On the Expressiveness of Event Notification in Data-driven Coordination Languages*

Nadia Busi and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
busi,zavattar@cs.unibo.it

Abstract. JavaSpaces and TSpaces are two coordination middlewares for distributed Java programming recently proposed by Sun and IBM, respectively. They are both inspired by the Linda coordination model: processes interact via the emission (*out*), consumption (*in*) and the test for absence (*inp*) of data inside a shared repository. The most interesting improvement introduced by these new products is the *event notification* mechanism (*notify*): a process can register interest in the incoming arrivals of a particular kind of data, and then receive communication of the occurrence of these events. We investigate the expressiveness of this new coordination mechanism and we prove that even if event notification strictly increases the expressiveness of a language with only input and output, the obtained language is still strictly less expressive than a language containing also the test for absence.

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 [19] 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 [18] and TSpaces [20] are two recent coordination middlewares for distributed Java programming proposed by Sun and IBM, respectively. These proposals incorporate the main features of both the two historical groups of coordination models [13]: the *data-driven* approach, initiated by Linda [8] and based on the notion of a shared data repository, and the *control-driven* model, advocated by Manifold [1] and centered around the concepts of raising and reaction to events. Besides the typical Linda-like

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

coordination primitives (processes interact via the introduction, consumption and test for presence/absence of data inside a shared repository) both JavaSpaces and TSpaces provide event registration and notification. This mechanism allows a process to register interest in the future arrivals of a particular kind of data, and then receive communication of the occurrence of these events.

In this paper we investigate the interplay of the event notification mechanism with the classical Linda-like coordination paradigm. In particular we focus on the expressive power of event notification and we prove the existence of a hierarchy of expressiveness among the possible combinations of coordination primitives: *in*, *out*, and *inp* are strictly more expressive than *in*, *out*, and *notify*, which in turn are strictly more expressive than *in* and *out* only.

These results are proved by introducing a minimal language containing all the coordination mechanisms we are dealing with, and by considering the sublanguages corresponding to the various combinations of the coordination primitives. The complete language (denoted by $\mathbf{L}_{ntf,inp}$) is obtained by extending a Linda based process algebra presented in [2] with the event notification mechanism. We consider the following sublanguages: $\mathbf{L}$ containing only *in* and *out*, $\mathbf{L}_{ntf}$ containing also *notify*, and $\mathbf{L}_{inp}$ containing *in*, *out* and *inp*.

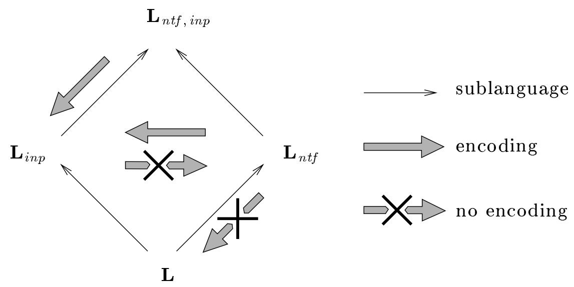

Fig. 1. Overview of the results.

The hierarchy of expressiveness sketched above follows from the three results summarized in Figure 1.

The expressiveness gap between $\mathbf{L}_{ntf}$ and $\mathbf{L}$ can be deduced by the following facts:

(1) There exists an encoding of $\mathbf{L}$ on finite Place/Transition nets [14, 16] which preserves the interleaving semantics. As the existence of a terminating computation is decidable in P/T nets [6], the same holds also in $\mathbf{L}$.
(2) There exists a *nondeterministic* implementation of Random Access Machines (RAM) [17], a well known Turing powerful formalism, in $\mathbf{L}_{ntf}$. The implementation preserves the terminating behaviour: a RAM terminates if and only if the corresponding implementation has a terminating computation. Thus, the existence of a terminating computation is not decidable in $\mathbf{L}_{ntf}$.

Hence there exists no encoding of $\mathbf{L}_{ntf}$ in $\mathbf{L}$ which preserves at least the existence of a terminating computation.

The discrimination between $\mathbf{L}_{inp}$ and $\mathbf{L}_{ntf}$ proceeds in a similar way:

(3) There exists an encoding of $\mathbf{L}_{ntf}$ on finite Place/Transition nets extended with transfer arcs [7] which preserves the existence of an infinite computation. As this property is decidable in this kind of P/T nets, the same holds also in $\mathbf{L}_{ntf}$.

(4) There exists a *deterministic* implementation of RAM in $\mathbf{L}_{inp}$ such that a RAM terminates if and only if all the computation of the corresponding implementation terminate. Thus, the existence of an infinite computation is not decidable in $\mathbf{L}_{inp}$.

Hence there exists no encoding of $\mathbf{L}_{inp}$ in $\mathbf{L}_{ntf}$ which preserves at least the existence of an infinite computation.

Finally, the last result is:

(5) The event notification mechanism can be realized by means of the *inp* operator; indeed we provide an encoding of $\mathbf{L}_{ntf,inp}$ in $\mathbf{L}_{inp}$ (and hence also of $\mathbf{L}_{ntf}$ in $\mathbf{L}_{inp}$).

The paper is organized as follows. Section 2 presents the syntax and semantics of the language. Section 3, 4, and 5 discuss respectively the discriminating results between $\mathbf{L}_{ntf}$ and $\mathbf{L}$, between $\mathbf{L}_{inp}$ and $\mathbf{L}_{ntf}$, and the encoding of $\mathbf{L}_{ntf,inp}$ in $\mathbf{L}_{inp}$. Section 6 reports some conclusive remarks.

2 The Syntax and the Operational Semantics

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

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

$C ::= \mathbf{0} \ \mid \ \mu.C \ \mid \ inp(a)?C\_C \ \mid \ C|C$

where:

$\mu ::= in(a) \ \mid \ out(a) \ \mid \ notify(a, C) \ \mid !in(a)$

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 denoted by C, D, ..., that share these data. A program can be a terminated program $\mathbf{0}$ (which is usually omitted for the sake of simplicity), a prefix form $\mu.P$, an *if-then-else* form $inp(a)?P\_Q$, or the parallel composition of programs.

A prefix μ can be one of the primitives $in(a)$ or $out(a)$, indicating the withdrawing or the emission of datum a respectively, and the $notify(a, P)$ 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 process P is spawned. 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 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: $inp(a)?P\_Q$ 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.

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

$$
(15) \quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad \alpha \neq \neg a, \vec{a}, \dot{a}
$$

Table 1. Operational semantics (symmetric rules omitted).

The semantics of the language is described via a labeled transition system $(Agent, Label, \longrightarrow)$ where $Label = \{\tau\} \cup \{a, \overline{a}, \neg a, \vec{a}, \dot{a} \mid a \in Name\}$ (ranged over by $\alpha, \beta, \ldots$) is the set of the possible labels. The labeled transition relation $\longrightarrow$ is the smallest one satisfying the axioms and rules in Table 1. For the sake of simplicity we have omitted the symmetric rules of $(9) - (15)$.

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) describes the output: in one step a new datum is produced and the corresponding continuation is activated. The production of this new instance of $\langle a \rangle$ is communicated to the environment by decorating this action with the label $\vec{a}$. Axiom (3) associates to the action performed by the prefix $in(a)$ the label a, which is the complementary of $\overline{a}$.

Axiom (4) indicates that $notify(a, P)$ produces a new kind of agent $on(a).P$ (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 (5) where the label $\dot{a}$ is used to describe this kind of computation step. The term $!in(a).P$ is able to activate a new copy of P by performing an action labeled with a that requires an instance of $\langle a \rangle$ to be consumed (axiom (6)).

Axioms (7) and (8) describe the semantics of $inp(a)?P\_Q$: if the required $\langle a \rangle$ is present it can be consumed (axiom (7)), otherwise its absence is guessed by performing an action labeled with $\neg a$ (axiom (8)). Rule (9) is the usual synchronization rule.

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 labelled 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 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 $\vec{a}$ indicating 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 waked-up (rule (13)); otherwise, the environment is left unchanged (rule (14)). The last rule (15) is the standard local rule that can be applied only to actions different from the non-standard $\neg a$, $\vec{a}$, and $\dot{a}$.

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 [9], condition that ensures (as proved in [9]) the existence of a unique transition system agreeing with it.

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

$$P \equiv P|\mathbf{0} \qquad P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R$$

As two structural congruent agents are observationally indistinguishable, in the remainder of the paper we will reason up to structural congruence.

In the following we will 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 τ, but also the non-standard labeled with $\neg a$ and $\vec{a}$. In fact, derivation $P \xrightarrow{\neg a} P'$ 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 [15] formulation of the semantics, but they correspond conceptually to internal steps. Formally, we define reduction steps as follows:

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

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 languages 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 Comparing $\mathbf{L}_{ntf}$ and $\mathbf{L}$

The discrimination between $\mathbf{L}_{ntf}$ and $\mathbf{L}$ is a direct consequence of the facts (1) and (2) listed in the Introduction.

The proof of (1) is a trivial adaptation of a result presented in [4]. Indeed, as we made in that paper, it is possible to define for $\mathbf{L}$ a Place/Transition net [14, 16] semantics such that for each agent P the corresponding P/T net is finite and preserves the interleaving semantics; thus, an agent can terminate if and only if the corresponding net has a terminating computation. As this property can be decided in finite P/T nets [6], we can conclude that given a process P of $\mathbf{L}$ it is decidable if $P\downarrow$.

Result (2) uses Random Access Machines (RAM) [17] which is a Turing equivalent formalism. A RAM is 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. If the program terminates, the result of the computation is the contents of the registers.

In [12] 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.

We present an encoding of RAM based on the *notify* primitive. The encoding we present is *nondeterministic* as it introduces some extra infinite computations; nevertheless, it is ensured that a RAM terminates if and only if the corresponding encoding has a terminating computation. As termination cannot be decided in Turing equivalent formalisms, the same holds also for $\mathbf{L}_{ntf}$. A question remains open in this section: "Is it possible to define in $\mathbf{L}_{ntf}$ a more adequate deterministic implementation of RAM which preserves also the divergent behaviour?". The answer is no, and it is motivated in Section 4 where we prove that the presence of an infinite computation can be decided in $\mathbf{L}_{ntf}$. On the other hand, we will show in the same Section that a deterministic implementation of RAM can be defined in $\mathbf{L}_{inp}$.

The encoding implements nondeterministically *DecJump* operations: two possible behaviours can be chosen, the first is valid if the tested register is not zero, the second otherwise. If the wrong choice is made, the computation is ensured to be infinite; in this case, we cannot say anything about the corresponding RAM. Nevertheless, if the computation terminates, it is ensured that it corresponds to the computation of the corresponding RAM. Conversely, any computation of the RAM is simulated by the computation of the corresponding encoding in which no wrong choice is performed.

$$
\begin{array}{ll}
[\![R]\!] & = [\![I_1]\!] | \dots | [\![I_k]\!] | in(loop).DIV \\
[\![i : Succ(r_j)]\!] & = !in(p_i).out(r_j).notify(zero_j, INC).out(p_{i+1}) \\
[\![i : DecJump(r_j, s)]\!] & = !in(p_i).out(loop).in(r_j).in(loop).notify(zero_j, DEC).out(p_{i+1}) \\
 & \quad |!in(p_i).out(zero_j).in(zero_j).out(p_s) \\
\text{where:} & \\
INC & = out(loop).in(match).in(loop) \\
DEC & = out(match) \\
DIV & = out(div).!in(div).out(div)
\end{array}
$$

Table 2. Encoding RAM in $\mathbf{L}_{ntf}$.

Given the RAM program R composed by the instructions $I_1 \dots I_k$ the corresponding encoding is defined in Table 2. Observe that DIV is an agent that cannot terminate; we will prove that it is activated whenever a wrong choice is made.

The basic idea of this encoding is to represent the actual content of each register r_j with a corresponding number of $\langle r_j \rangle$. Moreover, every time an increment (or a decrement) on the register r_j is performed, a new agent $on(zero_j).INC$ (or $on(zero_j).DEC$) is spawned by using the *notify* operation. In this way it is possible to check if the actual content of a register r_j is zero by verifying if the occurrences of $on(zero_j).INC$ corresponds to the ones of $on(zero_j).DEC$.

There are two possible wrong choices that can be performed during the computation: (i) a decrement on a register containing zero or (ii) a jump for zero on a non-empty register.

In the case (i), $out(loop).in(r_j).in(loop).notify(zero_j, DEC).out(p_{i+1})$ is activated with no $\langle r_j \rangle$ available. Thus, the program produces $\langle loop \rangle$ and blocks trying to execute $in(r_j)$. The produced $\langle loop \rangle$ will be not consumed and the agent DIV will be activated.

In the case (ii), the process $out(zero_j).in(zero_j).out(p_s)$ is activated when there are more occurrences of the auxiliary agent $on(zero_j).INC$ than the ones of $on(zero_j).DEC$. When $\langle zero_j \rangle$ is emitted, its production is notified to the auxiliary agents; then the corresponding processes INC and DEC start. Each

DEC emits an agent $\langle match\rangle$ while each INC produces a term $\langle loop\rangle$, and requires a $\langle match\rangle$ to be consumed before removing the emitted $\langle loop\rangle$. As there are more INC processes than DEC, one of the processes INC will block waiting for an unavailable $\langle match\rangle$; thus it will not consume its corresponding $\langle loop\rangle$. As before, DIV will be activated.

The formal proof of correctness of the encoding requires a representation of the actual state of a RAM computation: we use $(i, inc_1, dec_1, \ldots, inc_m, dec_m)$, where i is the index of the next instruction to execute while for each register index l, inc_l (resp. dec_l) represents the number of increments (resp. decrements) that have been performed on the register r_l. The actual content of r_l corresponds to $inc_l - dec_l$. In order to deal with correct configurations only, we assume that the number of increments is greater or equal than the number of decrements.

Given a RAM program R, we write

$$((i, inc_1, dec_1, \ldots, inc_n, dec_n), R) \longrightarrow ((i', inc'_1, dec'_1, \ldots, inc'_n, dec'_n), R)$$

to state that the computation moves from the first to the second configuration by performing the i^{th} instruction of R; $((i, inc_1, dec_1, \ldots, inc_n, dec_n), R) \not\longrightarrow$ means that the program R has no instruction i, i.e., the computation is terminated. As RAM computations are deterministic, given a RAM program R and a configuration $(i, inc_1, dec_1, \ldots, inc_m, dec_m)$, the corresponding computation will either terminate (denoted by $((i, inc_1, dec_1, \ldots, inc_m, dec_m), R) \downarrow$) or diverge ($((i, inc_1, dec_1, \ldots, inc_m, dec_m), R) \uparrow$). As RAM permits to model all the computable functions both the termination and the divergence of a computation are not decidable.

According to this representation technique a configuration is modeled as follows:

$$[\![(i, inc_1, dec_1, \ldots, inc_n, dec_n)]\!] =$$
$$\langle p_i\rangle | \prod_{i=1\ldots n}(\prod_{inc_i} on(zero_i).INC | \prod_{dec_i} on(zero_i).DEC | \prod_{inc_i - dec_i}\langle r_j\rangle)$$

where $\prod_{i\in I} P_i$ denotes the parallel composition of the indexed terms P_i.

It is not difficult to prove the following lemma stating that the encoding is complete as each RAM computation can be simulated by the corresponding encoding.

Theorem 1. *Let R be a RAM program, if*

$$((i, inc_1, dec_1, \ldots, inc_n, dec_n), R) \longrightarrow ((i', inc'_1, dec'_1, \ldots, inc'_n, dec'_n), R)$$

then also

$$[\![(i, inc_1, dec_1, \ldots, inc_n, dec_n)]\!] | [\![R]\!] \longrightarrow^* [\![(i', inc'_1, dec'_1, \ldots, inc'_n, dec'_n)]\!] | [\![R]\!]$$

On the other hand the encoding is not sound as it introduces infinite computations. Nevertheless, a weaker soundness for terminating computations holds.

Theorem 2. *Let R be a RAM program, if*

$$[\![(i, inc_1, dec_1, \ldots, inc_n, dec_n)]\!] | [\![R]\!] \longrightarrow^* P \not\longrightarrow$$

then $P = [\![(i', inc'_1, dec'_1, \ldots, inc'_n, dec'_n)]\!] | [\![R]\!]$ such that

$$((i, inc_1, dec_1, \ldots, inc_n, dec_n), R) \longrightarrow^* ((i', inc'_1, dec'_1, \ldots, inc'_n, dec'_n), R) \not\longrightarrow$$

Corollary 1. *Let R be a RAM program, then*

$$((i, inc_1, dec_1, \ldots, inc_n, dec_n), R) \downarrow \textit{ iff } [\![(i, inc_1, dec_1, \ldots, inc_n, dec_n)]\!] | [\![R]\!] \downarrow$$

4 Comparing $\mathbf{L}_{inp}$ and $\mathbf{L}_{ntf}$

The discrimination between $\mathbf{L}_{inp}$ and $\mathbf{L}_{ntf}$ is a direct consequence of the facts (3) and (4) listed in the Introduction.

The result (4) has been already proved in [4]. In that paper an encoding of RAM in a language corresponding to $\mathbf{L}_{inp}$ is presented. Also that encoding (that we do not report here due to the space limits) represents the content of register r_j by means of agents of kind $\langle r_j \rangle$. In this way, a *DecJump* instruction testing the register r_j can be simply implemented by means of an $inp(r_j)$ operation which either consumes an available $\langle r_j \rangle$ or observes that the register is empty. In [4] we prove that a RAM program can perform a computation step if and only if its encoding can perform the corresponding step.

In order to prove the result (3) we recall, using a notation convenient for our purposes, the definition of simple P/T nets extended with transfer arcs (see, e.g., [7]).

Definition 1. *Given a set S, we denote by $\mathcal{M}_{fin}(S)$ the set of the finite multisets on S and by $\mathcal{F}_p(S, S)$ the set of the partial functions defined on S. We use $\oplus$ to denote multiset union. A P/T net with transfer arcs is a triple $N = (S, T, m_0)$ where S is the set of* places, *T is the set of* transitions *(which are triples $(c, p, f) \in \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S) \times \mathcal{F}_p(S, S)$ such that the domain of the partial function f has no intersection with c and p), and m_0 is a finite multiset 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, $m(s)$ denotes the number of occurrences of s inside m and we say that the place s contains $m(s)$* tokens. *A P/T net with transfer arcs is finite if both S and T are finite.*

A transitions $t = (c, p, f)$ is usually written in the form $c \overset{f}{\rightarrowtail} p$ and f is omitted when empty. The marking c is called the preset *of t and represents the tokens to be* consumed. *The marking p is called the* postset *of t and represents the tokens to be* produced. *The partial function f denotes the* transfer arcs *of the transition which connect each place s in the domain of f to its image $f(s)$. The meaning of f is the following: when the transition fires all the tokens inside a place s in the domain of f are transferred to the connected place $f(s)$.*

A transition $t = (c, p, f)$ is enabled *at m if $c \subseteq m$. The execution of the transition produces the new marking m' such that $m'(s) = m(s) - c(s) + p(s) + \sum_{s':f(s')=s} m(s')$, if s is not in the domain of f, $m'(s) = \sum_{s':f(s')=s} m(s')$, otherwise. This is written as $m \xrightarrow{t} m'$ or simply $m \longrightarrow m'$ when the transition t is not relevant. We use σ, σ' to range over sequences of transitions; the empty sequence is denoted by ε; let $\sigma = t_1, \ldots, t_n$, we write $m \xrightarrow{\sigma} m'$ to mean the* firing sequence *$m \xrightarrow{t_1} \cdots \xrightarrow{t_n} m'$. The net $N = (S, T, m_0)$ has an* infinite *computation if it has a legal infinite firing sequence.*

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 tokens.

In our particular case we deal with different kinds of sequential components: programs of the form $\mu.P$ or $inp(a)?P\_Q$, agents $\langle a\rangle$, and terms $on(a,P)$ representing idle processes $on(a).P$. Besides these classes of components corresponding directly to terms of the language, we need to introduce a new kind of components $arrived(a,P)$ used to model event notification.

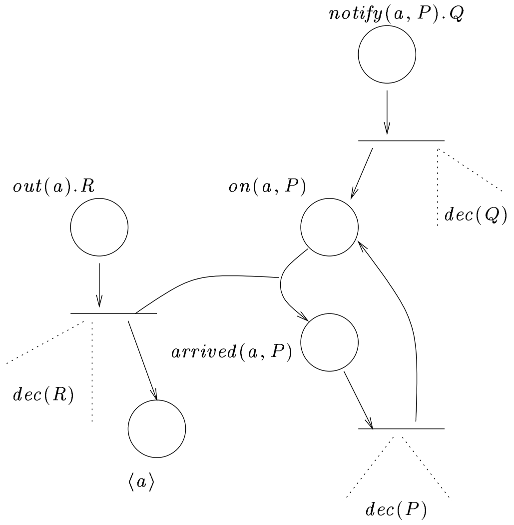

Fig. 2. Modeling event notification.

The way we represent input and output operations in our net semantics is standard. More interesting is the mechanism used to model event notification represented in Figure 2. Whenever a new token is introduced in the place $\langle a\rangle$, each token in a place $on(a,P)$ is transferred to the corresponding place $arrived(a,P)$. In order to realize this, we use a *transfer* arc that moves all the tokens inside the source place to the target one. Each token introduced in $arrived(a,P)$ will be responsible for the activation of the new instance of P. Moreover, when the activation happens, also a token in $on(a,P)$ is introduced in order to register interest in the next production of a token in $\langle a\rangle$.

The main drawback of this procedure used to model event notification is that it is not executed atomically. For instance, a new token in $\langle a\rangle$ can be produced before it terminates. In this case, the processes whose corresponding token is still in the place $arrived(a,P)$ will be not notified of the occurrence of this event. However, as we will prove in the following, even in the presence of this drawback the net semantics respects the existence of infinite computation.

After the informal description of the net semantics we introduce its formal definition. 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 $\{P \mid P \text{ sequential program}\} \cup \{\langle a \rangle \mid a \text{ message name}\} \cup \{on(a,P), arrived(a,P) \mid a \text{ message name and } P \text{ program}\}$.
- Let the function $dec : Agent \rightarrow \mathcal{M}_{fin}(\mathcal{S})$ 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 $\mathbf{0}$ generates no tokens; the decomposition of the terms $\langle a \rangle$ and of the other processes produces one token in the corresponding place; the decomposition of the idle process $on(a).P$ generates one token in place $on(a,P)$; and the parallel composition is interpreted as multiset union, i.e, the decomposition of $P|Q$ is $dec(P) \oplus dec(Q)$.

The axioms in Table 4 define the possible transitions. Axiom `in(a,Q)` deals with the execution of the primitives $in(a)$: a token from place $\langle a \rangle$ is consumed. Axiom `out(a,Q)` describes how the emission of new datum is obtained: a new token in the place $\langle a \rangle$ is introduced and the transfer arcs move all the tokens from the places $on(a,R)$ in the corresponding $arrived(a,R)$. In this way, all the idle agents are notified. The activation of the corresponding processes R requires a further step described by the axiom `arrived(a,Q)`: an instance of process Q is activated (by introducing tokens in the corresponding places) and a token is reintroduced in the place $on(a,Q)$ in order to register interest in the next token produced in $\langle a \rangle$. Axiom `!in(a,Q)` deals with the bang operator: if a token is present in place $!in(a).Q$ and a token can be consumed from place $\langle a \rangle$, then a new copy of $dec(Q)$ is produced and a token is reintroduced in $!in(a).Q$. Finally, axiom `notify(a,Q,R)` produces a token in the place $on(a,Q)$ in order to register interest in the arrival of the future incoming token in $\langle a \rangle$.

Definition 2. *Let P be an agent. We define the triple $Net(P) = (S, T, m_0)$ where:*

$$
\begin{aligned}
S &= \{Q \mid Q \textit{ sequential subprogram of } P\} \cup \\
&\quad\ \{\langle a \rangle \mid a \textit{ message name in } P\} \cup \\
&\quad\ \{on(a,Q), arrived(a,Q) \mid a \textit{ message name in } P \textit{ and } Q \textit{ subprogam of } P\} \\
T &= \{c \overset{f|_S}{\rightarrowtail} p \mid c \overset{f}{\rightarrowtail} p \in \mathcal{T} \textit{ and } dom(c) \subseteq S\} \\
m_0 &= dec(P)
\end{aligned}
$$

where by $f|_S$ we mean the restriction of function f to its subdomain S.

It is not difficult to see that $Net(P)$ is well defined, in the sense it is a correct P/T net with transfer arcs; moreover, it is finite. Moreover the net semantics is complete as it simulates all the possible computations allowed by the operational semantics.

$dec(\mathbf{0}) = \emptyset$	$dec(\langle a \rangle) = \{\langle a \rangle\}$
$dec(\mu.P) = \{\mu.P\}$	$dec(on(a).P) = \{on(a, P)\}$
$dec(P\|Q) = dec(P) \oplus dec(Q)$	

Table 3. Decomposition function.

`in(a,Q)`	$in(a).Q \oplus \langle a \rangle \rightarrowtail dec(Q)$
`out(a,Q)`	$out(a).Q \overset{f}{\rightarrowtail} \langle\langle a \rangle\rangle \oplus dec(Q)$
	where $f = \{(on(a, R), arrived(a, R)) \mid R$ is a program$\}$
`arrived(a,Q)`	$arrived(a, Q) \rightarrowtail dec(Q) \oplus on(a, Q)$
`!in(a,Q)`	$!in(a).Q \oplus \langle a \rangle \rightarrowtail !in(a).Q \oplus dec(Q)$
`notify(a,Q,R)`	$notify(a, Q).R \rightarrowtail on(a, Q) \oplus dec(R)$

Table 4. Transition specification.

Theorem 3. *Let $Net(P) = (S, T, m_0)$ and R be an agent s.t. $dom(dec(R)) \subseteq S$. If $R \longrightarrow R'$ then there exists a transition sequence σ s.t. $dec(R) \xrightarrow{\sigma} dec(R')$.*

The above theorem proves the completeness of the net semantics which, on the other hand, is not sound. Indeed, as we have already discussed, the encoding introduces some slightly different computations due to the non atomicity of the way we model the event notification mechanism. However, the introduction of these computations does not alterate the possibility to have an infinite computation. This is proved by the following Theorem.

Theorem 4. *Let $Net(P) = (S, T, m_0)$ and R an agent s.t. $dom(dec(R)) \subseteq S$. There exists an infinite firing sequence starting from $dec(R)$ iff $R\uparrow$.*

5 Comparing $\mathbf{L}_{ntf,inp}$ and $\mathbf{L}_{inp}$

In Section 3 we proved that *in* and *out* are not sufficiently powerful to encode the event notification mechanism; now we show that the addition of the *inp* operation permits to realize the encoding of $\mathbf{L}_{ntf,inp}$ in $\mathbf{L}_{inp}$.

In order to simulate event notification we force each process performing a $notify(a, P)$ to declare its interest in the incoming $\langle a \rangle$ by emitting $\langle wait_a \rangle$. Then, the process remains idle, waiting for $\langle arrived_a \rangle$, signaling that an instance of $\langle a \rangle$ appeared. When an output operation $out(a)$ is performed, a protocol composed of three phases is started.

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

$$[\![\mathbf{0}]\!] = \mathbf{0} \qquad [\![out(a).P]\!] = in(me_a).out(wc_{aP})|O(a,P)$$

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

$$[\![in(a).P]\!] = in(a).[\![P]\!] \qquad [\![on(a).P]\!] = \langle wait_a\rangle|W(a,P)|!in(w_{aP}).W(a,P)$$

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

$$[\![notify(a,P).Q]\!] = in(me_a).out(wait_a).out(w_{aP}).out(me_a).(!in(w_{aP}).W(a,P)|[\![Q]\!])$$

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

$$W(a,P) = in(arrived_a).out(wait_a).out(ack_a).out(w_{aP}).[\![P]\!]$$

$$\begin{aligned} O(a,P) = {} & !in(wc_{aP}).inp(wait_a)?(out(creating_a).out(wc_{aP}))\_(out(a).out(ca_{aP}))| \\ & !in(ca_{aP}).inp(creating_a)?(out(arrived_a).out(askack_a).out(ca_{aP}))\_out(ea_{aP})| \\ & !in(ea_{aP}).inp(askack_a)?(in(ack_a).out(ea_{aP}))\_(out(me_a).[\![P]\!]) \end{aligned}$$

Table 5. Encoding the *notify* primitive ($n(P)$ denotes the set of message names of P).

In the first phase, each $\langle wait_a\rangle$ is replaced by $\langle creating_a\rangle$. At the end of this phase $\langle a\rangle$ is produced.

In the second phase, we start transforming each $\langle creating_a\rangle$ in the pair of agents $\langle arrived_a\rangle$ and $\langle askack_a\rangle$.

The agents $\langle arrived_a\rangle$ will wake up the processes that were waiting for the notification of the addition of $\langle a\rangle$; each of these processes produces a new instance of $\langle wait_a\rangle$ (to be notified of the next emissions of $\langle a\rangle$) and an $\langle ack_a\rangle$, to inform that it has been waked. We use two separated renaming phases (from $wait_a$ to $creating_a$ and then to $arrived_a$) in order to avoid that a just waked process (that has emitted $\langle wait_a\rangle$ to be notified of the next occurrence of output of a) is waked two times.

In the third phase the $\langle ack_a\rangle$ emitted by the waked processes are matched with the $\langle askack_a\rangle$ emitted in the second phase; this ensures that all the processes waiting for emission of $\langle a\rangle$ have been waked.

The concurrent execution of two or more output protocols could provoke undesired behaviour (for example, it may happen that some waiting process is notified of a single occurrence of output, instead of two); for this reason the output protocol is performed in mutual exclusion with other output protocols producing a datum with the same name. For similar reasons we avoid also the concurrent execution of the output protocol with a notification protocol concerning the same kind of datum. This is achieved by means of $\langle me_a\rangle$, which is consumed at the beginning of the protocol and reproduced at the end.

Note that, in the implementation of this protocol, the *inp* operator is necessary in order to apply a transformation to all the occurrences of a datum in the dataspace. Indeed, with only a blocking input *in* it is not possible to solve this problem. The formal definition of the encoding is presented in Table 5.

The proof of the correctness of the encoding is essentially based on an intermediate mapping, where partially executed out and notify protocols are represented with an abstract notation. We report here only the enunciates of the main results.

The following theorem states that each move performed by a process in $\mathbf{L}_{ntf,inp}$ can be mimicked by a sequence of moves of its encoding.

Theorem 5. *Let P be a term of $\mathbf{L}_{ntf,inp}$ s.t. $n(P) \subseteq A$. If $P \longrightarrow P'$ then* $[\![P]\!]|ME_A|\prod_{i=1\ldots k} O(a_i, P_i) \longrightarrow^+ [\![P']\!]|ME_A|\prod_{i=1\ldots h} O(b_i, Q_i)$.

The next result says that any computation of the encoding of P can be extended in order to reach the encoding of a process reachable from P.

Theorem 6. *Let P be a term of $\mathbf{L}_{ntf,inp}$ s.t. $n(P) \subseteq A$.*
If $[\![P]\!]|ME_A|\prod_{i=1\ldots k} O(a_i, P_i) \longrightarrow^* Q$ then there exists P' such that $P \longrightarrow^* P'$ and $Q \longrightarrow^* [\![P']\!]|ME_A|\prod_{i=1\ldots h} O(b_i, Q_i)$.

6 Conclusion

We investigated the expressiveness of event notification in a data-driven coordination model. We proved that the addition of the *notify* primitive strictly increases the expressiveness of a language with only *in* and *out*, but leaves it unchanged if the language contains also *inp*. On the other hand, we showed that the *inp* primitive cannot be encoded by *in*, *out*, and *notify*.

We embedded the coordination primitives in a minimal language. The relevance of our results extends to richer languages in the following way. The encodability result extends to any language comprising the minimal features of our calculus. The negative results of non-encodability can be interpreted on a Turing complete language as the necessity for an encoding to exploit the specific computational features of the considered language.

We think that this kind of results has not only a theoretical relevance, but they could be of interest also for designers and implementors of coordination languages. For example, the powerful *inp* primitive has been a source of problems during the first distributed implementations of Linda (see, e.g., [10]). The results proved here suggest that the *notify* primitive may represent a good compromise between easiness of implementation and expressive power.

In [3] we consider three different interpretations for the *out* operation and in [4] we found an expressiveness gap between two of them. More precisely, we proved that a language with *in*, *out*, and *inp* is Turing powerful under the *ordered* semantics (the one considered here), while it is not under the *unordered* one (where the emission and the effective introduction of data in the dataspace are two independent steps). In [5] we investigate the impact of event notification on the unordered semantics: we prove that the addition of the *notify* primitive makes the language Turing powerful also under the *unordered* interpretation and it permits a faithful encoding of the ordered semantics on top of the unordered one.

Here, we have chosen the ordered interpretation as it is the semantics adopted by the actual JavaSpaces specifications, as indicated in the sections 2.3 and 2.8 of [18], and also confirmed us by personal communications with John McClain of Sun Microsystems Inc. [11].

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. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.
3. N. Busi, R. Gorrieri, and G. Zavattaro. Comparing Three Semantics for Linda-like Languages. *Theoretical Computer Science*, to appear. An extended abstract appeared in *Proc. of Coordination'97*.
4. N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, to appear. An extended abstract appeared in *Proc. of Express'97*.
5. N. Busi and G. Zavattaro. Event Notification in Data-driven Coordination Languages: Comparing the Ordered and Unordered Interpretation. In *Proc. of SAC2000*, ACM press. To appear.
6. A. Cheng, J. Esparza, and J. Palsberg. Complexity results for 1-safe nets. *Theoretical Computer Science*, 147:117–136, 1995.
7. C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of ICALP'98*, volume 1061 of *Lecture Notes in Computer Science*, pages 103–115. Springer-Verlag, Berlin, 1998.
8. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
9. J.F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.
10. 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.
11. J. McClain. Personal communications. March 1999.
12. M.L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, 1967.
13. G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46:329–400, 1998.
14. C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.
15. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
16. W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs in Computer Science. Springer-Verlag, Berlin, 1985.
17. J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.
18. Sun Microsystem, Inc. *JavaSpaces Specifications*, 1998.
19. Sun Microsystem, Inc. *Jini Architecture Specifications*, 1998.
20. P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998.