On the incomparability of Gamma and Linda

G. Zavattaro

Software Engineering (SEN)

SEN-R9827 October 1998

Report SEN-R9827
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

On the Incomparability of Gamma and Linda

G. Zavattaro*
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract
We compare Gamma and Linda, two of the most prominent coordination languages based on generative communication via a shared data space. In Gamma computation is obtained by applying multiset rewriting rules, reminiscent of the way chemical reactions happen in a solution. On the other hand, Linda permits interprocess communication by means of the creation and consumption of shared data. Also a non-blocking input operator is allowed: it terminates indicating if the required datum has been consumed or it is not actually available. We first recall two simple calculi based on Gamma and Linda coordination models. Even if both the languages are Turing powerful, we show that their expressive power is not comparable; in particular, we prove that there exists no program distribution preserving encoding of one language in the other that respects at least the input-output behaviour.

1991 Mathematics Subject Classification: 68N99, 68Q10
1991 Computing Reviews Classification System: D.3, F.1, I.1
Keywords & Phrases: Coordination languages and models, generative communication, expressiveness of concurrent languages

1 Introduction

The term *coordination* is used to indicate a new class of models, formalisms, and mechanisms for describing concurrent and distributed activities [CH96]. Basically, coordination is achieved either by generative communication via a shared data space (e.g. Linda [GC92] and Gamma [BM93]) or by dynamically evolving the interconnections among the processes as a consequence of observations of their state changes (e.g. Manifold [BABRSZ98]).

A recent survey [PA98] lists at least 30 different coordination languages and models. This amount of different proposals requires also the definition of criteria for their comparison and classification. This paper represents a first step towards the analysis of such this possible criteria. In particular, we compare Gamma [BM93] and Linda [GC92], two of the most prominent representative of the family of coordination languages based on generative communication via a shared data space.

In Gamma [BM93] computation is described by defining multiset rewriting rules, reminiscent of the way chemical reactions happen in a solution. For example, a program able to find the maximum element in a non-empty multiset of integers can be written in Gamma as follows:

$$GammaMaximum \ : \ \{x_1, x_2\} \longrightarrow \{x_1\} \quad \Longleftarrow \quad x_1 \geq x_2$$

The computation consists of the repeated execution of a rewriting rule that consumes an available pair of elements a and b (respectively matched by the variables x_1 and x_2) such that $a \geq b$ and produce a new occurrence of a. The computation terminates when only one element is available; this element is the maximum.

*Visiting from the University of Bologna, Italy.

Linda [GC92] consists of a set of primitives allowing interprocess communication via the introduction and consumption of data items to and from a shared data space. We consider three of the coordination primitives provided by Linda: $out(a)$ (emits an occurrence of datum of kind a), $in(a)$ (consumes an occurrence of a), $inp(a)$ (non-blocking predicative version of $in(a)$: if the required datum is available it is consumed and $true$ is returned, otherwise it terminates returning $false$). A Linda variant of the above Gamma program can be written as follows:

$$
\begin{array}{lll}
LindaMaximum & : & in(x_1) \\
 & & \text{while } inp(x_2) \\
 & & \quad \text{if } x_1 \geq x_2 \text{ then } out(x_1) \text{ else } out(x_2) \\
 & & \quad in(x_1) \\
 & & \text{end while} \\
 & & return(x_1)
\end{array}
$$

Observe that the while-loop terminates whenever no data are available; when the control exits the loop, the last consumed element is returned as it is the maximum.

In order to compare the Gamma and Linda coordination models we first recall two process algebraic representations of these languages given in [CGZ96] and [BGZ97b, BGZ98], respectively. The semantics is presented, as in [CGZ96], by means of a two-level transition system; the lower level describes the semantics of programs (denoted usually by P) while the upper level describes how a configuration $\langle P, m\rangle$, composed by the program P and the multiset m of data available in the shared data space, can evolve.

The small Linda calculus we use is proven to be Turing equivalent in [BGZ97b]. In this paper we show that also the simple Gamma calculus we consider is Turing powerful. Even if the languages are computationally equivalent, we compare them analyzing the possibility of encoding one language in the other up-to the preservation of some properties. Our comparing criterion is essentially a weakening of the notion of *uniform encoding preserving a reasonable semantics* used in [Pal97] to discriminate the expressive power of π-calculus [MPW92] and its asynchronous fragment [Bou92, HT91].

First of all we require that the *distribution of programs* is preserved by the encodings; more formally, we want that the encodings (usually denoted by $[\![\,]\!]$) are compositional at least with respect to the parallel composition operator (denoted by $|$ in both the languages):

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

The main reason to consider this requirement is to forbid the possibility of introducing new external managers or extra-coordinators. In fact, without the program distribution preservation requirement we could think to define encodings of the following kind:

$$[\![P_1|\ldots|P_n]\!] \;=\; Q|[\![P_1]\!]\,|\ldots|[\![P_n]\!]$$

where the programs $[\![P_1]\!]$, ..., $[\![P_n]\!]$ could use the new program Q as a controller for their interactions.

The encoding should also preserve some notion of *observational* semantics of programs. We consider as observable only the input-output behaviour. In particular we consider the (i) divergent and (ii) deadlock/termination behaviours: more formally, we require that, given the configuration $\langle P, m\rangle$, (i) it has a divergent computation if and only if also $\langle [\![P]\!], m\rangle$ has and (ii) it has a computation terminating in a state m' if and only if also $\langle [\![P]\!], m\rangle$ has.

We prove that Gamma and Linda are not comparable according to this comparing criterion. More precisely, we show that there exists no program distribution preserving encoding of one language in the other that respects the input-output behaviour.

The impossibility of encoding Linda in Gamma is rather intuitive and not difficult to prove. It directly follows from the *monotonicity* of Gamma: if a Gamma program P is able to transform the data space m_1 in m_2, it is also able to transform $m_1 \uplus m'$ in $m_2 \uplus m'$ for every m' ($\uplus$ denotes multiset union). Monotonicity does not hold in Linda as it permits to observe the absence of data by means of the inp primitive. Indeed, suppose that $inp(a)$ activates a particular computation; if

an instance of a is introduced in the data space, the considered computation cannot be activated any more.

On the other hand, the impossibility of encoding Gamma in Linda is less intuitive and it is more difficult to prove. It essentially follows from the fact that multiset rewriting rules are atomically executed in Gamma. This ensures that no conflict happens between two rewriting rules that can be both applied on the same multiset. On the other hand, Linda permits to consume or produce only *one* single data item in an atomic way. Usually, the atomicity of operations involving multisets of data is obtained in Linda by using synchronization mechanisms such as semaphores or monitors. This mechanisms are not built-in in Linda and their implementation usually requires the introduction of explicit managers. For example, a classical way to obtain mutual exclusion between two concurrent programs P and Q is to introduce a fresh token t (name t does not occur neither in P nor Q) that must be consumed before one of the programs is started:

$$out(t) \mid in(t).P \mid in(t).Q$$

It is immediately clear that this approach uses an extra-coordinator (the process $out(t)$) that is avoided by the above program distribution preserving property.

The paper has the following structure: in Section 2 we present syntax and semantics of the calculi based on Gamma and Linda we consider; Section 3 presents the formal proof of the incomparability result while Section 4 reports comparisons with related work and some concluding remarks.

2 The Languages and their Semantics

We recall two process algebraic representations of Gamma [BM93] and Linda [GC92] given in [CGZ96] and [BGZ97b, BGZ98], respectively. We present in a fresh way the operational semantics for the Linda calculus using the two-levels approach followed in [CGZ96] for the Gamma calculus: we first present the semantics of programs and then we describe how programs interact with the shared data space at the coordination level. This approach has been recently used also in the definition of a formal semantics for the coordination language Manifold [BABRSZ98].

2.1 The Language Gamma

Let *Reaction*, ranged over by R, R_i, ..., and *Action*, ranged over by A, A_i, ..., be two sets of reactions and actions, respectively. Let Γ, ranged over by P, Q, ..., be the set of Gamma programs defined by the following abstract syntax:

$$Programs_\Gamma \qquad P \ ::= \ (R, A) \ \mid \ P|P$$

A Gamma program P is the parallel composition of basic programs (R, A).

Let $\mathcal{N}$ be a countable set of names ranged over by a, b, ..., and let $\mathcal{M}(\mathcal{N})$, ranged over by m, m_i, ..., be the set containing the multisets on $\mathcal{N}$. Names are used to indicate the possible kinds of data: in the following $\langle a \rangle$ denotes an occurrence of a datum of kind a. Given two sets A and B, we denote with $\mathcal{F}(A, B)$ the set of functions from A to B.

We consider the existence of $\mathcal{I}_R$ and $\mathcal{I}_A$, two *interpretation functions* for reactions and actions, respectively:

$$\begin{array}{lcll} \mathcal{I}_R & : & Reactions & \longrightarrow \ \mathcal{F}(\mathcal{M}(\mathcal{N}), Boolean) \\ \mathcal{I}_A & : & Actions & \longrightarrow \ \mathcal{F}(\mathcal{M}(\mathcal{N}), \mathcal{M}(\mathcal{N})) \end{array}$$

Given a program (R, A), $\mathcal{I}_R(R)$ and $\mathcal{I}_A(A)$ are two functions: given a multiset of names m if $\mathcal{I}_R(R)(m) = true$ then m can be rewritten as indicated by $\mathcal{I}_A(A)(m)$.

Consider, as example, the Gamma program $P = (R, A)$ where:

$$\begin{array}{lcl} \mathcal{I}_R(R) & : & \{a, a\} \mapsto true \\ \mathcal{I}_A(A) & : & \{a, a\} \mapsto \{b\} \end{array}$$

This Gamma program is able to produce $\langle b \rangle$ every time two $\langle a \rangle$ are available for consumption.

An alternative description for Gamma programs $P = (R, A)$ we use in the following is:

$$P \ : \ m \longrightarrow \mathcal{I}_A(A)(m) \quad \Longleftarrow \quad \mathcal{I}_R(R)(m)$$

The alternative description of the example above is:

$$P \ : \ \{a, a\} \longrightarrow \{b\} \quad \Longleftarrow \quad true$$

2.1.1 Γ is Turing Powerful

We present how to encode in Gamma any Random Access Machine (RAM) [SS63], a well known Turing equivalent formalism. This permits to conclude that Gamma is Turing powerful; to the best of our knowledge we are not aware of previous papers analyzing the computational power of Gamma.

The translation for RAM we present is inspired by the one for the Linda Process Algebra (LINPA) presented in [BGZ97b].

A RAM is composed by a set of registers r_j holding arbitrary large natural numbers, and a sequence of numbered instructions.

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

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

If the contents of register r_j is n, we indicate this by means of $\langle j_n \rangle$, e.g, $\langle j_0 \rangle$ indicates that r_j is empty. The program counter is instead described by $\langle p_i \rangle$, indicating that the following instruction to execute is the i^{th} one.

A *Succ* instruction can be easily implemented:

$$[\![i : Succ(r_j)]\!] \quad : \quad \{p_i, j_n\} \longrightarrow \{p_{i+1}, j_{n+1}\} \quad \Longleftarrow \quad n \geq 0$$

For the *DecJump* instruction we consider the parallel composition of two Gamma programs: the first performs the decrement instruction and the second executes the jump instruction:

$$[\![i : DecJump(r_j, l)]\!] \quad = \quad P|Q$$

with:

$$\begin{array}{lcll} P & : & \{p_i, j_n\} \longrightarrow \{p_{i+1}, j_{n-1}\} & \Longleftarrow \quad n > 0 \\ Q & : & \{p_i, j_0\} \longrightarrow \{p_l, j_0\} & \Longleftarrow \quad true \end{array}$$

2.1.2 Operational Semantics

The operational semantics of Γ is defined by a labelled transition system $(\Gamma, Label_\Gamma, \longrightarrow_\Gamma)$. The subscript Γ is omitted when clear by the context. The set of labels $Label_\Gamma = \{m_1 \overline{m_2} \mid m_1, m_2 \in \mathcal{M}(\mathcal{N})\}$ is ranged over by α, β, ... A label $m_1 \overline{m_2}$ indicates that the multiset rewriting operation that transforms m_1 in m_2 can be performed.

The labelled transition relation $\longrightarrow_\Gamma$ is the minimal one satisfying the axiom and rule in Table 1. Axiom (1) indicates that the basic program (R, A) can perform a transition labelled with $m_1 \overline{m_2}$ if it is able to rewrite the multiset m_1 as m_2. Rule (2) ensures that P can perform its rewriting operation also when put in a context $P|Q$.

It is easy to observe that transitions of programs does not change the program itself. In other words, the transitions are *self-loops*.

Fact 2.1 Let P be a program of Γ. If $P \xrightarrow{\alpha} P'$ then $P' = P$. ■

(1)	$(R, A) \stackrel{m_1\overline{m_2}}{\longrightarrow} (R, A)$	$\mathcal{I}_R(R)(m_1) = true$ and $\mathcal{I}_A(A)(m_1) = m_2$
(2)	$\dfrac{P \stackrel{\alpha}{\longrightarrow} P'}{P\|Q \stackrel{\alpha}{\longrightarrow} P'\|Q}$	

Table 1: Transition system specification for Γ (symmetric rule of (2) omitted).

(1)	$in(a).P \stackrel{a}{\longrightarrow} P$	(2)	$out(a).P \stackrel{\overline{a}}{\longrightarrow} P$
(3)	$inp(a)?P\_Q \stackrel{a}{\longrightarrow} P$	(4)	$inp(a)?P\_Q \stackrel{\neg a}{\longrightarrow} Q$
(5)	$!in(a).P \stackrel{a}{\longrightarrow} !in(a).P\|P$	(6)	$\dfrac{P \stackrel{\alpha}{\longrightarrow} P'}{P\|Q \stackrel{\alpha}{\longrightarrow} P'\|Q}$

Table 2: Transition system specification for $\mathcal{L}$ (symmetric rule of (6) omitted).

2.2 A Subset of Linda

In [BGZ98] a Linda Process Algebra, called LINPA, is introduced; in [BGZ97b] a fragment of LINPA is shown to be Turing powerful. We recall this fragment of LINPA that we denote with $\mathcal{L}$. The programs of the language $\mathcal{L}$, ranged over by P, Q, ..., are defined by the following abstract syntax:

$$\begin{array}{lrcl} Programs_{\mathcal{L}} & P & ::= & 0 \mid \mu.P \mid inp(a)?P\_P \mid P|P \\ Prefixes_{\mathcal{L}} & \mu & ::= & out(a) \mid in(a) \mid !in(a) \end{array}$$

The unique primitive program is 0 representing inaction; it is usually omitted for the sake of simplicity. Structured programs are prefix forms $\mu.P$, if-then-else forms $inp(a)?P\_Q$, or parallel composed programs $P|Q$. The possible prefixes are $out(a)$, $in(a)$, and $!in(a)$. The first two prefixes represent the emission and the consumption of $\langle a \rangle$, respectively. Last prefix is used for unbounded replication of terms guarded on input operations: the program $!in(a).P$ is able to consume an unbounded amount of data of kind a, everytime activating a new program P. The if-then-else form $inp(a)?P\_Q$ represents a program that requires to consume a data of kind a: if it is available it is consumed and the program P is activated, otherwise Q is chosen.

For example, the program:

$$P \;=\; inp(a)?0\_out(b)$$

consumes an occurrence of $\langle a \rangle$, if available, otherwise creates a new occurrence of $\langle b \rangle$.

The operational semantics of the language $\mathcal{L}$ is described by a labelled transition system ($\mathcal{L}$, $Label_{\mathcal{L}}$, $\longrightarrow_{\mathcal{L}}$). Also the underscript $\mathcal{L}$ is omitted when clear by the context.

The set of labels $Label_{\mathcal{L}} = \{a, \overline{a}, \neg a \mid a \in \mathcal{N}\}$, is ranged over by α, β, ... The first two kinds of labels stand for input and output actions, respectively, while the third label $\neg a$ indicates that the absence of data of kind a is tested.

The transition relation $\longrightarrow_{\mathcal{L}}$ is the minimal one satysfying the axioms and rules of Table 2. Axioms (1) and (2) describe the semantics of input and output prefixes, respectively; a term $in(a).P$ requires to consume a datum of kind a by performing a transition labelled with a, while $out(a).P$ produces a new datum of kind a by means of a step labelled with $\overline{a}$. Axiom (3) and (4) indicate that a program $inp(a)?P\_Q$ is able to consume a datum of kind a and become P, or to test its absence by means of a transition labelled with $\neg a$ and become Q. Axiom (5) states that the term $!in(a).P$ can spawn a new instance of the program P by consuming a datum of kind

$$\begin{array}{lll} (1) & \dfrac{P \stackrel{m_1\overline{m_2}}{\longrightarrow} P'}{\langle P, m \uplus m_1\rangle \longrightarrow \langle P', m \uplus m_2\rangle} & \\ (2) & \dfrac{P \stackrel{\neg a}{\longrightarrow} P'}{\langle P, m\rangle \longrightarrow \langle P', m\rangle} & a \notin m \end{array}$$

Table 3: Transition system specification for the coordination level.

a. Rule (6) indicates that a program P able to perform a transition can perform it also when in parallel with other programs.

Observe that in the transition system no synchronization rules are considered. This reflects the fact that we are dealing with an asynchronous language in which programs cannot directly synchronize but they only interact by means of the coordination medium as described in the following.

2.3 The Coordination Level

In this section we described a unified coordination level for both Gamma and Linda programs.

A configuration representing the state of a computation at the coordination level is composed by the active program and the data actually available. Formally, the possible configurations are represented by the terms composing the set $\mathcal{C}$, ranged over by C, D, ..., defined by the following abstract syntax:

$$Configurations \qquad C \;::=\; \langle P, m\rangle$$

where P can be either a Gamma or a Linda program and m is a multiset of names representing the data actually available.

The computations at the coordination level are described by a transition system $(\mathcal{C}, \longrightarrow_{\mathcal{C}})$ parameterized on the possible transitions of programs described at the program level. Also the underscript $\mathcal{C}$ is omitted when made clear by the context.

For the sake of conciseness we consider the labels a and $\overline{a}$ as special cases of the more general labels $m_1\overline{m_2}$: $a = \{a\}\overline{\emptyset}$ and $\overline{a} = \emptyset\overline{\{a\}}$. The transition relation $\longrightarrow_{\mathcal{C}}$ is the minimal one satisfying the rules in Table 3. Rule (1) permits the consumption and the introduction in the coordination medium of new data. Rule (2) indicates that a transition testing the absence of data of kind a can be performed only if no data of this kind are available in the shared data space.

As example of computation at the coordination level we recall the Gamma program P presented above:

$$P \;:\; \{a, a\} \longrightarrow \{b\} \quad \Longleftarrow \quad true$$

The transition system at the program level ensures that $P \stackrel{m_1\overline{m_2}}{\longrightarrow} P$ with $m_1 = \{a, a\}$ and $m_2 = \{b\}$. From rule (1) of Table 3 immediately follows that:

$$\langle P, \{a, a\}\rangle \longrightarrow \langle P, \{b\}\rangle$$

A similar program, able to repeatedly consume two data $\langle a\rangle$ and produce $\langle b\rangle$, can be written also in the language $\mathcal{L}$:

$$Q \;=\; !in(a).in(a).out(c)$$

The transition system at the program level ensures:

$$Q \stackrel{a}{\longrightarrow} Q_1 \stackrel{a}{\longrightarrow} Q_2 \stackrel{\overline{b}}{\longrightarrow} Q_3 \stackrel{a}{\longrightarrow} Q_4 \ldots$$

Thus, at the coordination level we have:

$$\langle Q, \{a, a\}\rangle \longrightarrow \langle Q_1, \{a\}\rangle \longrightarrow \langle Q_2, \emptyset\rangle \longrightarrow \langle Q_3, \{b\}\rangle$$

This example could suggest that Gamma programs could be easily encoded in Linda, but we will show that this is not true in general. Consider, for example, the Gamma program $P = P_1|P_2$ where:

$$\begin{array}{lcll} P_1 & : & \{a, a\} \longrightarrow \{b\} & \Longleftarrow \quad true \\ P_2 & : & \{a, a\} \longrightarrow \{c\} & \Longleftarrow \quad true \end{array}$$

Consider now this program P in the configuration $\langle P, \{a, a\}\rangle$. It is ensured that the computation terminates in one of the configurations $\langle P, \{b\}\rangle$ or $\langle P, \{c\}\rangle$.

Following the above approach in order to encode Gamma programs in Linda we obtain $Q = Q_1|Q_2$ where:

$$\begin{array}{lcl} Q_1 & = & !in(a).in(a).out(b) \\ Q_2 & = & !in(a).in(a).out(c) \end{array}$$

This encoding is not satisfactory because it could introduce new deadlocks. Consider the configuration $\langle Q, \{a, a\}\rangle$. If Q_1 and Q_2 independently consume their first $\langle a\rangle$, then a deadlock is reached because the data space becomes empty and all the involved processes require to consume data.

The non-blocking input *inp* can be used to solve (partially) this problem. Consider the new encoding $Q' = Q'_1|Q'_2$ with:

$$\begin{array}{lcl} Q'_1 & = & !in(a).inp(a)?out(b)\_out(a) \\ Q'_2 & = & !in(a).inp(a)?out(c)\_out(a) \end{array}$$

Each program consumes its first $\langle a\rangle$ and, instead of blocking if the second one is not available, $\langle a\rangle$ is reemitted and a new attempt to consume both the data is tried. In this way, the undesired deadlock is avoided.

Even if new deadlocks are not introduced, also this encoding is not satisfactory because it introduces divergences. Consider the case in which both the programs consume their first $\langle a\rangle$; if the next two computation steps consist of the execution of the two non-blocking *inp* operations, both of them will fail. At this point, the initial state can be reached by reintroducing in the data space both the consumed data.

In the next section we will prove that there exists no program distribution preserving encoding of programs of Γ in $\mathcal{L}$ that preserves the input-output behaviour of programs. It is interesting to note that also the inverse holds, i.e., no encoding of $\mathcal{L}$ in Γ exists.

3 Comparing the Languages

As already indicated in the previous section, we consider the input-output behaviour. In particular, we observe the possibility of divergent computations or the existence of a computation that terminates/deadlocks with the data space in a certain state. Formally, consider the configuration $C = \langle P, m\rangle$; its divergence and deadlock/termination behaviour is defined as follows:

$$\begin{array}{ll} C \Uparrow & \text{iff there exist } C_i \text{ with } i \in I\!N \text{ such that} \\ & C_0 = C \text{ and } C_i \longrightarrow C_{i+1} \text{ for every } i \\ C \Downarrow m' & \text{iff } C \longrightarrow^* C' \text{ with } C' \not\longrightarrow C'' \text{ for any } C'' \text{ and} \\ & C' = \langle P', m'\rangle \text{ for some } P' \end{array}$$

In the previous section we have presented two proposals for encoding Γ in $\mathcal{L}$ and we have observed that none of them is satisfactory, as the first introduce new deadlocks and the second add divergences. Before presenting the proof of impossibility of defining a "satisfactory" encoding $[\![\,]\!]$ mapping programs of Γ in programs of $\mathcal{L}$ (and vice versa) we have to formalize our notion of "satisfactory".

First of all we require that the input-output behaviour is preserved:

$$\begin{array}{llll} \langle P, m\rangle \Uparrow & \text{iff} & \langle [\![P]\!], m\rangle \Uparrow & \text{for any } m \qquad \textit{divergence preservation} \\ \langle P, m\rangle \Downarrow m' & \text{iff} & \langle [\![P]\!], m\rangle \Downarrow m' & \text{for any } m \text{ and } m' \qquad \textit{deadlock/termination preservation} \end{array}$$

Moreover, we require also that the distribution of programs is preserved and no external managers or extra-coordinators are introduced during the encodings; more formally, we want that the encoding is compositional with respect to the parallel operator (observe that $|$ is used to denote parallel composition in both the languages we are dealing with):

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

3.1 Encoding $\mathcal{L}$ in Γ

We start showing the impossibility of encoding $\mathcal{L}$ in Γ. This result follows from the *monotonicity* of Gamma: the addition of data in the shared data space cannot forbid the possibility of executing previously available computations.

Lemma 3.1 *(Monotonicity)* Let Q be a program of Γ. If $\langle Q, m\rangle \longrightarrow \langle Q', m'\rangle$ for some m, Q', and m' then also $\langle Q, m \uplus m''\rangle \longrightarrow \langle Q', m' \uplus m''\rangle$ for every m''. ■

In the following, we will use two corollaries of the monotonicity lemma. The first, related to the *big-bang* property of Gamma [San93], follows from the fact that if a process is able to produce new data starting from the empty data space, then its computation surely diverges.

Corollary 3.2 Let Q be a program of Γ. If $\langle Q, \emptyset\rangle \Downarrow m$ then $m = \emptyset$. ■

The second corollary states that if a process has a divergent computation starting from the data space m, then it can diverge also if the data space is $m \uplus m'$ (for every multiset m').

Corollary 3.3 Let Q be a program of Γ. If $\langle Q, m\rangle \Uparrow$ then also $\langle Q, m \uplus m'\rangle \Uparrow$ for every multiset m'. ■

We now prove the impossibility of defining an encoding of $\mathcal{L}$ in Γ that preserves the deadlock/termination behaviour.

Theorem 3.4 There exists no encoding of Γ in $\mathcal{L}$ that respects the deadlock/termination behaviour.
Proof Consider the program $out(a)$ of $\mathcal{L}$ and observe that $\langle out(a), \emptyset\rangle \Downarrow \{a\}$. Suppose, by contradiction, that $[\![\,]\!]$ is an encoding that preserves the deadlock/termination behaviour; hence, $\langle [\![out(a)]\!], \emptyset\rangle \Downarrow \{a\}$ contradicting Corollary 3.2. ■

It is interesting to note that the program distribution preservation property is not considered in the above theorem. The impossibility result holds also if only the divergent behaviour is considered. In this case the proof requires the definition of the following program of $\mathcal{L}$ that has the property of diverging if and only if no $\langle a\rangle$ is available:

$$P \;=\; inp(a)?0\_(out(b)|!in(b).out(b))$$

Program P chooses its else branch $out(b)|!in(b).out(b)$ (which is a divergent program) if no $\langle a\rangle$ is present in the data space. Otherwise, $\langle a\rangle$ is consumed. From the point of view of the divergent behaviour we have:

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

The following theorem shows that there exists no program in Γ presenting this divergent behaviour.

Theorem 3.5 There exists no encoding of Γ in $\mathcal{L}$ that respects the divergent behaviour.
Proof Suppose by contradiction that $[\![\,]\!]$ is such an encoding. Consider the program P of $\mathcal{L}$ as defined above, then:

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

by divergence preservation. The fact that $\langle [\![P]\!], \emptyset\rangle \Uparrow$ implies, by Corollary 3.3, the contradiction $\langle [\![P]\!], \{a\}\rangle \Uparrow$. ■

3.2 Encoding Γ in $\mathcal{L}$

We now analyse the problem of encoding Γ in $\mathcal{L}$; we show that this is not possible if we require to preserve the program distribution, and we want to forbid the introduction of new deadlock or divergences. In this case the proof is more complex and uses all the three requirements introduced above. Intuitively, we proceed in the following way. We recall our running Gamma example:

$$P \ : \ \{a, a\} \longrightarrow \{b\} \quad \Longleftarrow \quad true$$

and we observe that:

$$\langle P|P, \{a, a\}\rangle \not\Downarrow \quad and \quad \langle P|P, \{a, a\}\rangle \Downarrow m \text{ iff } m = \{b\}$$

In fact, only one of the programs P will be able to perform its rewriting operation.

On the other hand, we show that given a Linda program $Q|Q$ of $\mathcal{L}$ which is the parallel composition of two identical programs, at each computation step of one program the other one is able to reply with the same step.

Fact 3.6 For Q in $\mathcal{L}$, $Q|Q \xrightarrow{\alpha} Q'$ implies $Q' = Q''|Q$ (or $Q' = Q|Q''$) and $Q \xrightarrow{\alpha} Q''$. ■

Lemma 3.7 Let Q be a program of $\mathcal{L}$ and let $\langle Q|Q, m \uplus m\rangle$ be a configuration. Then, the configuration is deadlocked or there exists a program Q' and a multiset m' such that $\langle Q|Q, m\rangle \longrightarrow \cdot \longrightarrow \langle Q'|Q', m' \uplus m'\rangle$ (where $\longrightarrow \cdot \longrightarrow$ indicates that two computation steps are performed).

Proof If the configuration $\langle Q|Q, m \uplus m\rangle$ is deadlocked, the thesis trivially holds.

Otherwise there exists a configuration C such that $\langle Q|Q, m \uplus m\rangle \longrightarrow C$. We proceed by case analysis on the rule of Table 3 used to prove the transition.

If rule (1) is used then an input or an output operation is performed. In the case of input we have $Q|Q \xrightarrow{a} Q'$ for some a such that $a \in m \uplus m$. This ensures $a \in m$; the fact that the data space is structured as union of two identical multisets ensures that another $\langle a\rangle$ is available. An occurrence of datum of kind a is removed: $C = \langle Q', (m \uplus m) \setminus \{a\}\rangle$ ($\setminus$ indicates multiset difference). Let $m' = m \setminus \{a\}$, then $(m \uplus m) \setminus \{a\} = m \uplus m'$. By Fact 3.6 we have $Q' = Q''|Q$ (or $Q' = Q|Q''$) and $Q \xrightarrow{a} Q''$. Then, the second subprogram Q can consume the other occurrence of $\langle a\rangle$: $C \longrightarrow \langle Q''|Q'', m' \uplus m'\rangle$. The case of output operation is treated symmetrically.

If rule (2) is used then $Q|Q \xrightarrow{\neg a} Q'$ for some a such that $a \notin m \uplus m$. Moreover, the data space is left unchanged: $C = \langle Q', m \uplus m\rangle$. By Fact 3.6 we have that $Q' = Q''|Q$ (or $Q' = Q|Q''$) and $Q \xrightarrow{\neg a} Q''$. The fact that the data space is left unchanged ensures that also the second subprogram Q can perform the transition $Q \xrightarrow{\neg a} Q''$ (indeed $a \notin m \uplus m$), thus $C \longrightarrow \langle Q''|Q'', m \uplus m\rangle$ is an available transition. ■

Corollary 3.8 Let Q be a program of $\mathcal{L}$. Then there exists a computation:

$$\langle Q|Q, m \uplus m\rangle \xrightarrow{\alpha_1} \cdot \xrightarrow{\alpha_1} C_1 \xrightarrow{\alpha_2} \cdot \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} \cdot \xrightarrow{\alpha_n} C_n \xrightarrow{\alpha_{n+1}} \cdot \xrightarrow{\alpha_{n+1}} \ldots$$

such that:

1. the computation diverges or;
2. there exists a deadlocked configuration C_m such that $C_m = \langle Q'|Q', m' \uplus m'\rangle$ for some Q' and m'. ■

Theorem 3.9 There exists no program distribution preserving encoding of Γ in $\mathcal{L}$ that respects the input-output behaviour.

Proof Suppose by contradiction that $[\![\,]\!]$ is such an encoding. Consider the running Gamma example P defined above and observe that:

$$[\![P|P]\!] = [\![P]\!] \,|[\![P]\!] \qquad \text{by program distribution preservation}$$

Let $Q = [\![P]\!]$, then $Q|Q = [\![P|P]\!]$. We have also:

$$\begin{array}{ll} \langle Q|Q, \{a,a\}\rangle \not\Uparrow & \text{by divergence preservation} \\ \langle Q|Q, \{a,a\}\rangle \Downarrow m \text{ iff } m = \{b\} & \text{by deadlock/termination preservation} \end{array}$$

As $\{a,a\} = \{a\} \uplus \{a\}$, Corollary 3.8 can be applied to the configuration $\langle Q|Q, \{a,a\}\rangle$. Thus, there are two cases to analize:

1. *The computation diverges.* This implies the contradiction $\langle Q|Q, \{a,a\}\rangle \Uparrow$.
2. *The computation terminates in the deadlocked configuration* $\langle Q'|Q', m' \uplus m'\rangle$. This implies $\langle Q|Q, \{a,a\}\rangle \Downarrow m' \uplus m'$; leading to the contradiction $m' \uplus m' = \{b\}$. ■

4 Related Work and Conclusion

Gamma and Linda are two of the most prominent representative of the family of coordination languages based on the shared-memory model. We have provided a uniform two-level presentation of two process algebraic representations of Gamma and Linda previously introduced in [CGZ96] and [BGZ97b, BGZ98], respectively. The main result we present is that the two languages are incomparable: there exists no program distribution preserving encoding of one language in the other respecting at least the input-output behaviour.

It is interesting to observe that the proof of non-encodability of Gamma in Linda essentially shows the impossibility of having in Linda a satisfactory implementation of *multiset* input operations such as, e.g., a $min(a_1, \ldots, a_n)$ primitive that consumes the data items $a_1, \ldots, a_n$. Intuitively, one could think to realize this kind of operations using well-established transaction protocols such as, e.g., the two- or three-phase commit algorithms. In this paper we formally prove that even this approach cannot permits to obtain completely satisfactory solutions; indeed, either new external managers or coordinator processes are needed, or deadlocks (or divergences) are introduced.

The idea to compare the relative expressive power of concurrent languages by studying the possibility of encoding one language in the others up-to the preservation of some properties has been recently used also in [Pal97], to discriminate the expressive power of π-calculus [MPW92] and its asynchronous fragment [Bou92, HT91], in [Zav98], to study the expressiveness of the so-called *negative test* operators, and in [BJ98], to prove separation and equivalence results regarding the expressiveness a class of Linda dialects.

In [Pal97] Palamidessi proves that there exists no *uniform encoding* of the π-calculus in its asynchronous fragment that preserves any *reasonable semantics*. The notion of uniform encoding requires modularity w.r.t. both the parallel composition operator and the substitution of free names, while the notion of reasonable semantics corresponds to the observation of particular actions performed on particular channels. Our comparing criterion is essentially a weakening of this one, as we do not consider name substitution and we analyze only the input-output behaviour of programs without taking care of intermediate actions.

We have used the comparison criterion presented in this paper also in [Zav98] to investigate the expressive power of the so-called *negative test* operators. In that paper we prove that a $tfa(a)$ operator (that tests if no message $\langle a\rangle$ is available) strictly increases the expressive power of a language with only *in* and *out*. The $tfa(a)$ operator permits to model the *inp* Linda primitive as follows: $inp(a)?P\_Q = in(a).P + tfa(a).Q$ (where + is a CCS-like alternative choice). Moreover, we show that the addition of another operator $t\&e(a)$, able to instantaneously produce a new instance of $\langle a\rangle$ after the execution of the negative test, further increases the expressiveness of the language. We left for future work the investigation of the possibility of implementing Gamma in a Linda-like language extended with the $t\&e(a)$ operation.

Brogi and Jacquet [BJ98] uses a notion inspired by the *modular embedding* of [BP91] to compare the relative expressiveness of all Linda-like languages obtainable taking into account a subset of the following coordination primitives: *tell*, *get*, *ask* (corresponding to the Linda *out*, *in*, *rd* respectively) and a (non-Linda) *nask* (the same as the above tfa operator) which is the negative form of *ask*. The comparing criterion that they use differs from ours for several aspects: e.g.,

they require compositionality w.r.t. all the operators (also an alternative choice operator +), do not observe the divergent behaviour, and allow also an encoding for the shared data space. These differences require also different proof techniques. Nevertheless, the results they obtain confirm our observation on the fact that the possibility of testing the absence of messages increases the expressiveness of languages. Indeed, they prove that each dialect without the *nask* primitive is strictly less expressive than the one obtained adding also this command.

The expressiveness of operators such as the above $min(a_1, \ldots, a_n)$ has been recently investigated also by Nestmann [Nes98]. In that paper, a *joint input*, inspired by the join-calculus [FG96], is added to the asynchronous π-calculus [HT91, Bou92]. This new operator permits to atomically receive from two separated channels. The expressiveness of the join input is investigated by showing possible encodings of the mixed guarded choice of the (synchronous) π-calculus in the obtained language. The fact that multiset input operators strictly increases the expressive power is formally proved in this paper, even in a more basic way; indeed, we consider a language containing also a negative test operator.

We left for future work the analysis of the expressive power of other coordination primitives such as the sequential composition operator for Gamma programs introduced in [HLS93] and revisited in [CGZ96].

Acknowledgement

We would like to acknowledge interesting discussions with Farhad Arbab, Marcello Bonsangue, and all members of the Amsterdam Coordination Group. A particular thanks is devoted to Jan Rutten for his precious suggestions on earlier versions of the paper. We thank David Sands for his indication about the *big-bang* property of Gamma, and to Oscar Nierstrasz for the suggestion to use transaction protocols in order to realize multiset rewritings in Linda. We are also grateful to Nadia Busi and Roberto Gorrieri for their comments.

References

[BM93] J.P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1): 98–111, 1993.

[BP91] 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.

[BABRSZ98] M.M. Bonsangue, F. Arbab, J.W. de Bakker, J.J.M.M. Rutten, A. Scutellà, G. Zavattaro. A transition system semantics for a control-driven coordination language. Forthcoming.

[Bou92] G. Boudol. Asynchrony and the π-calculus. Technical Report 1702, INRIA, Sophia–Antipolis, 1992.

[BJ98] A. Brogi and J.-M. Jacquet. On the Expressiveness of Linda-like Concurrent Languages. In Proc. *Express'98*, volume 16 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 1998.

[BGZ97a] 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.

[BGZ97b] N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing Equivalence of Linda Coordination Primitives. In Proc. *Express'97*, volume 7 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 1997.

[BGZ98] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2): 167–199, 1998.

[CH96] P. Ciancarini and C. Hankin, editors. Proc. of *Coordination'96*. Volume 1061 of *LNCS*, 1996.

[CGZ96] P. Ciancarini, R. Gorrieri, and G. Zavattaro. An Alternative Semantics for the Parallel Operator of the Calculus of Gamma Programs. In *Large Coordination Programming: Mechanism, Models and Semantics*, pages 232–248, Imperial College Press, 1996.

[FG96] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join-calculus. In Proc. *POPL'96*, pages 372–385, ACM, 1996.

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

[HLS93] C. Hankin, D. Le Métayer and D. Sands. A Parallel Programming Style and its algebra of Programs. In Proc. *Parle'93*, volume 694 of *LNCS*, pages 367–378, 1993.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.

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

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

[Nes98] U. Nestmann. On the Expressive Power of Join Input. In Proc. *Express'98*, volume 16 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 1998.

[Pal97] C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus. In Proc. *POPL'97*, pages 256–265, ACM, 1997.

[PA98] G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. Forthcoming.

[San93] D.Sands. Laws of synchronised termination. In Proc. of *1st Imperial College Departement of Computing Workshop on Theory and Formal Methods*, pages 276–288, Springer, 1993.

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

[Zav98] G. Zavattaro. Towards a Hierarchy of Negative Test Operators for Generative Communication. In Proc. *Express'98*, volume 16 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 1998.