# Event Notification in Data-driven Coordination Languages: Comparing the Ordered and Unordered Interpretations*

Nadia Busi and Gianluigi Zavattaro

*Dipartimento di Scienze dell'Informazione, Università di Bologna*

*Mura Anteo Zamboni 7, I-40127 Bologna, Italy*

*e-mail:* busi@cs.unibo.it, zavattar@cs.unibo.it

**Keywords:** Coordination models and languages, shared data space, event notification, expressiveness of concurrent languages, Java coordination middlewares.

## Abstract

JavaSpaces and TSpaces are two coordination middlewares for distributed Java programming recently proposed by Sun and IBM, respectively. They extend the data-driven coordination model of Linda with an event notification mechanism: a process can register interest in the incoming arrivals of a particular kind of data and receive communication of the occurrences of these events. In [2, 3] we introduce a process calculus $\mathbf{L}$, based on the Linda coordination model, and we prove that if processes are synchronous with the dataspace (ordered interpretation) $\mathbf{L}$ is Turing powerful, while this is not the case if they are asynchronous (unordered interpretation). Here, we introduce a new calculus $\mathbf{L}_n$, obtained by extending $\mathbf{L}$ with the event notification mechanism, and we prove two main results contrasting with what has been shown for $\mathbf{L}$: (i) $\mathbf{L}_n$ is Turing powerful also under the unordered interpretation and (ii) it allows a faithful encoding of the ordered semantics on top of the unordered one.

## 1 Introduction

In the last decades we assisted to a dramatic evolution of computing systems, leading from stand-alone mainframes to a worldwide network connecting smaller, yet much more powerful processors. The next expected step in this direction is represented by the so-called *ubiquitous computing*, based on the idea of dynamically reconfigurable federations composed of users and resources required by those users. For instance, the Jini architecture [15] represents a first proposal of Sun for a Java-based technology inspired by this new computing paradigm.

In this scenario, one of the most challenging topics is concerned with the coordination of the federated components. For this reason, a renewed interest in coordination languages – that have been around for more than fifteen years – has arisen. For example, JavaSpaces [14] and TSpaces [16] are two recent coordination middlewares for distributed Java programming proposed by Sun and IBM, respectively. These proposals extend the *data-driven* coordination model of Linda [6] (processes interact via the introduction ($out$), the consumption ($in$), and the test for absence ($inp$) of data inside a shared dataspace) with a new event notification mechanism. This mechanism allows a process to register interest in the future arrivals of a particular kind of data, by defining a process which is triggered as reaction to the occurrence of these events.

In [2, 3] we investigate the expressive power of the Linda coordination model under two different interpretations of the $out$ operator. This operator can be seen as composed of two phases: the *emission* phase in which a process executes its $out$ operation and sends the datum to the dataspace, and the *rendering* phase during which the new datum reaches the dataspace and becomes effectively available. The following two different semantics can be suggested:

- *Ordered*: The emission and rendering of one datum form together one single autonomous atomic action. In this way, the order of emission is respected by the rendering order. Under this semantics, the processes are synchronous with respect to the dataspace when they emit messages, hence a hand–shake protocol could be used in order to implement $out$: the sender sends the new datum and then waits for an acknowledgement.
- *Unordered*: The emission and the rendering of one datum are distinct autonomous actions. In this case the processes behave asynchronously with respect to the dataspace: the implementation of $out$ could consist of a send operation of message $a$ to the dataspace, 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 $b$.

In [2, 3] we define a process algebra $\mathbf{L}$ based on the Linda coordination primitives. That calculus is equipped with both the ordered and the unordered semantics, and it is proved that the expressive power dramatically changes under the two interpretations: under the ordered semantics $\mathbf{L}$ is Turing powerful, while this is not the case under the unordered semantics.

*Work partially supported by Esprit working group n.24512 "Coordina"

Turing completeness is proved by showing that it is possible to encode any Random Access Machine (RAM) [13]. On the other hand, the non Turing equivalence result is proved by resorting to a Place/Transition net semantics, a formalism in which the existence of a terminating computation is decidable.

In this paper we define a new calculus $\mathbf{L}_n$ by extending $\mathbf{L}$ with an event notification mechanism inspired by the notify primitive of JavaSpaces[14]. More precisely, we introduce a $\mathit{notify}(a, P)$ primitive, where $a$ is a possible kind of datum and $P$ is a process. By executing this primitive, a process registers its interest in the incoming arrivals of new instances of datum $a$. Every time this event occurs, a new instance of the process $P$ is spawn.

We prove two main results that contrast with what has been proved for $\mathbf{L}$:

(i) The calculus $\mathbf{L}_n$ is Turing powerful also under the unordered semantics. This result permits us to conclude that the event notification mechanism strictly increases the expressive power of a Linda-like coordination language that follows the unordered interpretation.

(ii) In $\mathbf{L}_n$ it is possible to faithfully encode the ordered semantics on top of the unordered one. According to this result we can state that the ordered *out* can be simulated also by the unordered one, provided that the underlying coordination model comprises also an event notification mechanism.

The paper is structured as follows. Section 2 introduces the syntax of the calculus and the two semantics corresponding to the ordered and unordered interpretations. Sections 3 and 4 discuss the results (i) and (ii), respectively. Section 5 reports some conclusive remarks.

## 2 The Syntax and the Operational Semantics

We first define the syntax of $\mathbf{L}_n$ and then we present the two operational semantics following the ordered and the unordered interpretations, respectively.

Let *Name* be a denumerable set of names of data, ranged over by $a, b, \ldots$. The syntax is defined by the following grammar:

$$\begin{array}{lcl} P & ::= & \langle a \rangle \mid C \mid P|P \\ C & ::= & 0 \mid \mu.C \mid \mathit{inp}(a)?C\_C \mid C|C \end{array}$$

where:

$$\mu \;::=\; \mathit{in}(a) \mid \mathit{out}(a) \mid \mathit{notify}(a, C) \mid !\mathit{in}(a)$$

Agents, ranged over by $P$, $Q$, ..., consist of the parallel composition of the data already in the dataspace (each one denoted by one agent $\langle a \rangle$) and the concurrent programs that share these data. A program can be a terminated program $0$ (which is usually omitted for the sake of simplicity), a prefix form $\mu.P$, an *if-then-else* form $\mathit{inp}(a)?P\_Q$, or the parallel composition of programs. Let $\mathbf{L}_n$ denote the calculus comprising all the possible agents and $\mathbf{L}$ be the subcalculus without the $\mathit{notify}(a, P)$ prefix.

A prefix $\mu$ can be one of the primitives $\mathit{in}(a)$ or $\mathit{out}(a)$, indicating the withdrawing or the emission of datum $a$ respectively, and the $\mathit{notify}(a, P)$ operation that registers interest in the incoming arrivals of new instances of datum $a$: every time a new instance of $\langle a \rangle$ is produced, a new copy of program $P$ is spawn. We also consider the bang operator $!\mathit{in}(a)$ which is a form of replication guarded on input operations: the term $!\mathit{in}(a).P$ is always ready to consume an instance of $\langle a \rangle$ and then activate a copy of $P$. The if-then-else form is used to model the *inp* primitive: $\mathit{inp}(a)?P\_Q$ is a program which requires an instance of $\langle a \rangle$ to be consumed; if it is present, the program $P$ is executed, otherwise $Q$ is chosen. In the following, *Agent* denotes the set containing all possible agents. We use $\prod$ to denote the parallel composition of a set of indexed agents, e.g. $\prod_i P_i = P_1|P_2|\ldots|P_n$.

The two semantics are defined as labeled transition systems of the kind $(\mathit{Agent}, \mathit{Label}, \longrightarrow)$ where $\mathit{Label} = \{\tau\} \cup \{a, \overline{a}, \neg a, \vec{a}, \dot{a} \mid a \in \mathit{Name}\}$ (ranged over by $\alpha$, $\beta$, ...) is the set of possible labels. Note that, despite of their graphical similarity, $\vec{a}$ and $\bar{a}$ are two different labels (with a different meaning, as explained below) . The labeled transition relation $\longrightarrow_o$ for the *ordered* semantics is the smallest one satisfying the axioms and rules $(1)-(14)$ (plus their symmetric versions) in Table 1 and the axiom (15) of Table 2; while $\longrightarrow_u$ for the unordered semantics considers the axioms and rules $(1)-(14)$ (plus their symmetric versions), (16) and (17). For the sake of simplicity, the symmetric versions of rules $(8)-(14)$ are not explicitly reported in Table 1. The indexes $o$ and $u$ (distinguishing the two transition relations) are omitted when no confusion arises.

Axiom (1) indicates that $\langle a \rangle$ is able to give its contents to the environment by performing an action labeled with $\overline{a}$. Axiom (2) associates to the action performed by the prefix $\mathit{in}(a)$ the label $a$, which is the complement of $\overline{a}$.

Axiom (3) indicates that the $\mathit{notify}(a, P)$ prefix produces a new kind of agent $\mathit{on}(a).P$ (that we add to the syntax as an auxiliary term). This process spawns a new instance of $P$ every time a new $\langle a \rangle$ is produced. This behaviour is described in axiom (4) where the label $\dot{a}$ is used to describe this kind of computation step.

Axioms (5) and (6) describe the semantics of $\mathit{inp}(a)?P\_Q$: if the required $\langle a \rangle$ is present it can be consumed (axiom (5)), otherwise, in the case $\langle a \rangle$ is not available, its absence is guessed by performing an action labeled with $\neg a$ (axiom (6)).

The term $!\mathit{in}(a).P$ permits to model infinite computations; it is always able to spawn an instance of program $P$ upon consumption of an occurrence of $\langle a \rangle$ (axiom (7)).

Rules (8) and (9) are the usual local and synchronization rules, respectively. Observe that the standard local rule can be applied only to actions different from the non-standard $\neg a$, $\vec{a}$, and $\dot{a}$.

Rules $(10)-(14)$ regard the way actions labeled with the non-standard labels $\neg a$, $\dot{a}$, and $\vec{a}$ are inferred to structured terms. Rule (10) indicates that actions labeled with $\neg a$ can be performed only if no $\langle a \rangle$ is present in the environment (i.e. no transition labeled with $\overline{a}$ can be performed). Rules (11) and (12) consider actions labelled with $\dot{a}$ indicating the interest in the incoming instances of $\langle a \rangle$. If one process able to perform this kind of action is composed in parallel with another one registered for the same event then their local actions are combined in a global one (rule (11)); otherwise, the process performs its own action leaving the environment unchanged (rule (12)). Rules (13) and (14) deal with two different cases regarding the label $\vec{a}$ which indicates the arrival of a new instance of $\langle a \rangle$: if processes waiting for the notification of this event are present in the environment they are woken-up (rule (13)); otherwise, the environment is left unchanged (rule (14)).

The rules which differentiate the *ordered* from the *unordered* semantics are presented in Table 2. In the *ordered* approach the output operation $\mathit{out}(a)$ consists of the cre-

$$
\begin{array}{llll}
(1) & \langle a \rangle \xrightarrow{\overline{a}} 0 & (2) & in(a).P \xrightarrow{a} P \\
(3) & notify(a, Q).P \xrightarrow{\tau} on(a).Q|P & (4) & on(a).P \xrightarrow{\dot{a}} P|on(a).P \\
(5) & inp(a)?P\_Q \xrightarrow{a} P & (6) & inp(a)?P\_Q \xrightarrow{\neg a} Q \\
(7) & !in(a).P \xrightarrow{a} P|!in(a).P & (8) & \dfrac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad \alpha \neq \neg a, \vec{a}, \dot{a} \\
(9) & \dfrac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P|Q \xrightarrow{\tau} P'|Q'} & (10) & \dfrac{P \xrightarrow{\neg a} P' \quad Q \not\xrightarrow{\overline{a}}}{P|Q \xrightarrow{\neg a} P'|Q} \\
(11) & \dfrac{P \xrightarrow{\dot{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P|Q \xrightarrow{\dot{a}} P'|Q'} & (12) & \dfrac{P \xrightarrow{\dot{a}} P' \quad Q \not\xrightarrow{\dot{a}}}{P|Q \xrightarrow{\dot{a}} P'|Q} \\
(13) & \dfrac{P \xrightarrow{\vec{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P|Q \xrightarrow{\vec{a}} P'|Q'} & (14) & \dfrac{P \xrightarrow{\vec{a}} P' \quad Q \not\xrightarrow{\dot{a}}}{P|Q \xrightarrow{\vec{a}} P'|Q}
\end{array}
$$

Table 1: Operational semantics (symmetric rules omitted).

$$
\textbf{Ordered:}\ (15)\ out(a).P \xrightarrow{\vec{a}} \langle a \rangle|P \qquad \textbf{Unordered:}\ \begin{array}{ll} (16) & out(a).P \xrightarrow{\tau} \langle\langle a \rangle\rangle|P \\ (17) & \langle\langle a \rangle\rangle \xrightarrow{\vec{a}} \langle a \rangle \end{array}
$$

Table 2: Two semantics for the *out* primitive.

ation of $\langle a \rangle$ in one step. In this way, when a sequence of outputs is executed, the new data are rendered in the same order as they are emitted. The production of this new instance of $\langle a \rangle$ is communicated to the environment by decorating this action with the label $\vec{a}$ (see rules (13) and (14)). In the *unordered* approach, the execution of $out(a)$ does not directly generate the corresponding term $\langle a \rangle$, but it creates an agent which will make $\langle a \rangle$ available only after a non–predictable delay. This agent, denoted by $\langle\langle a \rangle\rangle$ (term added to the syntax of the calculus in the case of unordered semantics) represents the fact that the datum $a$ has been sent to the dataspace but it is not received yet. This behaviour is obtained by allowing $\langle\langle a \rangle\rangle$ to become $\langle a \rangle$. This step is decorated with $\vec{a}$ for the reasons discussed above.

Note that rules (10), (12), and (14) use negative premises; however, our operational semantics is well defined, because our transition system specification is strictly stratifiable [7], a condition that ensures (as proved in [7]) the existence of a unique transition system agreeing with it.

The set of names occurring in an agent $P$, denoted as $n(P)$, is defined as follows:

$$
\begin{array}{lcl}
n(0) & = & \emptyset \\
n(\langle a \rangle) & = & \{a\} \\
n(\langle\langle a \rangle\rangle) & = & \{a\} \\
n(in(a).P) & = & \{a\} \cup n(P) \\
n(out(a).P) & = & \{a\} \cup n(P) \\
n(!in(a).P) & = & \{a\} \cup n(P) \\
n(on(a).P) & = & \{a\} \cup n(P) \\
n(notify(a, P).Q) & = & \{a\} \cup n(P) \cup n(Q) \\
n(inp(a)?P\_Q) & = & \{a\} \cup n(P) \cup n(Q) \\
n(P|Q) & = & n(P) \cup n(Q)
\end{array}
$$

We define a *structural congruence* (denoted by $\equiv$) as the minimal congruence relation satisfying the monoidal laws for the parallel composition operator:

$$
\begin{array}{l}
P \equiv P|0 \\
P|Q \equiv Q|P \\
P|(Q|R) \equiv (P|Q)|R
\end{array}
$$

The following fact states that two structural congruent agents are observationally indistinguishable.

**Fact 2.1** Let $P$ and $Q$ be two agents such that $P \equiv Q$. If $P \xrightarrow{\alpha} P'$ then there exists $Q'$ such that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q'$. ■

This fact permits us to reason up-to structural congruence in the remainder of the paper, i.e., we consider as the same term all agents $P$ and $Q$ such that $P \equiv Q$.

In the following we only consider computations consisting of reduction steps, i.e., the internal transitions that a *stand-alone* agent is able to perform independently of the context. In our calculus, we consider as reductions not only the usual transitions labeled with $\tau$, but also the non-standard labeled with $\neg a$ and $\vec{a}$. In fact, the transition $P \xrightarrow{\neg a} P'$ indicates that $P$ can become $P'$ if no $\langle a \rangle$ is available in the external environment, and $P \xrightarrow{\vec{a}} P'$ describes that a new agent $\langle a \rangle$ has been produced. Hence, in any of these cases, if $P$ is stand-alone (i.e. without external environment) it is able to become $P'$. Indeed, these labels have been used only for helping a SOS [12] formulation of the semantics, but they correspond conceptually to internal steps. Formally, we define reduction steps as follows:

$$
P \longrightarrow P' \quad \text{iff} \left\{ \begin{array}{l} P \xrightarrow{\tau} P' \text{ or} \\ P \xrightarrow{\neg a} P' \text{ or P } \xrightarrow{\vec{a}} \text{ P' for some } a \end{array} \right.
$$

We use $P \not\longrightarrow$ to state that there exists no $P'$ such that $P \longrightarrow P'$.

An agent $P$ has a terminating computation (denoted by $P\downarrow$) if it can block after a finite amount of internal steps: $P \longrightarrow^* P'$ with $P' \not\longrightarrow$. On the other hand, an agent $P$ has an infinite computation (denoted by $P\uparrow$) if there exists an infinite computation starting from $P$: for each natural index $i$ there exists $P_i$ such that $P = P_0$ and $P_i \longrightarrow P_{i+1}$. Observe that due to the nondeterminism of our calculi the two above conditions are not in general mutually exclusive, i.e., given a process $P$ both $P\downarrow$ and $P\uparrow$ may hold.

## 3 Encoding Random Access Machines

In order to prove that $\mathbf{L}_n$ is Turing powerful also under the unordered interpretation we present how to encode in the calculus any Random Access Machine (RAM) [13], a well known Turing equivalent formalism.

A RAM is a computational model composed of a finite set of registers, that can hold arbitrary large natural numbers, and a program, that is a sequence of simple numbered instructions, like arithmetical operations (on the content 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 content of the registers specified as outputs.

In [9] it is shown that the following two instructions are sufficient to model every recursive function:

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

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

$$\begin{array}{lcl} 1 & : & DecJump(r_2, 4) \\ 2 & : & Succ(r_1) \\ 3 & : & DecJump(r_3, 1) \end{array}$$

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 content 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:

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

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 $i'^{th}$ instruction, and the registers will contain $c'_1, \ldots, c'_n$.

Before presenting how to encode any RAM in $\mathbf{L}_n$ under the unordered interpretation, we report here an encoding of RAM in $\mathbf{L}$ that is proved valid under the ordered semantics in [2, 3].

### 3.1 The ordered case

We now recall the modeling of a RAM in $\mathbf{L}$ that we proposed in [2, 3]. It consists of one 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* $\langle p_i \rangle$ while the contents of register $r_l$ is modeled by $c_l$ occurrences of $\langle r_l \rangle$ (for $l = 1 \ldots n$):

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

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, adds an agent $r_j$ and updates the program counter by producing $p_{i+1}$. An instruction $DecJump(r_j, s)$ at position $i$ is modeled by an agent that, after consuming the program counter, performs an *inp* on message $r_j$; if the operation succeeds, then an instance of $\langle r_j \rangle$ has been withdrawn from the dataspace and the agent updates the program counter by adding $\langle p_{i+1} \rangle$, otherwise a jump to the *sth* instruction is performed by adding $\langle p_s \rangle$.

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

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.

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

The following theorem formally states the correctness of the encoding.

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

- *Soundness:* if
$$[\![R]\!] | [\![(i, c_1, c_2, \ldots, c_n)]\!] \longrightarrow^3_o Q$$
then there exists a configuration $(i', c'_1, c'_2, \ldots, c'_n)$, and it is unique, such that
$$\begin{array}{l} Q = [\![R]\!] | [\![(i', c'_1, c'_2, \ldots, c'_n)]\!] \text{ and} \\ (i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c'_1, c'_2, \ldots, c'_n) \end{array}$$
- *Completeness:* if
$$(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c'_1, c'_2, \ldots, c'_n)$$
then also
$$[\![R]\!] | [\![(i, c_1, c_2, \ldots, c_n)]\!] \longrightarrow^3_o [\![R]\!] | [\![(i', c'_1, c'_2, \ldots, c'_n)]\!]$$

■

As termination and divergence are not decidable in Turing powerful formalisms, we can conclude that both $P\downarrow$ and $P\uparrow$ are undecidable properties in $\mathbf{L}$ under the ordered semantics.

### 3.2 The unordered case

Under the unordered interpretation the calculus $\mathbf{L}$ is no more Turing powerful. The proof of this result can be found in [2, 3] where, by resorting to a Place/Transition net semantics (a formalism in which the existence of a terminating computation is decidable), we prove that both $P\downarrow$ and $P\uparrow$ are decidable in $\mathbf{L}$ under the unordered interpretation.

Here we recall in detail why the above implementation of the RAM fails in the unordered case. The problem can be localized in the way the $i : Succ(r_j)$ instruction is implemented: after consumption of the program counter tuple $\langle p_i\rangle$, it emits first $\langle r_j\rangle$ and then the new program counter $\langle p_{i+1}\rangle$. The order of production of these two items is relevant, as the following example shows.

Consider the execution of the program:

$$\begin{array}{lcl} 1 & : & Succ(r_1) \\ 2 & : & DecJump(r_1, 1) \end{array}$$

starting with the register $r_1$ empty. The implementation of the $Succ(r_1)$ instruction under the unordered interpretation creates two agents: $\langle\langle r_1\rangle\rangle$ and $\langle\langle p_2\rangle\rangle$. If the program counter $\langle p_2\rangle$ becomes available before $\langle r_1\rangle$, the following $DecJump(r_1, 1)$ instruction could execute the jump because no $\langle r_1\rangle$ is available. Then, the computation gives rise to a loop that is exited only when the first $\langle r_1\rangle$ becomes available in the dataspace. 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.

Now we wonder if the *notify* primitive could be useful to solve this kind of problem in the implementation of the instruction $i : Succ(r_j)$. The intuition is that *notify* permits to know that a new instance of $\langle r_j\rangle$ has been produced, i.e., that the register $r_j$ has been really incremented, before producing the new program counter. More precisely, the idea is to move the $out(p_{i+1})$ operation inside a *notify* primitive that registers interest in the next production of $\langle r_j\rangle$. This ensures that the program counter increment is performed only after the register has been incremented.

Unfortunately, the *notify* primitive registers interest in all the forthcoming productions of $\langle r_j\rangle$, and not only in the first one. To overcome this problem, we use a one-shot enabling mechanism consisting of the emission of a datum $\langle en_i\rangle$, that must be consumed before the program counter updating is performed.

Formally we redefine in the case of unordered interpretation:

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

In this case the assertion of the theorem stating the correctness of the encoding is more involved. This is because we have to take into account also the redundant (and harmless) processes derived from the *notify* primitive, a copy of which is generated every time an increment on the corresponding register is performed. These agents are represented by the terms

$$\prod_k on(r_{j_k}).in(en_{i_k}).out(p_{i_k+1})$$

representing the processes that wait for the production of new data $\langle r_{i_k}\rangle$, and the agents

$$\prod_l in(en_{h_l}).out(p_{h_l+1})$$

which are spawn when new $\langle r_{h_s}\rangle$ have been produced. Another difference with the previous encoding is that now a *DecJump* operation executes 4 reduction steps while a *Succ* requires 9 steps.

**Theorem 3.2** Let $R$ be a RAM program, then:

- *Soundness:* if

$$\begin{array}{l} [\![R]\!]|[\![(i, c_1, c_2, \ldots, c_n)]\!]| \\ \prod_k on(r_{j_k}).in(en_{i_k}).out(p_{i_k+1})| \\ \prod_l in(en_{h_l}).out(p_{h_l+1}) \\ \longrightarrow_u Q \end{array}$$

  then there exists an agent $Q'$ and a unique configuration $(i', c'_1, c'_2, \ldots, c'_n)$ such that

$$\begin{array}{l} Q \longrightarrow_u^3 Q' \text{ or } Q \longrightarrow_u^8 Q' \text{ and} \\ Q' = [\![R]\!]|[\![(i', c'_1, c'_2, \ldots, c'_n)]\!]| \\ \quad \prod_m on(r_{i'_m}).in(en_{j'_m}).out(p_{j'_m+1})| \\ \quad \prod_s in(en_{h'_s}).out(p_{h'_s+1}) \text{ and} \\ (i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c'_1, c'_2, \ldots, c'_n) \end{array}$$

- *Completeness:* if

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

  then also

$$\begin{array}{l} [\![R]\!]|[\![(i, c_1, c_2, \ldots, c_n)]\!]| \\ \prod_k on(r_{i_k}).in(en_{j_k}).out(p_{j_k+1})| \\ \prod_l in(en_{h_l}).out(p_{h_l+1}) \\ \longrightarrow_u^h \\ [\![R]\!]|[\![(i', c'_1, c'_2, \ldots, c'_n)]\!]| \\ \prod_m on(r_{i'_m}).in(en_{j'_m}).out(p_{j'_m+1})| \\ \prod_s in(en_{h'_s}).out(p_{h'_s+1}) \end{array}$$

  with $h = 4$ or $h = 9$. ■

## 4 Encoding the Ordered in the Unordered Semantics

In the previous section it is shown that $\mathbf{L}_n$ is Turing equivalent under both the ordered and the unordered interpretations. In this section we go a step further: by using an improvement of the technique adopted in the previous section to sequentialize the emission of data corresponding to the register increment and the program counter update, we provide an encoding of the ordered version of $\mathbf{L}_n$ in the unordered one.

A first attempt consists of modeling the ordered $out(a)$ in an unordered setting by putting the continuation of the $out(a)$ operation inside a $notify(a, P)$ primitive:

$$[\![out(a).P]\!] = notify(a, in(en_{a\cdot P}).[\![P]\!]).out(en_{a\cdot P}).out(a)$$

$$[\![P]\!] = [P]|ME_{n(P)}$$

$$[0] = 0 \qquad [notify(a, P).Q] = notify(a, [P]).[Q]$$

$$[\langle a\rangle] = \langle a\rangle \qquad [inp(a)?P\_Q] = inp(a)?[P]\_[Q]$$

$$[in(a).P] = in(a).[P] \qquad [on(a).P] = on(a).[P]$$

$$[!in(a).P] = !in(a).[P] \qquad [P|Q] = [P]|[Q]$$

$$[out(a).P] = in(me_a).notify(a, O(a, P)).out(en_{a\cdot P}).out(a)$$

$$ME_A = \prod_{a\in A}\langle me_a\rangle$$

$$O(a, P) = in(en_{a\cdot P}).out(me_a).[P]$$

Table 3: Encoding the ordered interpretation in the unordered one.

As we already explained in the previous section, we use $\langle en_{a\cdot P}\rangle$ as a one-shot enabler to prevent the spawn of more than one copy of output continuations. While this technique works well for agents with a sequential behaviour (as the ones arising from the RAM encoding), it provides an incorrect encoding if two or more outputs of the same datum are performed in parallel. As an example, consider the agent

$$\begin{array}{l} out(a).out(b) \mid \\ out(a).out(c) \mid \\ in(b).in(c).in(a).inp(a)?Div\_0 \end{array}$$

where $Div = out(div).!in(div).out(div)$ is a nonterminating term. In the ordered case we are sure that, when data $\langle b\rangle$ and $\langle c\rangle$ have reached the dataspace, also two copies of $\langle a\rangle$ are in the dataspace. Hence, this process cannot terminate. On the other hand, consider the encoding of the above agent under the unordered interpretation and suppose that both the encodings of $out(a)$ start their execution, reaching the term

$$\begin{array}{l} on(a).in(en_{a\cdot out(b)}).[out(b)] \mid \langle en_{a\cdot out(b)}\rangle \mid \langle\langle a\rangle\rangle \mid \\ on(a).in(en_{a\cdot out(c)}).[out(c)] \mid \langle en_{a\cdot out(c)}\rangle \mid \langle\langle a\rangle\rangle \mid \\ in(b).in(c).in(a).inp(a)?[Div]\_0 \end{array}$$

At this point, when the first copy of $\langle a\rangle$ reaches the dataspace, both the waiting processes are woken. Thus, data $\langle b\rangle$ and $\langle c\rangle$ can be produced even if a single copy of $\langle a\rangle$ appears in the dataspace. In this situation, the third process can perform its execution and terminate (the presence of only one instance of $\langle a\rangle$ ensures that the branch with the nonterminating program $Div$ is not chosen); hence, the whole agent may terminate.

To solve this problem, we avoid the existence of more than one process concurrently executing the output protocol on the same datum $a$ by means of a mutual exclusion object $me_a$, that must be acquired when the protocol starts and is released when it ends.

The formal definition of the encoding is presented in Table 3. It consists of a two level encoding: the outer level $[\![\ ]\!]$ introduces the mutual exclusion objects $me_a$, while the inner level $[\ ]$ transforms programs containing ordered outputs in equivalent programs interpreted under the unordered semantics.

The proof of the correctness of the encoding relies on two theorems enunciated below: the first one regards the *completeness* of the encoding (each step performed by a process under the ordered interpretation can be mimicked by a sequence of moves of its encoding under the unordered one) while the second considers *soundness* (each behaviour of a target process of the encoding function corresponds to a computation of the corresponding source).

Also in these theorems we have to deal with redundant terms which are created during the computation as effect of the use of the *notify* operation in a one-shot way. In these cases the terms are denoted by $\prod_i on(a_i).O(a_i, P_i)$ and $\prod_j O(b_j, Q_j)$.

**Theorem 4.1** - *Completeness* - Let $P$ be a term of $\mathbf{L}_n$ and $n(P) \subseteq A$. If $P \longrightarrow_o P'$ then

$$[\![P]\!]|ME_A|\prod_i on(a_i).O(a_i, P_i)|\prod_j O(b_j, Q_j) \longrightarrow_u^+ [\![P']\!]|ME_A|\prod_h on(a'_h).O(a'_h, P'_h)|\prod_k O(b'_k, Q'_k) \quad \blacksquare$$

The theorem regarding soundness has a more complex assertion. This is because given a program $P$ with ordered outputs and its encoding $[\![P]\!]$, if we take a computation $[\![P]\!] \longrightarrow^* P''$ the agent $P''$ in general is not the encoding of an agent $P'$ reachable from $P$. Nevertheless, we prove that it is always possible to extend the considered computation $[\![P]\!] \longrightarrow^* P''$ in order to reach an agent $P'''$ that satisfies the requirements, i.e., $P'' \longrightarrow^* P'''$ where $P''' = [\![P']\!]|ME_{n(P)}$ (plus some redundant terms) and $P \longrightarrow^* P'$. This is a direct corollary of the following more general theorem.

**Theorem 4.2** - *Soundness* - Let $P$ be a term of $\mathbf{L}_n$ and $n(P) \subseteq A$.
If $[\![P]\!]|ME_A|\prod_i on(a_i).O(a_i, P_i)|\prod_j O(b_j, Q_j) \longrightarrow_u^+ Q$ then there exists $P'$ such that $P \longrightarrow_o^+ P'$ and $Q \longrightarrow_u^+ [\![P']\!]|ME_A|\prod_h on(a'_h).O(a'_h, P'_h)|\prod_k O(b'_k, Q'_k)$. $\blacksquare$

## 5 Conclusion

In this paper we investigate the comparison between the ordered and the unordered interpretations for Linda-like coordination languages extended with an event notification mechanism inspired by JavaSpaces [14] and TSpaces [16].

In [2, 3] it is proved that a process calculus based on the Linda coordination paradigm is Turing powerful under the ordered interpretation and not under the unordered one. Here, we extend that calculus with an event notification mechanism that permits to register interest in the forthcoming arrivals of a particular kind of data. We prove that the extended calculus is Turing powerful also under the unordered interpretation and that it permits to encode the ordered interpretation on top of the unordered one. For the sake of simplicity we have not considered the non-consuming versions of the input primitives (e.g., the *rd* and *rdp* of Linda), but it is not difficult to see that the results proved here hold also in presence of these operations.

The results proved in this paper have not only a theoretical relevance, but we think that they are of interest also for designers and implementors of coordination languages. For instance, we present how to implement the ordered output on top of the unordered one. The existence of this encoding seems to justify, in presence of event notification, the choice of the weaker and easier to implement unordered semantics instead of the more expressive, but more expensive, ordered one.

Another interesting issue addressed by this paper is the study of the interplay between the classical Linda-like coordination paradigm and the event notification mechanism. Historically, these two coordination mechanisms have inspired two distinct and separated families of coordination languages [11]. On one side Linda has inspired a big amount of data-oriented languages while the event notification is the basis of the so-called control-oriented languages (e.g., Manifold [1]). To the best of our knowledge, the present paper represents the first attempt to give a formal treatment to a coordination language which embodies both the coordination mechanisms.

Moreover, we think it is worthwhile to study the expressiveness of the fragment of our calculus in which no test for absence (i.e. *inp*) is considered. The interest in this investigation is justified by the fact that the *inp* operator is an example of a coordination primitive which is powerful but also difficult to implement, in particular in distributed memory systems (see e.g. [8]). The ordered case is analysed in [4], where we show that the calculus with only *in*, *out*, and *notify* permits a nondeterministic implementation of RAM. More precisely, the implementation has a terminating computation if and only if the RAM terminates, but the implementation can exhibit additional nonterminating behaviours. On the other hand, it is easy to see that the calculus with only *in* and *out* is not Turing equivalent, neither in a nondeterministic way. As the addition of *notify* strictly increases the expressiveness of the calculus with only *in* and *out*, we think that it is worthwhile to investigate the *notify* primitive as a good compromise between expressiveness and implementability.

As future work we plan to investigate more sophisticated forms of event notification. For example, in TSpaces [16] a process can register interest not only in the arrival of a message, but also in the execution of other operations on the dataspace. A further step is taken in [10, 5] where it is also possible to rewrite the semantics associated to a dataspace operation, by specifying a reaction which is executed in place of its usual implementation.

## References


[1] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

[2] N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing Equivalence of Linda Coordination Primitives. In *Proc. of Express'97*, volume 7 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1997.

[3] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, to appear, 2000.

[4] N. Busi and G. Zavattaro. On the Expressiveness of Event Notification in Data-driven Coordination Languages. Technical Report UBLCS-99-22, Comp. Science Laboratory, Università di Bologna, Italy, 1999.

[5] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In *Proc. of MA'98*, volume 1467. Lecture Notes in Computer Science, 1998.

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

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

[8] J. Leichter. *Shared Tuple Memories, Shared Memories, Buses and LANS: Linda Implementations Across the Spectrum of Connectivity*. PhD thesis, Yale University Department of Computer Science, 1989.

[9] M. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, 1967.

[10] A. Omicini and F. Zambonelli. Tuple centres for the coordination of Internet agents. In *Proc. of SAC'99*, pages 183–190. ACM Press, 1999.

[11] G. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46:329–400, 1998.

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

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

[14] Sun Microsystem, Inc. *JavaSpaces Specifications*, 1998. Available at `http://www.sun.com`.

[15] Sun Microsystem, Inc. *Jini Architecture Specifications*, 1998. Available at `http://www.sun.com`.

[16] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998.


## Biographical notes

**Nadia Busi** is a researcher working at the Department of Computer Science at Bologna University. Her interests include concurrency theory, theory and application of Petri nets, semantics and expressiveness of concurrent languages.

**Gianluigi Zavattaro** is a Ph.D. student in Computer Science at Bologna University. His research interests include concurrency theory, semantics and expressiveness of coordination languages, and models for object-oriented distributed programming.