Electronic Notes in Theoretical Computer Science 16 No. 2 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume16.html 17 pages

Towards a Hierarchy of Negative Test Operators for Generative Communication [1]

Gianluigi Zavattaro

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

Abstract

Generative communication is a coordination paradigm that permits interprocess communication via the introduction and consumption of data to and from a shared common data space. We call negative test operators those coordination primitives able to test the absence of data in the common data space. In this paper we investigate the expressive power of this family of operators. To this aim, we concentrate on three possible primitives differing in the ability of instantaneously producing new data after the test: $tfa(a)$ tests the absence of data of kind a, $t\&e(a)$ instantaneously produces a new occurrence of datum a after having tested that no other occurrences are available, $t\&p(a, b)$ atomically tests the absence of data a and produces one instance of datum b. We prove the existence of a strict hierarchy of expressiveness among these operators.

1 Introduction

Many coordination languages allow interprocess communication via a shared data space sometimes called *Tuple Space* as in Linda [12], *Chemical Solution* as in Gamma [1], or *Blackboard* as in Shared Prolog [5].

The basic features common to these languages are:

- *Asynchronous communication:* Processes cannot directly synchronize, they only interact by means of the shared data space.
- *Anonymous data items:* After their introduction in the shared data space, data items become independent, in the sense that they are no longer related to the process that created them.
- *Associative access:* Data are accessed according to their contents.

[1] Work partially supported by Esprit Working Group n.24512 COORDINA.

This communication paradigm is usually called *generative communication* [11]. Representatives of this family of coordination languages are usually distinguished according to the type of data introduced in the shared data space, the different kind of coordination primitives, or the criteria used to realize the associative access to the data.

For example, both in Linda and Shared Prolog data are tuples, but different mechanisms are used to select the data to access: Linda uses pattern matching while Shared Prolog unification. On the other hand, the main difference between Gamma and Linda relies on the kind of coordination primitives: Gamma allows the atomic rewriting of entire multisets of data items while in Linda only one datum can be accessed at each computation step (see [19] for a formal comparison between these two coordination models).

In this paper we consider a family of coordination primitives, called *negative tests*, that have the ability of observing the absence of data. The idea to analize this kind of primitives has been inspired us by the non-blocking input operator *inp* of Linda formally modeled in [7,9]: $inp(a)?P\_Q$ activates P if at least a message a can be consumed, otherwise it behaves like Q. This operator has the ability of observing the absence of data as it activates Q only if no instance of datum a is available.

The main aim of the paper is to investigate the expressiveness of this kind of operators; in order to do this, we concentrate on three possible primitives differing in the ability of instantaneously producing new data after having performed the test for absence:

- *Test-for-absence*: $tfa(a)$
  A process $tfa(a).P$ activates its continuation P only if no instance of datum a is available. This primitive incorporates the ability of the Linda *inp* to observe the absence of data, indeed:
  $$inp(a)?P\_Q \;=\; in(a).P + tfa(a).Q$$
  where $in(a)$ consumes a datum a and $+$ is a CCS-like [14] alternative choice composition operator.

  This kind of primitive has been already considered in the setting of concurrent constraint programming in [2]. In that context the primitive is called $nask(a)$ to point out that it is the negative form of the $ask(a)$ that tests the presence of data (see also [6] for an analysis of the expressiveness of this primitive).
- *Test-and-emit*: $t\&e(a)$
  After having tested the absence of data of kind a, this primitive instantaneously produces also a new occurrence of datum a. In this way, an atomic test for absence and consequent emission of an instance of datum a is realized.

  A non-blocking variant of this operator is presented in [10]: $t\&s(a)?P\_Q$ checks the presence of datum a, if it is available P is activated, otherwise a new datum a is instantaneously produced and Q is chosen as continuation.

- *Test-and-produce*: $t\&p(a,b)$
  This primitive differs from the above test-and-emit operator for the ability of emitting an occurrence of a generic datum b potentially different from the one that has been tested. A test-and-emit primitive can be obtained by imposing b equal to a:
  $$t\&e(a).P \;=\; t\&p(a,a).P$$
  We have no knowledge of other papers considering this coordination primitive.

In order to compare the expressive power of these three negative test operators the idea is to start by considering an asynchronous version of CCS [14] (we denote by $\mathbf{L}_0$) without τ prefix, relabeling and alternative choice composition operator. It comprises only two coordination primitives $out(a)$ and $in(a)$ that produce and consume an occurrence of datum a, respectively. Then, three languages $\mathbf{L}_1$, $\mathbf{L}_2$, and $\mathbf{L}_3$ are obtained by extending $\mathbf{L}_0$ with the negative tests $tfa(a)$, $t\&e(a)$, and $t\&p(a,b)$, respectively. Finally, we show that there exists a strict hierarchy of expressiveness among these languages: $\mathbf{L}_i$ is strictly more expressive than $\mathbf{L}_j$ for any $i > j$.

From a computational point of view the four languages are equivalent because we will show that they are all Turing-powerful. As we are interested in the expressive power of coordination primitives, we adopt the approach used, e.g., in [3] and [18] to compare the relative expressiveness of concurrent languages. The idea is to investigate the possibility of encoding one language in the other up-to the preservation of some properties. We consider two classes of properties, those related to the *encoding* and those describing the *semantics* that the encoding should preserve.

In order to better understand the class of properties for the encoding, we discuss a possible approach for checking the absence of data also in a language with only inputs and outputs. The idea is to introduce a counter $Count_a$ for each kind of data a. The counters, after an appropriate initialization, could be incremented or decremented every time a new datum is produced or consumed, respectively. In this way, to check the absence of a datum a it is sufficient to verify if $Count_a$ is equal to zero. This approach has the disadvantage of introducing centralized control structures, the counters, that have to be accessed every time an operation is performed.

To prevent this, we require that no extra coordination managers or centralized control structures are introduced during the encoding. This requirement is formalized by imposing that the encoding $[\![\,]\!]$ should be modular with respect to the parallel composition operator:

$$[\![P|Q]\!] \;=\; [\![P]\!]\,|\,[\![Q]\!] \qquad \textit{program distribution preservation}$$

In [18] the notion of *uniform* encoding is considered in order to discriminate the expressive power of the synchronous π-calculus and its asynchronous fragment. Besides program distribution preservation also modularity with respect to renaming σP (that renames the free names of P according to function σ) is required: $[\![\sigma P]\!] = \sigma[\![P]\!]$. This kind of property is considered in that paper in order to ensure that the encodings preserve the *symmetry of networks*. In Section 5, we will introduce a relation $\bowtie$ for pairs of agents of our languages, that we call *symmetry*: $P \bowtie Q$ if and only if $\sigma P = Q$ for each σ belonging to a particular class of renaming functions. Instead of considering the more general renaming preservation property of [18], we will explicitly require that the encodings preserve the *symmetry* of agents:

$$P \bowtie Q \text{ implies } [\![P]\!] \bowtie [\![Q]\!] \qquad \textit{symmetry preservation}$$

It is easy to see that each encoding modular with respect to renaming preserves also the symmetry of agents. Indeed, if $P \bowtie Q$ then $\sigma P = Q$ for definition of $\bowtie$; thus, $[\![\sigma P]\!] = [\![Q]\!]$ and also $\sigma[\![P]\!] = [\![Q]\!]$ by modularity w.r.t. renaming. For this reason, we can conclude that we are dealing with a subcase of the notion of *uniform* encoding of [18].

The semantics that we consider only observes the *deadlock* and the *divergent behaviour* of agents:

$$P \Downarrow m \text{ iff } [\![P]\!] \Downarrow m \text{ for every agent } P \text{ and } m \qquad \textit{deadlock preservation}$$

$$P \Uparrow \text{ iff } [\![P]\!] \Uparrow \text{ for every agent } P \qquad \textit{divergence preservation}$$

where $P \Downarrow m$ means that there exists a computation of P that terminates/deadlocks with the shared data space in the state denoted by m, while $P \Uparrow$ indicates the existence of a non-terminating computation of P.

In the following we will denote with $\mathbf{L}_i \sqsubseteq \mathbf{L}_j$ the possibility of encoding $\mathbf{L}_i$ in $\mathbf{L}_j$ up-to the preservation of the above four properties. With $\mathbf{L}_i \sqsubset \mathbf{L}_j$ we mean that $\mathbf{L}_i \sqsubseteq \mathbf{L}_j$ and also $\mathbf{L}_j \not\sqsubseteq \mathbf{L}_i$.

The existence of the hierarchy of expressiveness is proved by showing that $\mathbf{L}_i \sqsubseteq \mathbf{L}_j$ and $\mathbf{L}_j \not\sqsubseteq \mathbf{L}_i$ for every $j = i+1$. An interesting result is that in some of the proofs of non-encodability not all the four properties are considered. For example, as schematized in the table below, to prove $\mathbf{L}_1 \not\sqsubseteq \mathbf{L}_0$ we do not take care of the symmetry and deadlock preserving properties. Indeed, we prove that there exists no program distribution preserving encoding of $\mathbf{L}_1$ in $\mathbf{L}_0$ that respects the divergent behaviour. Similarly, the proof of $\mathbf{L}_2 \not\sqsubseteq \mathbf{L}_1$ does not consider symmetry preservation.

	$\mathbf{L}_1 \not\sqsubseteq \mathbf{L}_0$	$\mathbf{L}_2 \not\sqsubseteq \mathbf{L}_1$	$\mathbf{L}_3 \not\sqsubseteq \mathbf{L}_2$
Program distribution	X	X	X
Symmetry			X
Deadlock behaviour		X	X
Divergent behaviour	X	X	X

The paper is organized as follows. Section 2 introduces syntax and semantics of $\mathbf{L}_0$. Sections 3, 4, and 5 contain the analysis of the three negative

$$(1) \quad out(a).P \xrightarrow{\tau} \langle a\rangle|P \qquad (2) \quad \langle a\rangle \xrightarrow{\overline{a}} 0$$

$$(3) \quad in(a).P \xrightarrow{a} P \qquad (4) \quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \ \alpha \neq \neg a$$

$$(5) \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad (6) \quad \frac{P \xrightarrow{\alpha} P'}{P\backslash a \xrightarrow{\alpha} P'\backslash a} \ \alpha \neq a, \overline{a}, \neg a$$

$$(7) \quad \frac{P[rec\, X.P/X] \xrightarrow{\alpha} P'}{rec\, X.P \xrightarrow{\alpha} P'}$$

Table 1
Operational semantics of $\mathbf{L}_0$ (symmetric rules of (4) and (5) omitted).

test operators $tfa(a)$, $t\&e(a)$, and $t\&p(a,b)$, respectively. Finally, Section 6 reports some comparisons with related work and conclusive remarks.

2 The Language $\mathbf{L}_0$

Language $\mathbf{L}_0$ is essentially an asynchronous version of CCS [14] (without τ prefix, relabeling and alternative choice), in which the standard input and output prefixes a and $\overline{a}$ are changed in $in(a)$ and $out(a)$, respectively.

Let $Names$, ranged over by a, b, ..., be an infinite countable set of kinds of data. We suppose $Names$ partitioned in two disjoint subsets Obs and $Unobs$ (i.e., $Names = Obs \cup Unobs$ and $Obs \cap Unobs = \emptyset$) of observable and unobservable names, respectively. The unobservable names are particular names not visible to external observers. In particular, we will use these unobservable names as auxiliary names in the encoding of $\mathbf{L}_1$ in $\mathbf{L}_2$ that we present in Section 4. Let Var, ranged over by X, Y, ..., be the set of agent variables. We define *processes* the terms obtained by the following grammar:

$$C \quad ::= \quad 0 \ | \ \mu.C \ | \ C|C \ | \ C\backslash a \ | \ X \ | \ rec\, X.C$$

where the possible prefixes μ are:

$$\mu \quad ::= \quad in(a) \ | \ out(a)$$

Term 0 denotes one inactive process, and it is usually omitted for the sake of simplicity. The possible prefixes are $in(a)$ and $out(a)$ standing for the input and output of an instance of datum a, respectively. We consider the usual parallel ($|$), restriction ($\backslash$) and recursion ($rec\, X.C$) operators. We consider only closed terms and guarded recursion [14].

Prefix $out(a)$ produces a new occurrence of datum a, denoted by $\langle a\rangle$. The input prefix $in(a)$ requires the presence of $\langle a\rangle$: if it is available, then it is removed.

Data, like $\langle a\rangle$, are not considered in the syntax of processes: we have to introduce *states*, defined as the terms obtained by the following grammar:

$$P \quad ::= \quad \langle a \rangle \;\mid\; C \;\mid\; P|P \;\mid\; P \backslash a$$

A state is the parallel composition of processes and data, with the possibility to define local names using the restriction operator. In the following P, Q, $\ldots$, are used to range over states and processes (the actual meaning will be clear by the context), and *Agent* denotes the set of possible states (called also agents in the following).

We use $fn(P)$ to denote the free names of P, i.e., those names appearing in P not under the scope of restriction. Given a function on names σ the term σP denotes the agent obtained by renaming in P each free occurrence of a with $\sigma(a)$.

The operational semantics of $\mathbf{L}_0$ is defined by means of a *labeled transition system* $(Agent_0,\ Label_0,\ \longrightarrow_0)$ specifying how states evolve. The index $_0$, as also other indexes we will introduce in the following sections, is omitted when clear by the context. $Label_0 \stackrel{def}{=} \{\tau\} \cup \{a, \overline{a} \mid a \in Names\}$ (ranged over by α, β, $\ldots$) is the set of the possible labels. The labeled transition relation $\longrightarrow_0$ is the smallest one satisfying the axioms and rules in Table 1. The side condition $\alpha \neq \neg a$ of rules (6) and (8) has no effect in $\mathbf{L}_0$ as $\neg a$ is not a legal label. The use of this side condition will be explained in the following section, where the labels $\neg a$ will be introduced.

Axiom (1) shows that an output prefix $out(a)$ can generate $\langle a \rangle$ performing an internal action labeled with τ; then $\langle a \rangle$ is able to give its contents to some process in the environment, by performing an action labeled with $\overline{a}$ (axiom (2)). Axiom (3) allows an input prefix to consume a message in the environment by performing one action labeled with a, the complementary of $\overline{a}$. The other rules are the usual ones for the parallel composition operator (rules (4) and (5)), for the restriction operator (rule (6)), and for recursion (rule (7)).

In [9] we have proved that a language, corresponding to $\mathbf{L}_0$ plus the prefix τ and the alternative choice composition operator of CCS [14], is Turing powerful by showing how to encode register machines in the language. The encoding make no use of the prefix τ and utilizes only input guarded choices. As this kind of choice is implementable in $\mathbf{L}_0$ following, e.g., the approach presented in [16], we can conclude that also $\mathbf{L}_0$ is Turing powerful. As an example of how to encode an input guarded choice operator, consider the term $in(a) + in(b)$ and its encoding:

$$\begin{array}{rl}
(\langle ok \rangle \mid in(a).(& in(ok).out(ko_b) \mid \\
 & in(ko_a).out(a)) \mid \\
in(b).(& in(ok).out(ko_a) \mid \\
 & in(ko_b).out(b)))\backslash ok \backslash ok_a \backslash ok_b
\end{array}$$

$$(8) \qquad tfa(a).P \xrightarrow{\neg a} P$$

$$(9)\ \frac{P \xrightarrow{\neg a} P' \quad Q \stackrel{\overline{a}}{\not\longrightarrow}}{P|Q \xrightarrow{\neg a} P'|Q} \qquad (10)\ \frac{P \xrightarrow{\neg a} P'}{P\backslash a \xrightarrow{\tau} P'\backslash a}$$

Table 2
Additional axiom and rules for $\mathbf{L}_1$ (symmetric rule of (9) omitted).

3 The Test-for-Absence Primitive

In this section we analize the $tfa(a)$ negative test operator. We first introduce syntax and semantics of language $\mathbf{L}_1$ which is the extension of $\mathbf{L}_0$ with the new prefix $tfa(a)$. After, we investigate the expressive power of $\mathbf{L}_1$ with respect to $\mathbf{L}_0$.

3.1 Syntax and Semantics of $\mathbf{L}_1$

The syntax of $\mathbf{L}_1$ is obtained by extending the set of prefixes of $\mathbf{L}_0$with the new $tfa(a)$:

$$\mu \ ::= \ in(a) \ | \ out(a) \ | \ tfa(a)$$

The semantics of $\mathbf{L}_1$ is defined by means of the labeled transition system $(Agent_1,\ Label_1,\ \longrightarrow_1)$. The set $Agent_1$ comprises the new agents containing prefix $tfa(a)$; the set of labels $Label_1 \stackrel{def}{=} Label_0 \cup \{\neg a \mid a \in Names\}$ contains also a new label $\neg a$ indicating that the absence of $\langle a \rangle$ is tested. Finally, the labeled transition relation $\longrightarrow_1$ is the smallest one satisfying the axioms and rules in Table 1 plus the new axiom and rules of Table 2 where $Q \stackrel{\overline{a}}{\not\longrightarrow}$ means that there exists no Q' such that $Q \xrightarrow{\overline{a}} Q'$.

Axiom (8) indicates that the execution of the test for absence is reflected by a transition step labeled with $\neg a$. A process P can perform a transition having the new label $\neg a$ when composed in parallel with an agent Q only if Q does not offer any $\langle a \rangle$ (rule (9)). Instead, if P is restricted on name a, it is no more necessary to verify the availability in the environment of $\langle a \rangle$ because the name has become local; this is the reason why the label becomes τ (rule (10)). The side condition $\alpha \neq \neg a$ of rules (4) and (6) of Table 1 is necessary in order to avoid conflicts with the new rules (9) and (10).

Rule (9) uses a negative premise; the transition system specification is strictly stratifiable [13], thus there exists a unique transition system agreeing with it.

Notation

Before analizing the expressive power of $\mathbf{L}_1$ with respect to $\mathbf{L}_0$ we need some further notation.

The relation $P \longrightarrow P'$ and its reflexive and transitive closure $P \Longrightarrow P'$ are used to indicate how agents can reduce when no environment is considered:

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

$$P \Longrightarrow P' \text{ iff } P \longrightarrow^{*} P'$$

We use $P \not\longrightarrow$ to indicate that there exists no P' such that $P \longrightarrow P'$; in other words, P cannot reduce.

Let m be the multiset of observable data $\{a_1, a_2, \dots, a_n\}$; the notation $P \downarrow m$ indicates that m is the multiset of observable data that P makes available to the outside. This can be operationally characterized as follows:

$$P \downarrow m \text{ iff } P \xrightarrow{\overline{a_1}} P_1 \xrightarrow{\overline{a_2}} P_2 \xrightarrow{\overline{a_3}} \dots \xrightarrow{\overline{a_n}} P_n \text{ for some } P_1, P_2, \dots P_n$$
$$\text{such that } P_n \stackrel{\overline{a}}{\not\longrightarrow} \text{ for any } a \in Obs$$

We will use $P \Downarrow m$ to denote the existence of a computation of P terminating in the agent P' such that P' cannot reduce and $P' \downarrow m$:

$$P \Downarrow m \text{ iff } P \Longrightarrow P' \not\longrightarrow \text{ for some } P' \text{ such that } P' \downarrow m$$

On the other hand, we use $P \Uparrow$ to indicate the existence of a non-terminating computation starting from the agent P:

$$P \Uparrow \quad \text{iff there exist } P_i \text{ with } i \in I\!N \text{ such that}$$
$$P_0 = P \text{ and } P_i \longrightarrow P_{i+1} \text{ for every } i$$

3.2 On the Expressiveness of $\mathbf{L}_1$

In order to prove that $\mathbf{L}_0 \sqsubset \mathbf{L}_1$ we first observe that $\mathbf{L}_0$ is trivially encodable in $\mathbf{L}_1$ (it is a sublanguage) and then we show that there exists no *program distribution* preserving encoding of $\mathbf{L}_1$ in $\mathbf{L}_0$ that respects at least the *divergent behaviour.*

In order to prove the non-encodability result, we consider the following agent of $\mathbf{L}_1$:

$$P \;=\; tfa(a).(rec\, X.out(b).in(b).X)\backslash b$$

Agent P tests the absence of $\langle a \rangle$; if it is not available then the divergent computation of the agent $(rec\, X.out(b).in(b).X)\backslash b$ is activated. On the other hand, if at least one $\langle a \rangle$ is present then agent P blocks. This behaviour is formalized as follows:

$$P \Uparrow \quad \textit{and} \quad P|\langle a \rangle \not\Uparrow$$

This example reflects the fact that in $\mathbf{L}_1$ the addition of new data could forbid the execution of previously available computations. This does not happen in $\mathbf{L}_0$, where the addition of new agents cannot prevent the execution of some previous available computations. This is formalized in the following Lemma.

$$(11) \qquad t\&e(a).P \xrightarrow{\neg a} \langle a\rangle|P$$

Table 3
Additional axiom for $\mathbf{L}_2$.

Lemma 3.1 *Let Q be an agent of $\boldsymbol{L}_0$. If $Q \Longrightarrow Q'$ then also $Q|R \Longrightarrow Q'|R$ for any agent R.*

Proof. As $\neg a$ is not a label in $\mathbf{L}_0$, the reduction $Q \Longrightarrow Q'$ is composed of only τ steps. The thesis directly follows from rule (4) of Table 1. $\Box$

Corollary 3.2 *Let Q be an agent of $\boldsymbol{L}_0$. If $Q \Uparrow$ then also $Q|R \Uparrow$ for every agent R.*

The non-encodability result can be now proved taking into account this corollary and the above agent P.

Theorem 3.3 *There exists no* program distribution *preserving encoding of $\boldsymbol{L}_1$ in $\boldsymbol{L}_0$ that respects the* divergent behaviour.

Proof. Suppose by contradiction that $[\![\ ]\!]$ is such an encoding. Consider the program P of $\mathbf{L}_1$ defined above, then:

$$[\![P]\!] \Uparrow \quad \textit{and} \quad [\![P]\!] \,|\, [\![\langle a\rangle]\!] \not\Uparrow$$

by divergence preservation. The fact that $[\![P]\!] \Uparrow$ leads, by Corollary 3.2, to the contradiction $[\![P]\!] \,|\, [\![\langle a\rangle]\!] \Uparrow$. $\Box$

4 The Test-and-Emit Primitive

In this section we analyze $\mathbf{L}_2$, the extension of $\mathbf{L}_0$ with the new prefix $t\&e(a)$ that atomically tests the absence of $\langle a\rangle$ and, if it is not available, instantaneously emits a new occurrence of it. We first present syntax and semantics of $\mathbf{L}_2$ and then analyze its expressivity with respect to $\mathbf{L}_1$.

4.1 Syntax and Semantics of $\boldsymbol{L}_2$

The syntax of $\mathbf{L}_2$ is obtained by extending the set of prefixes of $\mathbf{L}_0$ with the new $t\&e(a)$:

$$\mu \quad ::= \quad in(a) \;\mid\; out(a) \;\mid\; t\&e(a)$$

The semantics is defined by means of the labeled transition system ($Agent_2$, $Label_2$, $\longrightarrow_2$). The set $Agent_2$ comprises the agents containing also the new prefix $t\&e(a)$; the set of labels $Label_2$ is the same as $Label_1$; and finally $\longrightarrow_2$ is the smallest labeled transition relation satisfying the axioms and rules in Table 1 plus rules (9) and (10) of Table 2 and axiom (11) of Table 3.

Axiom (11) indicates that the new prefix operation $t\&e(a)$ is performed by means of a transition step labeled with $\neg a$; in the reached state a new $\langle a\rangle$ is made instantaneously available. In this way, the atomic test for absence and consequent emission of $\langle a\rangle$ is obtained.

4.2 On the Expressiveness of $\boldsymbol{L}_2$

In order to prove that $\mathbf{L}_1 \sqsubset \mathbf{L}_2$ we first show how to encode $\mathbf{L}_1$ in $\mathbf{L}_2$ and then we show that there exists no *program distribution* preserving encoding of $\mathbf{L}_2$ in $\mathbf{L}_1$ that preserves at least the *divergent* and *deadlock behaviour* of agents.

The first intuitive approach to encode $\mathbf{L}_1$ in $\mathbf{L}_2$ is to map each $tfa(a)$ in the sequence of actions $t\&e(a).in(a)$, where the second input action is used to remove the new instance of $\langle a\rangle$ produced by the test-and-emit operation. This idea is not correct because the occurrence of $\langle a\rangle$ produced by $t\&e(a)$ could be consumed by other agents in the environment before beeing removed by the consequent input action. In order to avoid this, we use, for every name a, all distinct auxiliary names $a', a'' \in Unobs$ to encode, with a sort of hand-shake protocol, the consumption of data. The idea is that each instance of $\langle a\rangle$ comes in pair with an extra agent $in(a').in(a).out(a'')$ which is the unique responsible of its consumption. Every time a process needs to input a datum $\langle a\rangle$, it explicitly requires to consume it (emitting $\langle a'\rangle$) and waits for the acknowledgement $\langle a''\rangle$. Formally, we define the encoding $[\![P]\!]$ inductively on the structure of P as follows:

$$\begin{array}{llll}
[\![\langle a\rangle]\!] & = \langle a\rangle | in(a').in(a).out(a'') & [\![0]\!] & = 0 \\
[\![P|Q]\!] & = [\![P]\!] | [\![Q]\!] & [\![P\backslash a]\!] & = [\![P]\!]\backslash a\backslash a'\backslash a'' \\
[\![X]\!] & = X & [\![rec X.P]\!] & = rec X.[\![P]\!] \\
[\![in(a).P]\!] & = out(a').in(a'').[\![P]\!] & [\![tfa(a).P]\!] & = t\&e(a).in(a).[\![P]\!]
\end{array}$$

$$[\![out(a).P]\!] \;=\; out(a).(in(a').in(a).out(a'')|[\![P]\!])$$

In order to prove the non-encodability of $\mathbf{L}_2$ in $\mathbf{L}_1$, we consider the problem of implementing mechanisms of *mutual exclusion* between identical agents. The idea is to consider a generic agent $Q|Q$ obtained as the parallel composition of identical terms. Then, we investigate if the agent have the ability of blocking the opposite one. We will show that this kind of mutual exclusion is not implementable in $\mathbf{L}_1$. Indeed, every time a process performs a certain kind of step, the opposite is always able to answer with the same kind of step. In this way, the computation should diverge or terminate in a state $Q'|Q'$ that is still the composition of two identical agents (thus no mutual exclusion is obtained).

Before presenting the formal proof of this result, we adapt to our setting Lemma 4.1 of [18] stating a sort of *confluence* property.

Lemma 4.1 *Let Q be an agent such that $Q \xrightarrow{a} Q'$ and $Q \xrightarrow{\overline{a}} Q''$ for some*

Q' and Q''. Then, there exists R such that $Q' \xrightarrow{\overline{a}} R$ and $Q'' \xrightarrow{a} R$.

Besides this confluence property, we need also the following lemma stating that no new data can be produced during a step labeled with $\neg a$.

Lemma 4.2 *Let Q be an agent of $\boldsymbol{L}_1$ such that $Q \stackrel{\overline{b}}{\not\longrightarrow}$. If $Q \xrightarrow{\neg a} Q'$ (for some a) then also $Q' \stackrel{\overline{b}}{\not\longrightarrow}$.*

We are now able to prove that after a step performed by an agent Q, a second instance of Q composed in parallel is always able to answer with the same kind of step.

Lemma 4.3 *Let Q be a program of $\boldsymbol{L}_1$. Then, the agent $Q|Q$ is deadlocked or there exists an agent Q' such that $Q|Q \longrightarrow^2 Q'|Q'$ (where $\longrightarrow^2$ indicates that two reduction steps are performed).*

Proof. If $Q|Q$ is deadlocked the thesis already holds. Otherwise, there exists R such that $Q|Q \xrightarrow{\alpha} R$ with $\alpha = \tau$ or $\neg a$ for some a. It is sufficient to proceed by case analysis on the last rule applied in order to derive transition $Q|Q \xrightarrow{\alpha} R$.

The possible rules are (4), (5), and (9). The first two cases are treated similarly to the proof of Theorem 4.2 of [18], where the confluence property of Lemma 4.1 is used in the case of synchronization (rule (5)).

In the case of rule (9) we have $Q \xrightarrow{\neg a} Q'$ and $Q \stackrel{\overline{a}}{\not\longrightarrow}$. By Lemma 4.2, also $Q' \stackrel{\overline{a}}{\not\longrightarrow}$. Thus, after the initial step $Q|Q \xrightarrow{\neg a} Q'|Q$ (or $Q|Q \xrightarrow{\neg a} Q|Q'$) also the other instance of Q can perform its negative test on name a; hence $Q'|Q \xrightarrow{\neg a} Q'|Q'$ (or $Q|Q' \xrightarrow{\neg a} Q'|Q'$). □

Corollary 4.4 *Let Q be a program of $\boldsymbol{L}_1$. Then there exists a computation:*

$$Q|Q \longrightarrow^2 Q_1|Q_1 \longrightarrow^2 \ldots \longrightarrow^2 Q_n|Q_n \longrightarrow^2 \ldots$$

such that:

(i) *the computation diverges or*

(ii) *there exists m such that $Q_m|Q_m$ is deadlocked.*

We are now able to present the non-encodability result.

Theorem 4.5 *There exists no* program distribution *preserving encoding of $\boldsymbol{L}_2$ in $\boldsymbol{L}_1$ that respects the* deadlock *and* divergent behaviour.

Proof. Consider the agent $t\&e(a)|t\&e(a)$ of $\mathbf{L}_2$. This agent incorporates a mutual exclusion mechanism between its two components. Indeed, after the first step is performed by one of the agents, a first instance of $\langle a\rangle$ is available; the presence of this datum blocks the opposite agent which is testing the absence of $\langle a\rangle$. The deadlock and divergent behaviour of this agent is summarized as follows:

$$t\&e(a)|t\&e(a) \Downarrow m \text{ iff } m = \{a\} \quad \textit{and} \quad t\&e(a)|t\&e(a) \not\Uparrow$$

$$(12) \qquad t\&p(a,b).P \xrightarrow{\neg a} \langle b\rangle | P$$

Table 4
Additional axiom for $\mathbf{L}_3$.

Suppose, by contradiction, the existence of a program distribution encoding $[\![\,]\!]$ of $\mathbf{L}_2$ in $\mathbf{L}_1$ that preserves the deadlock and divergent behaviour. We have that, by program distribution preservation:

$$[\![t\&e(a)|t\&e(a)]\!] = [\![t\&e(a)]\!] \mid [\![t\&e(a)]\!]$$

Let $Q = [\![t\&e(a)]\!]$. As $Q|Q = [\![t\&e(a)|t\&e(a)]\!]$ the following holds:

$$Q|Q \Downarrow m \text{ iff } m = \{a\} \qquad \text{by deadlock preservation}$$

$$Q|Q \not\Uparrow \qquad \text{by divergence preservation}$$

By Corollary 4.4 there exists a computation such that:

(i) *The computation diverges.* This leads to the contradiction $Q|Q \Uparrow$.

(ii) *The computation terminates in a deadlocked configuration $Q'|Q'$.* It is easy to see that $Q'|Q' \downarrow m' \uplus m'$ where $Q' \downarrow m'$. Then $Q|Q \Downarrow m' \uplus m'$, leading to the contradiction $m' \uplus m' = \{a\}$.

□

5 The Test-and-Produce Primitive

In this section we analyze $\mathbf{L}_3$, the extension of $\mathbf{L}_0$ with the new prefix $t\&p(a,b)$ that atomically tests the absence of $\langle a\rangle$ and, if it is not available, produces a new occurrence of $\langle b\rangle$. We first present the syntax and semantics of $\mathbf{L}_3$ and then analyze its expressivity with respect to $\mathbf{L}_2$.

5.1 Syntax and Semantics of $\mathbf{L}_3$

The syntax of $\mathbf{L}_3$ is obtained by extending the set of prefixes of $\mathbf{L}_0$ with the new $t\&p(a,b)$:

$$\mu \quad ::= \quad in(a) \;\mid\; out(a) \;\mid\; t\&p(a,b)$$

The semantics of $\mathbf{L}_3$ is defined as for $\mathbf{L}_2$ with the only difference that rule (11) of Table 3 is changed with rule (12) introduced in Table 4. This new rule permits the production of a new instance of $\langle b\rangle$ instead of $\langle a\rangle$.

5.2 On the Expressiveness of $\mathbf{L}_3$

In order to prove that $\mathbf{L}_2 \sqsubset \mathbf{L}_3$ we first observe that $\mathbf{L}_2$ is trivially encodable in $\mathbf{L}_3$ as it corresponds to the sublanguage of $\mathbf{L}_3$ in which $b = a$ in each use of the prefix $t\&p(a,b)$. After, we show that there exists no *program distribution* and *symmetry* preserving encoding of $\mathbf{L}_3$ in $\mathbf{L}_2$ that respects at least the *deadlock* and *divergent behaviour*.

First of all we need to introduce the notion of *symmetry* between agents.

Definition 5.1 *Let P and Q be two agents. They are* symmetric *(denoted by $P \bowtie Q$) if and only if there exists a* corresponding bijection $\sigma : Obs \to Obs$, *such that $\sigma(a) \neq a$ for any name a, for which $\sigma P = Q$.*

Observe that relation $\bowtie$ is symmetric because the renaming function σ is invertible and $\sigma^{-1}Q = P$.

Fact 5.2 *Let P and Q be two symmetric agents with corresponding bijection σ. Then, $P \xrightarrow{\alpha} P'$ if and only if $Q \xrightarrow{\sigma(\alpha)} Q'$ where P' and Q' are still symmetric and $\sigma(\alpha)$ is the label obtained by changing each name a appearing in α to $\sigma(a)$.*

In the proof presented in the previous section we have observed that the $t\&e(a)$ prefix permits to implement a protocol of mutual exclusion between two identical agents. In this section we consider the problem of mutual exclusion between two *symmetric* agents.

The mutual exclusion was obtained, in the previous section, by means of a competition on the execution of a $t\&e(a)$ operation on a particular name a. In the case of symmetric agents, there cannot be an initial agreement on this particular name; this follows from the fact that the bijection σ in Definition 5.1 maps each name a on a different name $\sigma(a)$ (i.e. $\sigma(a) \neq a$).

Similarly to the previous section we show that given the parallel composition of two symmetric agents of $\mathbf{L}_2$, every time a process performs a certain kind of step, the opposite is always able to answer with the symmetric one. Also in this case, the computation should diverge or terminate in a state which is still the composition of two symmetric agents. This permits to conclude that the test-and-emit primitive is not enough powerful to ensure an agreement on the name a to use to realize mutual exclusion by means of $t\&e(a)$.

Before formally proving the existence of this particular computation in which the symmetry is never broken, we need to adapt to $\mathbf{L}_2$ Lemma 4.2. In fact, the test-and-emit primitive of $\mathbf{L}_2$ permits the instantaneous production of new data, but only of the same kind of the tested one.

Lemma 5.3 *Let Q be an agent of $\boldsymbol{L}_2$ such that $Q \stackrel{\overline{b}}{\not\longrightarrow}$. If $Q \xrightarrow{\neg a} Q'$ (for some $a \neq b$) then also $Q' \stackrel{\overline{b}}{\not\longrightarrow}$.*

Lemma 5.4 *Let P and Q be two symmetric agents of $\boldsymbol{L}_2$. Then, the agent $P|Q$ is deadlocked or there exist two symmetric agents P' and Q' such that $P|Q \longrightarrow {}^2 P'|Q'$.*

Proof. As in the proof of Lemma 4.3 where Fact 5.2 is used to deal with the symmetry of P and Q instead of their identity.

The most interesting case to consider is the one in which a test-and-emit operation is executed. We have $P|Q \xrightarrow{\neg a} P'|Q$ with $P \xrightarrow{\neg a} P'$ and $Q \stackrel{\overline{a}}{\not\longrightarrow}$.

Let σ be the corresponding bijection for the symmetric agents P and Q. By Fact 5.2 we have $P \overset{\overline{\sigma(a)}}{\not\longrightarrow}$ and $Q \overset{\neg\sigma(a)}{\longrightarrow}$ The fact that $\sigma(a) \neq a$ permits to apply Lemma 5.3, hence also $P' \overset{\overline{\sigma(a)}}{\not\longrightarrow}$. Thus, agent Q can perform the symmetric test-and-emit operation testing the absence of $\sigma(a)$. $\square$

Corollary 5.5 *Let P and Q be two symmetric agents of $\boldsymbol{L}_2$. Then there exists a computation:*

$$P|Q \longrightarrow^2 P_1|Q_1 \longrightarrow^2 \ldots \longrightarrow^2 P_n|Q_n \longrightarrow^2 \ldots$$

such that:

(i) *the computation diverges or*

(ii) *there exists m such that $P_m|Q_m$ (where P_m and Q_m are symmetric) is deadlocked.*

Theorem 5.6 *There exists no* program distribution *and* symmetry *preserving encoding of $\boldsymbol{L}_3$ in $\boldsymbol{L}_2$ that respects the* deadlock *and* divergent behaviour*.*

Proof. Consider the term $t\&p(a,b)|t\&p(b,a)$ with $a,b \in Obs$, composed by the parallel composition of two symmetric agents of $\mathbf{L}_3$(with a corresponding bijection in which $a \mapsto b$ and $b \mapsto a$). The two agents realize a mechanism of mutual exclusion because after the first step is performed by one of the agents, the opposite cannot proceed because blocked by the new produced datum. The deadlock and divergent behaviour of the above agent is summarized as follows:

$$t\&p(a,b)|t\&p(b,a) \Downarrow m \text{ iff } m = \{a\} \text{ or } m = \{b\} \quad \textit{and}$$

$$t\&p(a,b)|t\&p(b,a) \not\Uparrow$$

Consider the agents $P = [\![t\&p(a,b)]\!]$ and $Q = [\![t\&p(b,a)]\!]$. We have that:

$$[\![t\&p(a,b)|t\&p(b,a)]\!] = P \mid Q \qquad \text{by program distribution preservation}$$

$$P \bowtie Q \qquad \text{by symmetry preservation}$$

At this point, the proof proceeds similarly to the one of Theorem 4.5 where Corollary 5.5 is used instead of Corollary 4.4. $\square$

6 Related Work and Conclusion

In this paper we have investigated the expressiveness of negative test primitives able to verify the absence of data in coordination languages based on generative communication. We have pointed out a strict hierarchy of expressiveness among three typical negative tests.

The comparing criterion we have adopted is inspired by the notion of *uniform encoding* preserving a *reasonable semantics* of [18]. Similarities and differences between our criterion and the one of [18] have been already discussed in Section 1. The main result presented in [18] is that the π-calculus [15] is strictly more expressive than its asynchronous fragment [4,17]. It is interesting to observe that instead of comparing synchrony with asynchrony, we have compared four asynchronous languages different in the ability of observing the absence and instantaneously emitting data. Even if we left for future work the comparison among our different levels of asynchrony and synchronous communication, we have already observed that the *leader election problem in symmetric networks* having solution in the synchronous π-calculus and not in its asynchronous subset, can be solved in our asynchronous language $\mathbf{L}_3$. Indeed, adapting the definition of [18] to our CCS-like setting, we have that the following agent P is a *symmetric network*:

$$
\begin{aligned}
P &\stackrel{def}{=} P_0|P_1|\ldots|P_{(n-1)} \\
P_i &\stackrel{def}{=} t\&p(a_i, a_{i\oplus 1}).t\&p(a_i, a_{i\oplus 2})\ldots t\&p(a_i, a_{i\oplus(n-1)}).out(w_i) \mid \\
&\quad \textstyle\prod_{j\in 0\ldots(n-1)} in(w_j).out(w_j).out(o_j)
\end{aligned}
$$

where $\oplus$ is sum modulo n and $\prod$ is used as a shorthand for the parallel composition of a set of agents. Agent P is also an *electoral system* as all the agents P_i will agree sooner or later on their leader P_j by emitting their vote $\langle o_j\rangle$. Indeed, it is not difficult to see that one and only one of the agents P_j will produce $\langle w_j\rangle$; this indicates that P_j is the winner of the competition. The presence of $\langle w_j\rangle$ has the effect of forcing each agent to vote P_j as leader, by means of the emission of $\langle o_j\rangle$.

In [8] two possible interpretations for the *out* operator of Linda are discussed: the *ordered* output, that immediately introduces the emitted data in the shared data space, and the the *unordered* one, that requires an unpredictable delay before the effective rendering of the emitted data. Using a CCS-like [14] τ prefix, the two different outputs can be modeled as follows:

$$
\begin{aligned}
out_o(a).P &\stackrel{\tau}{\longrightarrow} \langle a\rangle|P \\
out_u(a).P &\stackrel{\tau}{\longrightarrow} (\tau.\langle a\rangle)|P
\end{aligned}
$$

where out_o and out_u stands for the ordered and the unordered output, respectively. The expressive power of a Linda-like process algebra is investigated under the two interpretations; surprisingly, the calculus is Turing equivalent considering the ordered output and not under the unordered one. In this paper we have adopted the ordered output, but it is not difficult to show that the same results hold also if we move to the unordered interpretation.

The comparing criterion we have used in this paper has been used also in [19] to prove that Gamma [1] and Linda [12] are incomparable, in the sense that there exists no program distribution preserving encoding of one language in the other that respects at least the divergent and deadlock behaviour. To prove that Gamma is not encodable in Linda, we prove that it is not possible to embed in Linda the atomic consumption of a multiset of data. On the other hand, to prove that Linda cannot be encoded in Gamma, the ability of observing the absence of data (that the Linda *inp* has) plays a basic role.

Brogi and Jacquet [6] use the notion of *modular embedding* [3] to compare the relative expressiveness of all Linda dialects obtainable taking into account a subset of the following coordination primitives: *tell*, *get*, *ask* (corresponding to the Linda *out*, *in*, *rd* respectively) and $nask(a)$ (corresponding to our $tfa(a)$). The comparing criterion they use differs from ours for several aspects, e.g., they require modularity w.r.t. all the operators and do not observe the divergent behaviour. Nevertheless, the results they obtain confirm our observation that the ability of testing the absence of data increases the expressiveness of languages. Indeed, they prove that each dialect without the $nask(a)$ primitive is strictly less expressive than the one obtained adding also this operator. The different comparing criterion they use requires also different proof techniques. For example, they prove that *nask* cannot be encoded with only *get* and *tell* operations (the result corresponding to our Theorem 3.3) by taking into account modularity with respect to a sequential composition operator $P;Q$. Instead, we only consider modularity with respect to the parallel operator.

Acknowledgement
We would like to acknowledge discussions with Farhad Arbab, Frank de Boer, Marcello Bonsangue, Jan Rutten and all members of the Amsterdam Coordination Group, as also interesting communications with Antonio Brogi and Jean-Marie Jacquet on differences and similarities among criteria for the comparison of coordination languages. We are also grateful to Nadia Busi, Roberto Gorrieri and the anonymous referees for their comments.

References

[1] J.P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1): 98–111, 1993.

[2] F.S de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. Non-Monotonic Concurrent Constraint Programming. In Proc. *International Logic Programming Symposium*, pages 315–334, The MIT press, 1993.

[3] F.S de Boer, C. Palamidessi. Embedding as a Tool for Language Comparison: On the CSP Hierarchy. In Proc. *Concur'91*, volume 527 of *LNCS*, pages 127–141, 1991.

[4] G. Boudol. Asynchrony and the π-calculus. Technical Report 1702, INRIA, Sophia–Antipolis, 1992.

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

[6] A. Brogi and J.M. Jacquet. On the Expressiveness of Linda-like Concurrent Languages. In Proc. *Express'98*, volume 16 of *ENTCS*, 1998.

[7] N. Busi, R. Gorrieri, and G. Zavattaro. Three Semantics of the Output Operation for Generative Communication. In Proc. *Coordination'97*, volume 1282 of *LNCS*, pages 205–219, 1997.

[8] N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing Equivalence of Linda Coordination Primitives. In Proc. *Express'97*, volume 7 of *ENTCS*, 1997.

[9] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2): 167–199, 1998.

[10] N. Busi, R. Gorrieri, and G. Zavattaro. Comparing Three Semantics for Linda-like Languages. To appear in *Theoretical Computer Science*.

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

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

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

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

[15] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.

[16] U. Nestmann and B.C. Pierce. Decoding Choice Encodings. In *CONCUR'96*, volume 1119 of *LNCS*, pages 179–194. Springer, 1996.

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

[18] C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus. In Proc. *POPL'97*, pages 256–265, ACM, 1997.

[19] G. Zavattaro. On the Incomparability of Gamma and Linda. Technical Report (to appear), Centrum voor Wiskunde en Informatica, Amsterdam, 1998.