the two semantics is not solved in [20] and other similar publications (e.g., [18]). For instance, on page 2-6 of [20] 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 [20] a comment to a program reported on page 2-25 expresses a concern very similar to the above about the possible executability of $Q$, hence validating that the intended semantics is the unordered one.

In this paper we present a process algebra $\mathbf{L}$ containing all the coordination primitives of Linda (with the exception of eval), together with parallel composition and a limited form of recursion (the so-called bang operator [15], guarded on inputs). We provide this language with both the ordered and unordered semantics, and we compare the expressive power of the two variations, called $\mathbf{L}_o$ and $\mathbf{L}_u$ respectively. Quite amazingly, the two semantics/languages possess a radically different expressive power; indeed, under the ordered semantics we get a Turing powerful language, while this is not the case under the unordered semantics.

Turing equivalence of $\mathbf{L}_o$ is proved by showing that it is possible to encode a Random Access Machine (RAM) [21]. On the contrary, the proof that $\mathbf{L}_u$ is not Turing equivalent is rather elaborate and constitutes the main achievement of this paper. The proof consists in showing that the problem of termination is decidable in $\mathbf{L}_u$. We proceed as follows:

- *Net semantics*: we define a net semantics for $\mathbf{L}_u$ in terms of contextual P/T nets, i.e., P/T nets with inhibitor as well as read arcs (see, e.g., [17,7]). This semantics, defined following the style of [3,4], preserves the interleaving behaviour, hence also the possibility of deadlock.
- *Deadlock-preserving simulation*: 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; hence $\mathbf{L}_u$ is not Turing powerful.

The paper is organized as follows. In Section 2 we introduce the syntax of language $\mathbf{L}$ and we report on the two operational semantics for the *out* primitive. Section 3 presents the RAM implementation in $\mathbf{L}_o$. Section 4 presents the proof of the non Turing equivalence of $\mathbf{L}_u$: basic definitions on P/T nets and contextual nets are recalled, the distributed semantics for $\mathbf{L}_u$ is presented, and finally a mapping on finite P/T nets that preserves deadlock is shown.

## 2 The Language and its Operational Semantics

The syntax of the language $\mathbf{L}$ is defined by the following grammar:

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

$$C ::= 0 \ \mid \ \eta.C \ \mid \ \mu?C\_C \ \mid \ C|C$$

where:

$$\eta ::= in(a) \ \mid \ rd(a) \ \mid \ out(a) \ \mid !in(a)$$

$$\mu ::= inp(a) \ \mid \ rdp(a)$$

Agents, ranged over by $P$, $Q$, ..., consist of the parallel composition of the messages already in TS (each one denoted by one agent $\langle a \rangle$ where $a$, $b$, ...,

Table 1
Operational semantics (symmetric rules omitted).

| | | | |
|---|---|---|---|
| (1) | $\langle a\rangle \xrightarrow{\overline{a}} 0$ | (2) | $in(a).P \xrightarrow{a} P$ |
| (3) | $rd(a).P \xrightarrow{\underline{a}} P$ | (4) | $!in(a).P \xrightarrow{a} P\|!in(a).P$ |
| (5) | $inp(a)?P\_Q \xrightarrow{a} P$ | (6) | $rdp(a)?P\_Q \xrightarrow{\underline{a}} P$ |
| (7) | $inp(a)?P\_Q \xrightarrow{\neg a} Q$ | (8) | $rdp(a)?P\_Q \xrightarrow{\neg a} Q$ |
| (9) | $\dfrac{P \xrightarrow{\alpha} P'}{P\|Q \xrightarrow{\alpha} P'\|Q} \quad \alpha \neq \neg a$ | (10) | $\dfrac{P \xrightarrow{\neg a} P' \quad Q \not\xrightarrow{\overline{a}}}{P\|Q \xrightarrow{\neg a} P'\|Q}$ |
| (11) | $\dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P\|Q \xrightarrow{\tau} P'\|Q'}$ | (12) | $\dfrac{P \xrightarrow{\underline{a}} P' \quad Q \xrightarrow{\overline{a}} Q'}{P\|Q \xrightarrow{\tau} P'\|Q}$ |

are used to range over the possible messages) and the concurrent programs denoted by $C$, $D$, ..., that share the tuples. A program can be a terminated program 0 (which is usually omitted for the sake of simplicity), a prefix form $\eta.P$, an *if–then–else* form $\mu?P\_Q$, or the parallel composition of programs. In the following prefix and if–then–else forms are also called *sequential programs* or *sequential subprograms* when they appear as subterms. A prefix $\eta$ can be one of the Linda primitives $in(a)$, $rd(a)$, or $out(a)$ indicating the withdrawing, the reading (without consumption) or the emission of message $a$, respectively. We also consider the bang operator $!in(a)$ which is a form of replication guarded on input operations: the term $!in(a).P$ is always ready to consume a message $a$ in TS and then activate a copy of the program $P$. The if–then–else forms are used to model the $inp(a)$ and $rdp(a)$ primitives: $inp(a)?P\_Q$ ($rdp(a)?P\_Q$) is a program which requires the message $a$ to be consumed (or simply read); if $a$ is present, the program $P$ is executed, otherwise $Q$ is chosen. In the following, *Agent* denotes the set containing all possible agents.

We present two different operational semantics (in terms of labeled transition systems) for our language, one according to the ordered interpretation of the *out* operator, and the other following the unordered interpretation presented in the previous section. The two variations are called $\mathbf{L}_o$ and $\mathbf{L}_u$, respectively.

The labeled transition systems are of the kind ($Agent$, $Label$, $\longrightarrow$) where $Label = \{\tau, a, \underline{a}, \overline{a}, \neg a\}$ (ranged over by $\alpha$, $\beta$, ...) is the set of the possible labels. The labeled transition relation $\longrightarrow_o$ for the *ordered* semantics is the smallest one satisfying the axioms and rules $(1)\ldots(12)$ in Table 1 and the axiom (13) of Table 2; while $\longrightarrow_u$ for the unordered semantics considers the axioms and rules $(1)\ldots(12)$, (14) and (15). For the sake of simplicity we omit the symmetric rules of $(9)\ldots(12)$. The indexes $o$ and $u$ (distinguishing the two variations of the language and the two transition relations) are omitted when no confusion arises.

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) describe the action performed by the prefixes $in(a)$ and $rd(a)$, whose labels are

Table 2
Two semantics for the *out* primitive.

| | | | |
|---|---|---|---|
| **Ordered:** | (13)$out(a).P \xrightarrow{\tau} \langle a \rangle \vert P$ | **Unordered:** | (14) $out(a).P \xrightarrow{\tau} \langle\langle a \rangle\rangle \vert P$ <br> (15) $\langle\langle a \rangle\rangle \xrightarrow{\tau} \langle a \rangle$ |

$a$ and $\underline{a}$, respectively. The term $!in(a).P$ is able to activate a new copy of the program $P$ by performing a derivation labeled with $a$ that requires a tuple $\langle a \rangle$ to be consumed (axiom (4)). Axioms (5) ... (8) describe the semantics of the two if–then–else forms $inp(a)?P\_Q$ and $rdp(a)?P\_Q$: if the required message $a$ is present, it can be consumed by the former (by means of a derivation labeled with $a$) or only read by the latter (by means of a derivation labeled with $\underline{a}$). In both cases, if the required tuple $\langle a \rangle$ is not found, its absence is guessed by performing an action labeled with $\neg a$ (axioms (7) and (8)). The usual rule (9) for the parallel operator can be applied only to the labels different from $\neg a$. In fact, when an agent $P$ willing to perform a derivation labeled with $\neg a$ is composed in parallel with another agent $Q$, the executability of $\neg a$ by $P|Q$ depends on the inability of $Q$ to offer the tuple $\langle a \rangle$. Otherwise, the guess of $P$ is wrong and the derivation labeled with $\neg a$ cannot be executed (rule (10)). Rule (11) is the usual one for synchronization between the complementary actions $a$ and $\overline{a}$, while the rule (12) represents a new form of synchronization between the labels $\underline{a}$ and $\overline{a}$ in which the agent performing the derivation labeled with $\overline{a}$ is left unchanged. In fact, a reading operation does not change the state of TS (i.e., it does not remove the read tuple).

The rules which differentiate the *ordered* from the *unordered* semantics are presented in Table 2. In the *ordered* approach the output operation consists of one internal move which creates the tuple $\langle a \rangle$. In this way, when a sequence of outputs is executed, the tuples are rendered in the same order they are emitted.

In the *unordered* approach, the execution of an output operation emitting the tuple with contents $a$ does not directly generate the corresponding term $\langle a \rangle$, but it creates an agent which will make the tuple $\langle a \rangle$ available only after a non predictable delay. This agent, denoted by $\langle\langle a \rangle\rangle$ (term added to the syntax of the language in the case of unordered semantics) represents the fact that the message $a$ has been sent to the TS but it has not been received yet. This behaviour is obtained by allowing $\langle\langle a \rangle\rangle$ to perform an internal step, becoming $\langle a \rangle$.

Note that rule (10) uses a negative premise; it is easy to see that our transition system specification is strictly stratifiable [13], thus there exists a unique transition system agreeing with it.

In the following we only consider computations consisting of reduction steps, i.e. the internal derivations that a *stand–alone* agent is able to perform independently of the context. In our language, we consider as reductions not only the usual derivations labeled with $\tau$, 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 only introduced for helping an SOS formulation of the semantics, but it is conceptually an internal step. Formally:

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

We use $P \longrightarrow^i P'$ to state that $P'$ can be reached from $P$ by means of $i$ reduction steps, while $P \longrightarrow^* P'$ is the reflexive and transitive closure of $\longrightarrow$ .

An agent $P$ is *dead* if $P \not\longrightarrow$ . As in process algebra there is no explicit notion of state, the only reasonable form of termination of an agent is the absence of outgoing transitions. Because of the presence of bang, it is senseless to distinguish between properly terminated (i.e. consisting of the parallel composition of agents 0) and dead agents. Indeed, each nontrivial (containing a bang) agent with no outgoing transitions, cannot be properly terminated.

In order to deal with the encoding of recursive functions in a nondeterministic setting, as a process algebra, we have to specify what do we mean for an agent to terminate. Following the approach of other formalisms, such as $\lambda$–calculus, (nondeterministic) Turing machines, phrase structure grammars, we say that an agent terminates if there exists a computation leading to a dead agent. Formally, the agent $P$ terminates if it *has a deadlock*, i.e., there exists a dead agent $Q$ such that $P \longrightarrow^* Q$.

## 3 $\mathbf{L}_o$ is Turing Powerful

We show that under the *ordered* semantics the language is Turing powerful because it is expressive enough to model any Random Access Machine (RAM) [21].

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 $r_1, \ldots, r_m$; if other registers $r_{m+1}, \ldots, r_n$ 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 contents of the registers specified as outputs.

In [16] it is shown that the following two instructions are sufficient to model every recursive function:

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

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

$1 : DecJump(r_2, 4)$

$2 : Succ(r_1)$

$3 : DecJump(r_3, 1)$

The state of the computation is represented by means of configurations $(i, c_1, c_2, \ldots, c_n)$ where $i$ indicates that the next instruction to execute is the $i^{th}$ and $c_l$ is the contents of the register $r_l$ for $l$ in $1 \ldots n$. Given a program $R$ and a configuration $(i, c_1, c_2, \ldots, c_n)$ we use the notation:

$$(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (j, c'_1, c'_2, \ldots, c'_n)$$

to state that after the execution of the $i^{th}$ instruction of the program $R$ with contents of the registers $c_1, \ldots, c_n$, the program counter points to the $j^{th}$ instruction, and the registers will contain $c'_1, \ldots, c'_n$.

To model a RAM in our language we present an encoding for the configurations and one for the programs. The program counter in the configuration $(i, c_1, c_2, \ldots, c_n)$ is represented by the *program counter tuple* $\langle p_i \rangle$ while the contents of register $r_l$ is modeled by $c_l$ occurrences of the tuple $\langle r_l \rangle$ (for $l = 1 \ldots n$):

$$[\![(i, c_1, c_2, \ldots, c_n)]\!] \stackrel{def}{=} \langle p_i \rangle | \underbrace{\langle r_1 \rangle | \ldots | \langle r_1 \rangle}_{c_1 \text{ times}} | \ldots | \underbrace{\langle r_n \rangle | \ldots | \langle r_n \rangle}_{c_n \text{ times}}$$

To model the program $R$ composed of the sequence of instructions $I_1 \ldots I_k$, we consider the parallel composition of $k$ agents, each one modeling an instruction. A $Succ$ instruction on register $r_j$ at position $i$ is represented by an agent that consumes the program counter tuple, adds a tuple $r_j$ and updates the program counter by adding a tuple $p_{i+1}$. 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 $\langle p_s \rangle$.

$$\begin{array}{ll} [\![R]\!] & \stackrel{def}{=} [\![I_1]\!] | \ldots | [\![I_k]\!] \\ [\![i\ :\ Succ(r_j)]\!] & \stackrel{def}{=} !in(p_i).out(r_j).out(p_{i+1}) \\ [\![i\ :\ DecJump(r_j, s)]\!] & \stackrel{def}{=} !in(p_i).inp(r_j)?out(p_{i+1})\,\_\,out(p_s) \end{array}$$

The use of the bang operator in the representation of the instructions permits to reuse them. It is interesting to note that each RAM instruction is realized in our implementation by means of three internal steps: the first consumes the program counter, the second updates/tests the contents of the registers, and the third introduces the new program counter tuple.

The agent modeling the program $R$ that starts its computation from the configuration $(1, c_1, c_2, \ldots, c_n)$ is $[\![R]\!] | [\![(1, c_1, c_2, \ldots, c_n)]\!]$.

**Theorem 3.1** *Let $R$ be a RAM program, then:*

- Soundness: *if $[\![R]\!] | [\![(i, c_1, c_2, \ldots, c_n)]\!] \longrightarrow^3 Q$ then there exists a unique configuration $(j, c'_1, c'_2, \ldots, c'_n)$ such that:*
  $Q = [\![R]\!] | [\![(j, c'_1, c'_2, \ldots, c'_n)]\!]$ *and* $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (j, c'_1, c'_2, \ldots, c'_n)$
- Completeness: *if $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (j, c'_1, c'_2, \ldots, c'_n)$ then also:*
  $[\![R]\!] | [\![(i, c_1, c_2, \ldots, c_n)]\!] \longrightarrow^3 [\![R]\!] | [\![(j, c'_1, c'_2, \ldots, c'_n)]\!]$

# 4 $\mathbf{L}_u$ is not Turing Powerful

The implementation of the RAM we have presented does not satisfy the soundness property of Theorem 3.1 under the unordered semantics. Consider an execution of the program:

$1 : Succ(r_1)$

$2 : DecJump(r_1, 1)$

starting with the register $r_1$ empty. The implementation of the $Succ(r_1)$ instruction creates two tuples: $\langle r_1 \rangle$ and the new program counter tuple $\langle p_2 \rangle$. If the tuple $\langle p_2 \rangle$ becomes available before $\langle r_1 \rangle$, the following $DecJump(r_1, 1)$ instruction could execute the jump because no tuple $\langle r_1 \rangle$ is available. Then, the computation gives rise to a loop that is exited only when the first tuple $\langle r_1 \rangle$ becomes available in TS. Instead, the expected behaviour of the RAM is that the program terminates with the register $r_1$ empty, after it has been incremented and then decremented.

We assert not only that the implementation we have presented for $\mathbf{L}_o$ is not sound under the *unordered* semantics, but we will also prove that no implementation of the RAM exists. In fact, in the remainder of this section we show that $\mathbf{L}_u$ is not Turing powerful. In order to prove this, we give a contextual P/T net semantics to $\mathbf{L}_u$ for which there exists a deadlock preserving mapping on finite (standard) P/T nets. As deadlock is decidable on these nets, we can conclude that the termination is decidable for $\mathbf{L}_u$. In the following sections we recall some definitions about contextual P/T nets, we present the net semantics for $\mathbf{L}_u$, and finally the corresponding deadlock preserving simulation on finite P/T nets.

## *4.1 Contextual P/T Nets*

We recall simple Place/Transition nets without capacity constraints on places (see, e.g., [12]). Then, we extend them with the contextual arcs (see, e.g., [7,17]). Here we provide a characterization of this model which is convenient for our aims.

**Definition 4.1** *Given a set $S$, a* finite multiset *over $S$ is a function $m : S \rightarrow I\!N$ such that the set $dom(m) = \{s \in S \mid m(s) \neq 0\}$ is finite. The* multiplicity *of an element $s$ in $m$ is given by the natural number $m(s)$. The set of all finite multisets over $S$, denoted by $\mathcal{M}_{fin}(S)$, is ranged over by $m$. A multiset $m$ such that $dom(m) = \emptyset$ is called* empty *. The set of all finite sets over $S$ is denoted by $\wp_{fin}(S)$.*

*Given the multiset $m$ and $m'$, we write $m \subseteq m'$ if $m(s) \leq m'(s)$ for all $s \in S$ while $\oplus$ denotes their* multiset union*: $m \oplus m'(s) = m(s) + m'(s)$. The operator $\setminus$ denotes* multiset difference*: $(m \setminus m')(s) =$ if $m(s) \geq m'(s)$ then $m(s) - m'(s)$ else $0$. The* scalar product*, $j \cdot m$, of a number $j$ with $m$ is $(j \cdot m)(s) = j \cdot (m(s))$.*

To lighten the notation, we sometimes use the following abbreviation. If $m$ is a multiset containing only one occurrence of an element $s$ (i.e., $dom(m) = \{s\}$ and $m(s) = 1$) we denote $m$ with only $s$. Multiset union is represented also with comma, i.e., $m, m' = m \oplus m'$. Let $m$ be a multiset over $S$ and $m'$ a multiset over $S \cup S'$, such that $(m'(s') = 0)$ for each $s' \in S'$; with abuse of notation, we sometimes use $m$ in place of $m'$, and vice versa.

**Definition 4.2** *A P/T system is a triple $N = (S, T, m_0)$ where $S$ is the set of* places*, $T \subseteq \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S)$ is the set of* transitions*, and $m_0$ is a finite multiset over the set $S$ of places. Finite multisets over the set $S$ of places are called* markings*; $m_0$ is called* initial marking*. Given a marking $m$ and a place $s$, we say that the place $s$ contains $m(s)$* tokens*. A P/T system is finite if both $S$ and $T$ are finite.*

*A transition $t = (c, p)$ is usually written in the form $c \longrightarrow p$. The marking $c$, usually denoted with ${}^\bullet t$, is called the* preset *of $t$ and represents the tokens to be* consumed*; the marking $p$, usually denoted with $t^\bullet$, is called the* postset *of $t$ and represents the tokens to be* produced*. A transition $t$ is* enabled *at $m$ if ${}^\bullet t \subseteq m$. The execution of a transition $t$ enabled at $m$ produces the marking $m' = (m \setminus {}^\bullet t) \oplus t^\bullet$. This is written as $m \xrightarrow{t} m'$ or simply $m \longrightarrow m'$ when the transition $t$ is not relevant. We use $\sigma, \tau$ to range over sequences of transitions; the empty sequence is denoted by $\varepsilon$; let $\sigma = t_1, \ldots, t_n$, we write $m \xrightarrow{\sigma} m'$ to mean the* firing sequence $m \xrightarrow{t_1} \cdots \xrightarrow{t_n} m'$.

*A marking $m$ is* dead *if $m \not\longrightarrow$ . The net $N = (S, T, m_0)$ has a* deadlock *if there exists a dead marking reachable from $m_0$. The deadlock problem for a net consists in deciding if it has a deadlock.*

*A transition $t$ is* live *if for each marking $m$ reachable from $m_0$ there exists a marking $m'$ reachable from $m$ such that $t$ is enabled at $m$. A net is* live *if each of its transitions is live. The liveness problem for a net consists in deciding if it is live.*

The deadlock problem for a finite P/T system is decidable; this can be proved by reducing it to the liveness problem, which is known to be decidable (see, e.g., [19]). The theorem below is a slight generalization of the result proved in [9] for nets where the preset and postset of transitions are sets.

**Theorem 4.3** *Deadlock is reducible to liveness.*

**Definition 4.4** *A* contextual *P/T system is a triple $N = (S, T, m_0)$ where $S$ is the set of places, $T \subseteq \mathcal{M}_{fin}(S) \times \wp_{fin}(S) \times \wp_{fin}(S) \times \mathcal{M}_{fin}(S)$ is the set of transitions, and $m_0$ is the* initial marking.

*A transition $t = (c, r, i, p)$ is usually written in the form $r, \not{i} : c \longrightarrow p$ where $r$ and $\not{i}$ are omitted when empty. The set $r$, denoted with $\hat{t}$, is called*

Table 3
Decomposition function.

| | | | |
|---|---|---|---|
| $dec(0)$ | $= \emptyset$ | $dec(\langle a \rangle)$ | $= \{\langle a \rangle\}$ |
| $dec(\langle\langle a \rangle\rangle)$ | $= \{\langle\langle a \rangle\rangle\}$ | $dec(\eta.P)$ | $= \{\eta.P\}$ |
| $dec(\mu?P\_Q)$ | $= \{\mu?P\_Q\}$ | $dec(P\|Q)$ | $= dec(P), dec(Q)$ |

Table 4
Net semantics.

| | |
|---|---|
| `in(a,P)` | $in(a).P, \langle a \rangle \longrightarrow dec(P)$ |
| `rd(a,P)` | $\langle a \rangle : rd(a).P \longrightarrow dec(P)$ |
| `out(a,P)` | $out(a).P \longrightarrow \langle\langle a \rangle\rangle, dec(P)$ |
| `rend(a)` | $\langle\langle a \rangle\rangle \longrightarrow \langle a \rangle$ |
| `inp+(a,P,Q)` | $inp(a)?P\_Q, \langle a \rangle \longrightarrow dec(P)$ |
| `rdp+(a,P,Q)` | $\langle a \rangle : rdp(a)?P\_Q \longrightarrow dec(P)$ |
| `inp-(a,P,Q)` | $\langle \not{a} \rangle : inp(a)?P\_Q \longrightarrow dec(Q)$ |
| `rdp-(a,P,Q)` | $\langle \not{a} \rangle : rdp(a)?P\_Q \longrightarrow dec(Q)$ |
| `!in(a,P)` | $!in(a).P : \langle a \rangle \longrightarrow dec(P)$ |

*the* contextual set *of $t$ and represents the tokens to be* tested for presence*; the set $i$, denoted with $^\circ t$, is called the* inhibitor set *of $t$ and represents the tokens to be* tested for absence*; markings $c$ and $p$ are as above. This changes the definition of enabling: a transition $t$ is enabled at $m$ if $^\bullet t \oplus \hat{t} \subseteq m$ and $dom(m) \cap {}^\circ t = \emptyset$.*

*The execution of a transition $t$ enabled at $m$ producing the marking $m'$, written $m \xrightarrow{t} m'$, is defined as above.*

### *4.2 Net Semantics*

The basic idea underlying the definition of an operational net semantics for a process algebra is to decompose a process $P$ into a multiset of sequential components, which can be thought of as running in parallel. Each sequential component has a corresponding place in the net, and will be represented by a token in that place. Reductions are represented by transitions which consume and produce multisets of sequential components if the contextual conditions are satisfied.

We extend the approach to our language by representing tuples as tokens in the corresponding places in the net; in this way we can faithfully model the *rd* operation on a tuple by a transition with a contextual arc that tests for presence of a token in the tuple place. In the representation of the *inp* and *rdp* predicates, we use inhibitor arcs that test a tuple place for absence (of tokens) in the transitions representing the selection of the else branch.

Given the agent $P$, we define the corresponding contextual P/T system $Net(P)$. In order to do this, we need the following notations.

- Let $\mathcal{S}$ be the set $\{P \mid P$ is a sequential program$\} \cup \{\langle a \rangle, \langle\langle a \rangle\rangle \mid a$ is a message name$\}$.
- Let the function $dec : Agent \to \mathcal{M}_{fin}(\mathcal{S})$ be the decomposition of agents into markings, reported in Table 3.
- Let $\mathcal{T}$ contain the transitions obtained as instances of the axiom schemata presented in Table 4.

The axioms in Table 3, describing the decomposition of agents, state that the agent 0 generates no tokens; the decomposition of the terms $\langle a \rangle$, $\langle\langle a \rangle\rangle$, and of the sequential programs produces one token in the corresponding place; while the parallel composition is interpreted as multiset union, i.e, the decomposition of $P|Q$ is $dec(P), dec(Q)$.

The axioms in Table 4 deal with the possible transitions. Axioms `in(a,P)` and `rd(a,P)` deal with the execution of the primitives $in(a)$ and $rd(a)$: in the first case a token from place $\langle a \rangle$ is consumed, while in the second case it is only tested for presence. Axiom `out(a,P)` describes how a new token in place $\langle\langle a \rangle\rangle$ can be generated by the output prefix $out(a)$; this new token can be consumed (according to axiom `rend(a)`) to generate a new one in place $\langle a \rangle$. The axioms `inp+(a,P,Q)` and `rdp+(a,P,Q)` are similar to the corresponding axioms `in(a,P)` and `rd(a,P)`. In the axioms `inp-(a,P,Q)` and `rdp-(a,P,Q)`, tuple $\langle a \rangle$ is tested for absence instead: if it is not present, the agent $Q$ is selected. Finally, axiom `!in(a,P)` deals with the bang operator: if a token is present in place $!in(a).P$ and a token can be consumed from place $\langle a \rangle$, then a new copy of the term $P$ is produced.

With the name of a transition without parameters we denote any instance of that transition, i.e. we use `in` to mean any instance of `in(a,P)`.

**Definition 4.5** *Let $P$ be an agent. We define the triple $Net(P)$ where:*

$$S = \{Q \mid Q \textit{ is a sequential subprogram of } P\} \cup \{\langle a \rangle, \langle\langle a \rangle\rangle \mid a \textit{ is a message name in } P\}$$

$$T = \{r, \not{i} : c \longrightarrow p \in \mathcal{T} \mid \exists\, Q \textit{ sequential subpr. of } P \textit{ s.t. } Q \in dom(c) \cup r\} \cup \{\texttt{rend(a)} \mid a \textit{ is a message name in } P\}$$

$$m_0 = dec(P)$$

It is easy to see that given $P$, the tuple $Net(P) = (S, T, m_0)$ is a contextual P/T system. In fact, the domain of the initial marking is a (finite) multiset on the set of places, i.e., $dom(dec(P)) \subseteq S$. Moreover, for each sequential subprogram $Q \in S$ reachable from $P$, any possible transition $t$, in which $Q$ is involved, is in $T$ and also the places in ${}^{\bullet}t$, $t^{\bullet}$, ${}^{\circ}t$, and $\hat{t}$ are in $S$ (in fact, if $t$ is the transition $r, \not{i} : c \longrightarrow p \in \mathcal{T}$, the condition $Q \in dom(c) \cup r$ ensures also $r \cup i \cup dom(c) \cup dom(p) \subseteq S$). In addition, $Net(P)$ is finite as the sets $S$ and $T$ are both finite.

**Proposition 4.6** *Given the agent $P$, $Net(P)$ is a finite contextual P/T system.*

The following theorem states that our net translation is *sound* and *complete* with respect to the reduction semantics introduced in the previous section.

**Theorem 4.7** *Let $P$ be an agent.*

- Soundness: *if* $P \longrightarrow P_1 \ldots \longrightarrow P_n$ *then* $dec(P) \longrightarrow dec(P_1) \ldots \longrightarrow dec(P_n)$ *in* $Net(P)$
- Completeness: *if* $dec(P) \longrightarrow m_1 \ldots \longrightarrow m_n$ *in* $Net(P)$ *then there exist* $P_1 \ldots P_n$ *such that* $P \longrightarrow P_1 \ldots \longrightarrow P_n$ *and* $dec(P_i) = m_i$ *for* $i = 1 \ldots n$.

The above theorem ensures that the net semantics preserves deadlock, in the sense that a process $P$ has a deadlock iff $Net(P)$ has a deadlock.

### *4.3 Deadlock is decidable*

Let $P$ be a process. We show that there exists a finite P/T system $DNet(P)$, which is equivalent to $Net(P)$ w.r.t. deadlock. In particular, given $Net(P)$, we show how to transform it in the net $DNet(P)$, without contextual arcs, respecting the behaviour w.r.t. deadlock.

It is well known that the firing sequences are respected by replacing each read arc with a self loop.

Regarding inhibitor arcs, the first attempt is to remove all of them, replacing e.g. the transition `inp-(a,P,Q)` with
`(newinp-(a,P,Q))` $\quad inp(a)?P\_Q \longrightarrow dec(Q)$.
This transformation seems to work, because of the structure of the net we are studying: the only places tested for absence are $\langle a\rangle$, and the only transition producing tokens in $\langle a\rangle$ is `rend(a)`; moreover, transition `rend(a)` performs very smooth transformations on the marking, leaving unchanged the places corresponding to sequential subprograms. For this reason, it seems possible to move some `rend` transitions in a firing sequence, without compromising its firability, and reaching the same marking: consider a firing sequence where some transitions are fired even if their inhibiting place is not empty; we can obtain a legal firing sequence, leading to the same marking, by delaying all `rend` transitions that produce tokens in the inhibiting places mentioned above.

However, the following two problems arise:

- Let $Div$ be a divergent agent, e.g. $Div = out(c).!in(c).out(c)$. Consider the agent $Q = \langle a\rangle | inp(a)?Div\_0$; in the net $Net(Q)$ the only firable transition is `inp+(a,`$Div$`,0)`, leading to the marking $dec(Div)$; it is easy to see that no dead marking is reachable from it. In the corresponding P/T net, the transition `newinp-(a,`$Div$`,0)` can fire, leading to the dead marking containing only one token in $\langle a\rangle$; hence the behaviour w.r.t. deadlock is not preserved.

  This is due to the fact that place $\langle a\rangle$ contains some tokens at the initial marking; as these tokens are not produced by the occurrence of a transition `rend(a)`, we cannot apply the trick described above to obtain a legal firing sequence, because all previously reached markings already contain tokens in $\langle a \rangle$. To apply the trick, we have to wait that all tokens in $\langle a \rangle$ at the initial marking are consumed; at this point, if the current marking contains $k$ tokens in place $\langle a \rangle$, then at least $k$ transitions `rend(a)` have occurred.

We solve the problem by adding an auxiliary place $be_a$, whose contents represents the number of tokens consumed from place $\langle a \rangle$; this place is used by transitions inhibited by $\langle a \rangle$, to ensure that all the tokens in the initial marking have been consumed. Place $be_a$ is empty in the initial marking and is filled with a token by each transition consuming a message $a$. Let $\#_a(P)$ be the number of tuples $\langle a \rangle$ available in the agent $P$; we add a self loop on $be_a$, with arc weights $\#_a(P)$, to all transitions inhibited by $\langle a \rangle$.

- Another problem arises with transitions testing for presence of a token in $\langle a \rangle$. Consider the agent $R = out(a)|rd(a).out(b)|in(b).inp(a)?Div\_0$; the only firable sequence in $Net(R)$ is: $\sigma =$ `out(a,0) rend(a) rd(a,`$out(b)$`) out(b,0) rend(b) in(b,`$inp(a)?Div\_0$`)`. After its firing, a token is contained in $\langle a \rangle$, hence the only firable transition is `inp+(a,`$Div$`,0)`, leading to the marking $dec(Div)$, from which no dead marking is reachable. In the corresponding P/T net, after the firing of $\sigma$ also `newinp-(a,`$Div$`,0)` can fire, leading to the dead marking with a unique token, contained in place $\langle a \rangle$.

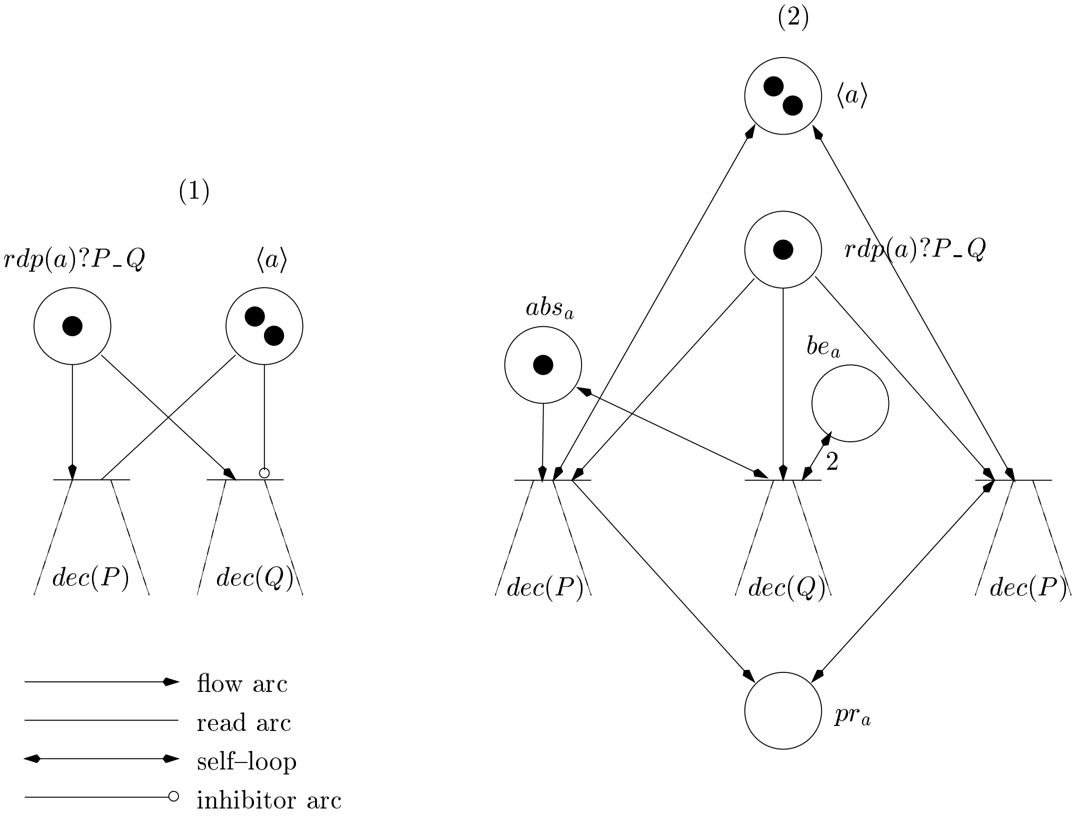
  This is due to the fact that if a transition, testing without consuming a token in $\langle a \rangle$, happens, we are sure that at least one token is contained in $\langle a \rangle$ in the reached marking; thus we should not fire transition `newinp-(a,P,Q)` until that token is consumed by some transition.

  In terms of a firing sequence, suppose it leads to a marking where place $\langle a \rangle$ contains a single token; suppose also that the transition $t =$ `rend(a)` producing it occurs before $u =$ `rd(a,P)`, which in turn is not followed by any transition removing tokens from $\langle a \rangle$. Then the token tested for presence by $u$ is exactly the one produced by $t$, hence the sequence obtained by removing $t$ is not firable.

  We solve the problem by adding two auxiliary places, $ab_a$ and $pr_a$. A token in $pr_a$ denotes the fact that we are sure that at least one token is contained in $\langle a \rangle$. $ab_a$ is the complementary place of $pr_a$, i.e. it contains a token iff $pr_a$ is empty; it is used to enable each transition in the P/T net corresponding to a transition inhibited by place $\langle a \rangle$ in $Net(P)$. The sum of tokens in $pr_a$ and $ab_a$ is equal to 1 in each reached marking. The correct information in these places is maintained by transitions testing or consuming tokens from $\langle a \rangle$: after a transition testing without consuming, we are sure that at least one token is present in $\langle a \rangle$; after a consuming transition, we are no longer sure of this fact, i.e. $\langle a \rangle$ could be empty. We split each of these transitions into two transitions, according to the two possible contents of the auxiliary places: the $pr_a$-version of the consuming transition removes the token from $pr_a$ and puts a token in $ab_a$, whereas the $ab_a$-version has a self loop on place $ab_a$; the $pr_a$-version of the testing transition has a self loop on place $pr_a$, whereas the $ab_a$-version removes the token from $ab_a$ and puts a token in $pr_a$.

Fig. 1 presents the contextual P/T net for the agent $rdp(a)?P\_Q|\langle a \rangle|\langle a \rangle$

Fig. 1. (1) $Net$ and (2) $DNet$ of the agent $rdp(a)?P\_Q|\langle a\rangle|\langle a\rangle$.




and the corresponding P/T system, without inhibitor and contextual arcs, obtained by following the above observations. In the figure the number 2 beside the arc connected to the place $be_a$ represents the weight of the arc.

Putting together the transformations listed above, we obtain the P/T net described below, after introducing some auxiliary notation.

- With $\#_a(P)$ we denote the number of tuples $\langle a\rangle$ available in the term $P$:

$$\begin{array}{ll} \#_a(Q) & = 0 \quad \text{if } Q \text{ is a sequential program} \\ \#_a(\langle a\rangle) & = 1 \\ \#_a(\langle\langle a\rangle\rangle) & = 0 \\ \#_a(Q_1|Q_2) & = \#_a(Q_1) + \#_a(Q_2) \end{array}$$

- $\mathcal{T}'$ contains all the transitions obtained as instances of the axiom schemata presented in Table 5. This set is parametric w.r.t. the number of initially available tuple $\langle a\rangle$, denoted by $\#_a$.

**Definition 4.8** *Let $P$ be an agent. We define the triple $DNet(P) = (\tilde{S}, \tilde{T}, \tilde{m}_0)$, where:*

Table 5
P/T net semantics that preserves deadlock.

| | |
|---|---|
| $\mathtt{In}_{pr}\mathtt{(a,P)}$ | $in(a).P,\ \langle a\rangle,\ pr_a \longrightarrow dec(P),\ be_a,\ ab_a$ |
| $\mathtt{In}_{ab}\mathtt{(a,P)}$ | $in(a).P,\ \langle a\rangle,\ ab_a \longrightarrow dec(P),\ be_a,\ ab_a$ |
| $\mathtt{Rd}_{pr}\mathtt{(a,P)}$ | $rd(a).P,\ \langle a\rangle,\ pr_a \longrightarrow dec(P),\ \langle a\rangle,\ pr_a$ |
| $\mathtt{Rd}_{ab}\mathtt{(a,P)}$ | $rd(a).P,\ \langle a\rangle,\ ab_a \longrightarrow dec(P),\ \langle a\rangle,\ pr_a$ |
| $\mathtt{Out(a,P)}$ | $out(a).P \longrightarrow \langle\langle a\rangle\rangle,\ dec(P)$ |
| $\mathtt{Rend(a)}$ | $\langle\langle a\rangle\rangle \longrightarrow \langle a\rangle$ |
| $\mathtt{Inp+}_{pr}\mathtt{(a,P)}$ | $inp(a)?P\_Q,\ \langle a\rangle,\ pr_a \longrightarrow dec(P),\ be_a,\ ab_a$ |
| $\mathtt{Inp+}_{ab}\mathtt{(a,P)}$ | $inp(a)?P\_Q,\ \langle a\rangle,\ ab_a \longrightarrow dec(P),\ be_a,\ ab_a$ |
| $\mathtt{Rdp+}_{pr}\mathtt{(a,P)}$ | $rdp(a)?P\_Q,\ \langle a\rangle,\ pr_a \longrightarrow dec(P),\ \langle a\rangle,\ pr_a$ |
| $\mathtt{Rdp+}_{ab}\mathtt{(a,P)}$ | $rdp(a)?P\_Q,\ \langle a\rangle,\ ab_a \longrightarrow dec(P),\ \langle a\rangle,\ pr_a$ |
| $\mathtt{Inp-(a,P,Q)}$ | $inp(a)?P\_Q,\ \#_a\cdot be_a,\ ab_a \longrightarrow dec(Q),\ \#_a\cdot be_a,\ ab_a$ |
| $\mathtt{Rdp-(a,P,Q)}$ | $rdp(a)?P\_Q,\ \#_a\cdot be_a,\ ab_a \longrightarrow dec(Q),\ \#_a\cdot be_a,\ ab_a$ |
| $\mathtt{!In}_{pr}\mathtt{(a,P)}$ | $!in(a).P,\ \langle a\rangle,\ pr_a \longrightarrow dec(P),\ !in(a).P,\ ab_a,\ be_a$ |
| $\mathtt{!In}_{ab}\mathtt{(a,P)}$ | $!in(a).P,\ \langle a\rangle,\ ab_a \longrightarrow dec(P),\ !in(a).P,\ ab_a,\ be_a$ |

$$
\begin{aligned}
\tilde{S} &= \{Q \mid Q \textit{ is a sequential subprogram of } P\} \cup \\
&\quad \{\langle a\rangle, \langle\langle a\rangle\rangle, be_a, pr_a, ab_a \mid a \textit{ is a message name in } P\} \\
\tilde{T} &= \{c \longrightarrow p \in \mathcal{T}' \mid \exists\ Q \textit{ sequential subprogram of } P \textit{ s.t. } Q \in dom(c)\} \cup \\
&\quad \{\mathtt{Rend(a)} \mid a \textit{ is a message name in } P\} \\
\tilde{m}_0 &= dec(P) \oplus \{ab_a \mid a \textit{ is a message name in } P\}
\end{aligned}
$$

Also in this case $DNet(P)$ is well defined, as it is a P/T system; moreover it is finite.

**Proposition 4.9** *Given the agent $P$, $DNet(P)$ is a finite P/T system.*

Now we are ready to show that $Net(P)$ and $DNet(P)$ have the same behaviour w.r.t. deadlock. The proof proceeds as follows: first we show that to a dead marking in $Net(P)$ corresponds a dead marking in $DNet(P)$, coinciding with the first one on common places; then the vice versa. Both proofs make use of results relating firing sequences of $Net(P)$ with firing sequences in $DNet(P)$, and vice versa.

The following function, parametric w.r.t. $\{pr, ab\}$, maps transitions in

$Net(P)$ to transitions in $DNet(P)$:

$$F_{pr}(\texttt{rend(a)}) = F_{ab}(\texttt{rend(a)}) = \texttt{Rend(a)}$$
$$F_{pr}(\texttt{out(a,P)}) = F_{ab}(\texttt{out(a,P)}) = \texttt{Out(a,P)}$$
$$F_{pr}(\texttt{inp-(a,P,Q)}) = F_{ab}(\texttt{inp-(a,P,Q)}) = \texttt{Inp-(a,P,Q)}$$
$$F_{pr}(\texttt{rdp-(a,P,Q)}) = F_{ab}(\texttt{rdp-(a,P,Q)}) = \texttt{Rdp-(a,P,Q)}$$

$$\begin{array}{ll}
F_{pr}(\texttt{in(a,P)}) = \texttt{In}_{pr}\texttt{(a,P)} & F_{ab}(\texttt{in(a,P)}) = \texttt{In}_{ab}\texttt{(a,P)} \\
F_{pr}(\texttt{rd(a,P)}) = \texttt{Rd}_{pr}\texttt{(a,P)} & F_{ab}(\texttt{rd(a,P)}) = \texttt{Rd}_{ab}\texttt{(a,P)} \\
F_{pr}(\texttt{inp+(a,P)}) = \texttt{Inp+}_{pr}\texttt{(a,P)} & F_{ab}(\texttt{inp+(a,P)}) = \texttt{Inp+}_{ab}\texttt{(a,P)} \\
F_{pr}(\texttt{rdp+(a,P)}) = \texttt{Rdp+}_{pr}\texttt{(a,P)} & F_{ab}(\texttt{rdp+(a,P)}) = \texttt{Rdp+}_{ab}\texttt{(a,P)} \\
F_{pr}(\texttt{!in(a,P)}) = \texttt{!In}_{pr}\texttt{(a,P)} & F_{ab}(\texttt{!in(a,P)}) = \texttt{!In}_{ab}\texttt{(a,P)}
\end{array}$$

The reverse mapping is defined as $G(\tilde{t}) = t$ iff $(F_{pr}(t) = \tilde{t}) \vee (F_{ab}(t) = \tilde{t})$. $G()$ is extended to sequences of transitions by elementwise application.

We define some sets of transitions that will be useful in the following: $IN(a)$ and $RD(a)$ are the sets of transitions (in nets $Net(P)$) consuming (resp. testing for presence of) tokens from place $\langle a \rangle$; $\tilde{IN}(a)$ and $\tilde{RD}(a)$ are the corresponding sets for nets $DNet(P)$.

$$\begin{aligned}
IN(a) &= \{\texttt{in(a,P)}, \texttt{inp+(a,P)}, \texttt{!in(a,P)} \mid P \text{ is a sequential program}\} \\
RD(a) &= \{\texttt{rd(a,P)}, \texttt{rdp+(a,P)} \mid P \text{ is a sequential program}\} \\
\tilde{IN}(a) &= \{F_{pr}(t), F_{ab}(t) \mid t \in IN(a)\} \\
\tilde{RD}(a) &= \{F_{pr}(t), F_{ab}(t) \mid t \in RD(a)\}
\end{aligned}$$

The transitions in $IN(a)$ and $\tilde{IN}(a)$ are called in-transitions; the ones in $RD(a)$ and $\tilde{RD}(a)$ rd-transitions.

The proposition below provides some relation between two corresponding transitions in $\mathcal{T}$ and $\mathcal{T}'$:

(i) the changes produced by the firing of two corresponding transitions on respective markings are the same on common places;

(ii) the second item will be used to relate the firability of two corresponding transitions in markings agreeing on common places.

**Proposition 4.10** *Let $t \in \mathcal{T}$, $\tilde{t} \in \mathcal{T}'$ and $t = G(\tilde{t})$ (or, equivalently, $\tilde{t} = F_{ab}(t)$ or $\tilde{t} = F_{pr}(t)$).*

*(i)* $(\tilde{t}^\bullet \setminus {}^\bullet\tilde{t})(s) = (t^\bullet \setminus {}^\bullet t)(s)$ *for all* $s \in \mathcal{S}$

*(ii) if $t$ is not an instance of* `inp-` *or* `rdp-` *then*

- ${}^{\bullet}\tilde{t}(s) = ({}^{\bullet}t \oplus \hat{t})(s)$ *for all* $s \in \mathcal{S}$
- ${}^{\circ}t = \emptyset$

The following proposition enunciates some general properties about reachable markings in $DNet(P)$, telling that places $be_a$, $ab_a$ and $pr_a$ behave as expected:

(i) the number of tokens in place $be_a$ corresponds to the number of occurred in-transitions;

(ii) the sum of tokens in places $ab_a$ and $pr_a$ is always equal to 1;

(iii) place $ab_a$ contains a token iff each rd-transition is followed by an in-transition.

**Proposition 4.11** *Let* $P$ *be an agent and* $DNet(P) = (\tilde{S}, \tilde{T}, \tilde{m}_0)$.
*If* $\tilde{m}_0 \xrightarrow{\tilde{t}_1} \tilde{m}_1 \ldots \xrightarrow{\tilde{t}_n} \tilde{m}_n$ *then*

*(i)* $\tilde{m}_n(be_a) = |\{k \mid 1 \leq k \leq n \wedge \tilde{t}_k \in \tilde{IN}(a)\}|$

*(ii)* $\tilde{m}_n(pr_a) + \tilde{m}_n(ab_a) = 1$

*(iii)* $\tilde{m}_n(ab_a) = 1$ *iff* $\forall k((1 \leq k \leq n \wedge \tilde{t}_k \in \tilde{RD}(a)) \Rightarrow \exists h(k < h \leq n \wedge \tilde{t}_h \in \tilde{IN}(a)))$

The lemma below relates the firing sequences of $Net(P)$ with the ones in $DNet(P)$: for each firing sequence in $Net(P)$ there exists a firing sequence in $DNet(P)$, obtained by replacing transitions with the corresponding ones, such that the reached markings coincide on common (i.e. not auxiliary) places.

**Lemma 4.12** *Let* $P$ *be an agent,* $Net(P) = (S, T, m_0)$ *and* $DNet(P) = (\tilde{S}, \tilde{T}, \tilde{m_0})$.
*If* $m_0 \xrightarrow{t_1} m_1 \ldots \xrightarrow{t_n} m_n$ *then* $\tilde{m_0} \xrightarrow{\tilde{t_1}} \tilde{m_1} \ldots \xrightarrow{\tilde{t_n}} \tilde{m_n}$, *where*

*(i)* $\tilde{m}_n(s) = m_n(s)$ *for all* $s \in S$

*(ii)* $\tilde{t}_i = \begin{cases} F_{pr}(t_i) & \text{if } \tilde{m}_{i-1}(pr_a) = 1 \\ F_{ab}(t_i) & \text{otherwise} \end{cases}$

The following theorem states that to a dead marking in $Net(P)$ corresponds a dead marking in $DNet(P)$, reached by firing the corresponding transitions. To show that the reached marking in $DNet(P)$ is dead, the only difficult case is if an instance of `Inp-` or `Rdp-` can fire, solved by showing that places $inp(a)?P\_Q$ and $rdp(a)?P\_Q$ are always empty in dead markings of $Net(P)$.

**Theorem 4.13** *Let* $P$ *be an agent,* $Net(P) = (S, T, m_0)$ *and* $DNet(P) = (\tilde{S}, \tilde{T}, \tilde{m_0})$. *If* $Net(p)$ *has a deadlock then* $DNet(P)$ *has a deadlock.*

We need the following auxiliary functions on sequences of transitions: $rmlast^a_k$ is used for transition sequences in $Net(P)$ and removes the last $k$ occurrences of transition `rend(a)`, whereas $rmrend$ is used for transition sequences in $DNet(P)$ and removes all transition `Rend`.

**Definition 4.14**

$$rmlast^a_{i+1}(\sigma t) = \begin{cases} rmlast^a_i(\sigma) & \textit{if } t = \texttt{rend(a)} \\ rmlast^a_{i+1}(\sigma)t & \textit{otherwise} \end{cases}$$

$rmlast^a_{i+1}(\varepsilon) = \varepsilon$
$rmlast^a_0(\sigma) = \sigma$

$$rmrend(\tilde{\sigma}\tilde{t}) = \begin{cases} rmrend(\tilde{\sigma}) & \textit{if } \exists a(\tilde{t} = \texttt{Rend(a)}) \\ rmrend(\tilde{\sigma})t & \textit{otherwise} \end{cases}$$

$rmrend(\varepsilon) = \varepsilon.$

The following lemma enunciates an important property of firing sequences in $Net(P)$: if

(i) the reached marking contains at least $k$ tokens in place $\langle a \rangle$,

(ii) all the tokens in $\langle a \rangle$ at the initial marking have been consumed by in-transitions (i.e. the number of occurrences of in-transitions is greater or equal to the number of tokens in $\langle a \rangle$ at the initial marking) and

(iii) each rd-transition is followed by an in-transition

then the firing sequence obtained by dropping the last $k$ occurrences of `rend(a)` is still firable at the initial marking.

**Lemma 4.15** *Let $P$ be an agent and $Net(P) = (S, T, m_0)$. If*

*(i)* $m_0 \xrightarrow{t_1} \ldots \xrightarrow{t_n} m \oplus k \cdot \langle a \rangle$,

*(ii)* $|\{i \mid 1 \leq i \leq n \wedge t_i \in IN(a)\}| \geq m_0(\langle a \rangle)$ *and*

*(iii)* $\forall i(1 \leq i \leq n \wedge t_i \in RD(a) \Rightarrow \exists j(i < j \leq n \wedge t_j \in IN(a)))$

*then* $m_0 \xrightarrow{\sigma} m \oplus k \cdot \langle\langle a \rangle\rangle$, *where* $\sigma = rmlast^a_k(t_1 \ldots t_n)$.

The following lemma relates firing sequences of $DNet(P)$ with firing sequences of $Net(P)$; though the correspondence on markings is preserved, we do not have the same close correspondence on transitions as for Theorem 4.12: it may happen that the transition `Inp-(a,P,Q)` fires in $DNet(P)$ even if place $\langle a \rangle$ is not empty, thus forbidding the firing of the corresponding transition `inp-(a,P,Q)` in $Net(P)$. However, the enabledness of `Inp-(a,P,Q)` ensures that the hypotheses of Lemma 4.15 are satisfied; let $k$ be the number of tokens in $\langle a \rangle$; we can obtain a firing sequence in $Net(P)$ by dropping the last $k$ `rend(a)` transitions, then fire `inp-(a,P,Q)` (that has become enabled) and finally fire the dropped $k$ `rend(a)` transitions.

**Lemma 4.16** *Let $P$ be an agent, $Net(P) = (S, T, m_0)$ and $DNet(P) = (\tilde{S}, \tilde{T}, \tilde{m_0})$.*
*If* $\tilde{m_0} \xrightarrow{\tilde{\sigma}} \tilde{m}$ *then there exists a permutation $\tilde{\tau}$ of $\tilde{\sigma}$ such that*

*(i)* $rmrend(\tilde{\sigma}) = rmrend(\tilde{\tau})$,

*(ii)* $m_0 \xrightarrow{G(\tilde{\tau})} m$ *and*

*(iii)* $m(s) = \tilde{m}(s)$ *for all* $s \in S$.

Finally, we have that to a dead marking of $DNet(P)$ corresponds a dead marking of $Net(P)$.

**Theorem 4.17** *Let $P$ be an agent, $Net(P) = (S, T, m_0)$ and $DNet(P) = (\tilde{S}, \tilde{T}, \tilde{m_0})$. If $DNet(P)$ has a deadlock then also $Net(P)$ has a deadlock.*

The above theorem allows us to conclude that the termination problem is decidable in $\mathbf{L}_u$, as we have shown that also the presence of deadlock in finite P/T systems is decidable.

**Corollary 4.18** *$\mathbf{L}_u$ is not Turing equivalent.*

## 5 Conclusion

This paper represents the continuation of our paper [5], where we studied the operational and observational semantics of three different interpretations for the output operation for the subcalculus not containing $rd$ and $rdp$. Besides the ordered and unordered semantics presented here, we also studied the so-called *instantaneous*, according to which $out(a).P$ is structural equivalent to $\langle a \rangle | P$ (adopted in the asynchronous $\pi$-calculus [14,1]). In [5] we also claimed the non Turing–equivalence of the unordered semantics; this result is strengthened (it holds also in presence of $rd$ and $rdp$) and proved in full detail here.

The expressiveness of the ordered and unordered semantics for the output operation in Linda has been also studied in order to provide formal support to the reasoning on the implementations of Linda-like languages. The relevant conclusion of this paper is the following: it is not possible to use an implementation of Linda agreeing with the unordered semantics to implement another version of Linda agreeing with the ordered semantics.

Moreover, we claim that the non Turing equivalence result for $\mathbf{L}_u$ holds also if the language is extended with restriction, choice, and full recursion operators, provided that restriction and choice do not appear inside recursion.

## References

[1] G. Boudol. Asynchrony and the $\pi$-calculus. Technical Report 1702, INRIA, Sophia–Antipolis,1992.

[2] A. Brogi and P. Ciancarini. The concurrent language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.

[3] N. Busi, R. Gorrieri, A Petri Net Semantics for $\pi$–calculus, In *Concur'95*, volume 962 of *LNCS*, 145–159, 1995.

[4] N. Busi, R. Gorrieri, and G. Zavattaro. A Truly Concurrent view of Linda Interprocess Communication. Technical Report UBLCS 97–2, Department of Computer Science, University of Bologna, February 1997.

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

[6] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science* (to appear).

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

[8] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 66–76. Springer Verlag, 1995.

[9] A. Cheng, J. Esparza, J. Palsberg. Complexity results for 1-safe nets. In *Theoretical Computer Science*, 147:117–136, 1995.

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

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

[12] U. Goltz. CCS and Petri Nets. LNCS 469, 334–357, Springer–Verlag, 1990.

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

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

[15] R. Milner. The Polyadic $\pi$–Calculus: A Tutorial Technical Report, Department of Computer Science, University of Edinburgh, ECS–LFCS–91–180, October 1991.

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

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

[18] J. E. Narem. An Informal Operational Semantics of C-Linda V2.3.5. Technical Report, Department of Computer Science, Yale University, YALEU/DCS/TR-839, December 1989.

[19] C. Reutenauer. *The Mathematics of Petri Nets.* Masson, 1988.

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

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