# On the Expressiveness of Coordination Models

Antonio Brogi[1] and Jean-Marie Jacquet[2]

[1] Department of Computer Science, University of Pisa, Italy
[2] Institut d'Informatique, Facultés Universitaires de Namur, Belgium

**Abstract.** A number of different coordination models for specifying inter-process communication and synchronisation rely on a notion of shared dataspace. Many of these models are extensions of the Linda coordination model, which includes operations for adding, deleting and testing the presence/absence of data in a shared dataspace.
We compare the expressive power of three classes of coordination models based on shared dataspaces. The first class relies on Linda's communication primitives, while a second class relies on the more general notion of multi-set rewriting (e.g., like Bauhaus Linda or Gamma). Finally, we consider a third class of models featuring communication transactions that consist of sequences of Linda-like operations to be executed atomically (e.g., like in Shared Prolog or PoliS).

## 1 Introduction

### 1.1 Motivations

As motivated by the constant expansion of computer networks and illustrated by the development of distributed applications, the design of modern software systems centers on re-using and integrating software components. The corresponding paradigm shift from stand-alone applications to interacting distributed systems calls for well-defined methodologies and tools for integrating heterogeneous software components.

One of the key issues in this perspective is a clear separation between the *interaction* and the *computation* aspects of software components. Such a separation was advocated by Gelernter and Carriero in [19] as a promising approach to master the complexity of large applications, to enhance software reusability and to ease global analysis. The importance of separating interaction and computation aspects may be also summarised by Wegner's provocative argument that "interaction is more important than algorithms" [27].

Accordingly, the last decade has seen an increasing attention towards models and languages which support a neat separation of the design of individual software components from their interaction. Such models and languages are often referred to as *coordination models and languages*, respectively [13, 18].

Linda [10] was the first coordination language, originally presented as a set of inter-agent communication primitives which may be virtually added to any programming language. Besides process creation, this set includes primitives for adding, deleting, and testing the presence/absence of data in a shared dataspace.

P. Ciancarini and A.L. Wolf (Eds.): COORDINATION'99, LNCS 1594, pp. 134-149, 1999.

A number of other coordination models have been proposed after Linda. Some of them extend Linda in different ways, for instance by introducing multiple dataspaces and meta-level control rules (e.g., Bauhaus Linda [11], Bonita [23], $\mu Log$ ([20]), PoliS [12], Shared Prolog [6]), by addressing open distributed systems (e.g., Laura [26]), middleware web-based environments (e.g., Jada [14]), or mobility (e.g., KLAIM [16]). A number of other coordination models rely on a notion of shared dataspace, e.g., Concurrent Constraint Programming [24], Gamma [3], Linear Objects [2] and Manifold [1], to cite only a few. A comprehensive survey of these and other coordination models and languages has been recently reported in [22].

The availability of a considerable number of coordination models and languages stimulates a natural question:

*Which is the best language or model for expressing coordination issues?*

Of course the answer depends on what we mean by the "best" model. A formal way of specifying this question is to reformulate it in terms of the expressive power of models and languages.

### 1.2 Comparing the expressive power of coordination languages

As pointed out in [15], from a computational point of view all "reasonable" sequential programming languages are equivalent, as they express the same class of functions. Still it is common practice to speak about the "power" of a language on the basis of the expressibility or non-expressibility of programming constructs. In general [17], a sequential language $L$ is considered to be more expressive than another sequential language $L'$ if the constructs of $L'$ can be translated in $L$ without requiring a "global reorganisation of the program", that is, in a compositional way. Of course the translation must preserve the meaning, at least in the weak sense of preserving termination.

When considering concurrent languages, the notion of termination must be reconsidered as each possible computation represents a possible different evolution of a system of interacting processes. Moreover *deadlock* represents an additional case of termination. De Boer and Palamidessi introduced in [15] the notion of *modular embedding* as a method to compare the expressive power of concurrent languages.

In this paper we use the notion of modular embedding to compare the relative expressive power of three classes of coordination languages that employ data-driven communication primitives. The first family, denoted by $\mathcal{L}_L$, is based on a set of communication primitives à la Linda: *tell*, *get*, *ask*, and *nask* for respectively adding, deleting, and checking the presence and the absence of data in a shared dataspace. The second family, denoted by $\mathcal{L}_{MR}$, adopts an alternative view of these primitives by considering them as the rewriting of pre- and post-conditions on a shared data space, namely as multi-sets of *tell*, *get*, *ask*, and *nask* operations. The third family, denoted by $\mathcal{L}_{CS}$, imposes an order on the evaluation of the primitives, hence introducing communication sequences to be evaluated atomically as "all-or-nothing" transactions.

All the languages considered contain sequential, parallel and choice operators. For each family (viz., $\mathcal{L}_L$, $\mathcal{L}_{MR}$, $\mathcal{L}_{CS}$) we consider three different languages that differ from one another in the set $\mathcal{X}$ of communication primitives used, syntactically denoted by a set parameter. For instance, if $\mathcal{X}$ is the set $\{ask, tell\}$ then the language $\mathcal{L}_L(\mathcal{X})$ is Linda restricted to *ask* and *tell* operations, and it corresponds to a basic form of concurrent constraint programming [24]. Analogously (set brackets will be omitted for the ease of reading), $\mathcal{L}_L(ask, get, tell)$ corresponds to non-monotonic concurrent constraint programming [4], while $\mathcal{L}_L(ask, nask, get, tell)$ corresponds to Linda without process creation. Moreover, $\mathcal{L}_{MR}(ask, get, tell)$ corresponds to Gamma [3], $\mathcal{L}_{MR}(ask, nask, get, tell)$ extends Gamma with negative (non-local) pre-conditions , while $\mathcal{L}_{CS}(ask, nask, get, tell)$ generalises the communication transactions introduced in Shared Prolog [6].

As just suggested, the families $\mathcal{L}_L$, $\mathcal{L}_{MR}$, and $\mathcal{L}_{CS}$ are thus representatives of a substantial amount of coordination languages. We turn in this paper to an exhaustive pair-wise comparison of the expressive power of the languages obtained by taking $\mathcal{X}$ as $\{ask, tell\}$, $\{ask, get, tell\}$, and $\{ask, nask, get, tell\}$, for each of the three classes.

### 1.3 Results of the comparisons

It is easy to see that a number of (modular) embeddings can be trivially established by considering sub-languages. For instance, for any considered class of languages (viz., for any possible subscript of $\mathcal{L}$):

$$\mathcal{L}(ask, tell) \;\leq\; \mathcal{L}(ask, get, tell) \;\leq\; \mathcal{L}(ask, nask, get, tell)$$

holds, where $L' \leq L$ denotes that $L'$ can be (modularly) embedded by $L$. However, the most interesting results are *separation* results, where a language is shown to be strictly more powerful than another language, and *equivalence* results, where two languages are shown to have the same expressive power.

An expected result proved in the paper is that the above disequalities are strict in the sense that, on the one hand, it is not possible to simulate the destructive *get* primitives via *ask* and *tell* operations and, on the other hand, it is not possible to reduce *nask* tests to *ask*, *get*, and *tell* primitives. Hence for instance concurrent constraint programming languages are strictly less expressive than their non-monotonic versions, which are in turn strictly less expressive than Linda.

Another interesting result is that, for any subset $\mathcal{X}$ of communication primitives, $\mathcal{L}_L(\mathcal{X}) < \mathcal{L}_{MR}(\mathcal{X})$. This establishes that Linda without *nask* operations is strictly less expressive than Gamma. Similarly, for each $\mathcal{X}$, $\mathcal{L}_L(\mathcal{X}) < \mathcal{L}_{CS}(\mathcal{X})$, which shows that the introduction of communication transactions strictly increases the expressive power of languages. However, $\mathcal{L}_L(ask, nask, get, tell)$ and $\mathcal{L}_{MR}(ask, get, tell)$ are incomparable, which proves that full Linda and Gamma are incomparable.

It is interesting to observe that communication transactions get more and more expressiveness as they are enriched with primitives, as evidenced by the

following relations:

$$\mathcal{L}_{CS}(ask, tell) < \mathcal{L}_{MR}(ask, tell)$$
$$\mathcal{L}_{CS}(ask, get, tell) = \mathcal{L}_{MR}(ask, get, tell)$$
$$\mathcal{L}_{CS}(ask, nask, get, tell) > \mathcal{L}_{MR}(ask, nask, get, tell)$$

Finally, it worth observing that $\mathcal{L}_{CS}(ask, nask, get, tell)$ is the most expressive languages of the nine languages under study.

Our study of the languages is complete in the sense that all possible relations between pairs of languages have been analysed. For each pair of languages we have established whether they have the same expressive power ($L = L'$), or one is strictly more powerful than the other ($L < L'$), or none of the above two cases holds (i.e., $L$ and $L'$ are incomparable).

This study provides useful insights for both the theory and the practice of coordination-based approaches. Indeed, the resulting hierarchy depicted in figure 4 shows the equivalence of different models and indicates which extensions may be worth considering because of their additional expressive power.

### 1.4 Related work

The specificities of our work may be highlighted by contrasting it with related work. The closest pieces of work are [29] and [28].

The expressiveness of four coordination languages is analysed in [29]. Using our terminology, they are obtained by enriching the language $L_0 = \mathcal{L}_L(get, tell)$ with three forms of negative tests: $nask(a)$ which tests for the absence of $a$, $t\&e(a)$ which instantaneously produces $a$ after testing that $a$ is not present, and $t\&p(a, b)$ which atomically tests for the absence of $a$ and produces an instance of $b$. Consequently, the first extension $L_1$ is $\mathcal{L}_L(nask, get, tell)$, which is proved equivalent in [7] to $\mathcal{L}_L(ask, nask, get, tell)$. The second extension $L_2$ is a restricted version of the language $\mathcal{L}_{CS}(nask, get, tell)$ reduced by considering as communication primitives operations of the form $[get(t)]$, $[tell(t)]$, and $[nask(t); tell(t)]$, where the $[\cdots]$ construct denotes a communication transaction. Finally, the third extension $L_3$ is obtained by allowing communication transactions of the form $[nask(t); tell(u)]$ for possibly different data $t$ and $u$. In [29] the languages are compared on the basis of three properties: compositionality of the encoding with respect to parallel composition, preservation of divergence and deadlock, and a symmetry condition. It is worth noting that the resulting hierarchy $L_0 < L_1 < L_2 < L_3$ is consistent with our results. Similar properties are used in [28] to establish the incomparability of Linda and Gamma.

Compared to our work, we shall use compositionality of the encoding with respect to sequential composition, choice, and parallel composition operator. We will use the preservation of termination marks too, and require an element-wise decoding of the set of observables. However, in contrast to [29] and [28], we shall be more liberal with respect to the preservation of termination marks in requiring these preservations on the store resulting from the execution from the empty store of the coded versions of the considered agents and not on the same

store. In particular, these ending stores are not required to be of the form $\sigma \cup \sigma$ if this is so for the stores resulting from the agents themselves. Moreover, as the reader may appreciate, this paper presents a wider comparison of a larger class of languages, which requires new proof techniques at the technical level.

The paper [5] compares nine variants of the $\mathcal{L}_L(ask, nask, get, tell)$ language. They are obtained by varying both the nature of the shared data space and its structure. On the one hand, one distributed model and two centralised models, preserving or not the order in which data values are produced, are proposed. On the other hand, a multi-set structure, a set structure, and a list structure of the dataspace are considered. Rephrased in the [15] setting, this amounts to considering different operational semantics. In contrast, we fix an operational semantics and compare different languages on the basis of this semantics. The goals are thus different, and call for completely different treatments and results.

In [9], a process algebraic treatment of a family of Linda-like concurrent languages is presented. A lattice of eight languages is obtained by considering different sets of primitives out of *ask*, *get*, *tell* primitives, cited above, and conditional *ask* and *get* variants. The authors also show that this lattice collapses to a smaller four-points lattice of different bisimulation-based semantics. Again, compared to our work, different semantics are considered whereas we shall stick to one semantics and compare languages on this basis. Different goals are thus aimed, which call also for different results and treatments.

Busi, Gorrieri and Zavattaro also recently studied in [8] the issue of Turing-completeness in Linda-like concurrent languages. They define a process algebra containing Linda's communication primitives and compare two possible semantics for the *tell* primitive: an ordered one, with respect to which the execution of *tell* is considered to be finished when the data has reached the dataspace, and an *unordered* one for which *tell* terminates just after having sent the insertion request to the dataspace. The main result presented in [8] is that the process algebra is not Turing-complete under the second interpretation of *tell*, while it is so under the first interpretation. Besides the fact that we tackle in this paper a broader class of languages, including among others the $\mathcal{L}_{MR}$ and $\mathcal{L}_{CS}$ family, the work [8] and ours are somehow orthogonal. While [8] studies the *absolute* expressive power of different variants of Linda-like languages (using Turing-completeness as a yard-stick), we study the *relative* expressive power of different variants of such languages (using modular embedding as a yard-stick).

Finally, this paper extends the exhaustive comparison of the languages in $\mathcal{L}_L$, that was reported in [7].

### 1.5 Plan of the paper

The remainder of the paper is organised as follows. Section 2 formally defines the syntax of the three classes of concurrent languages considered, while section 3 defines their operational semantics. Section 4 introduces the notion of modular embedding proposed in [15]. Section 5 contains the exhaustive comparison of the expressive power of the languages. The presentation of the propositions establishing the results of the comparisons is preceded by an informal analysis of

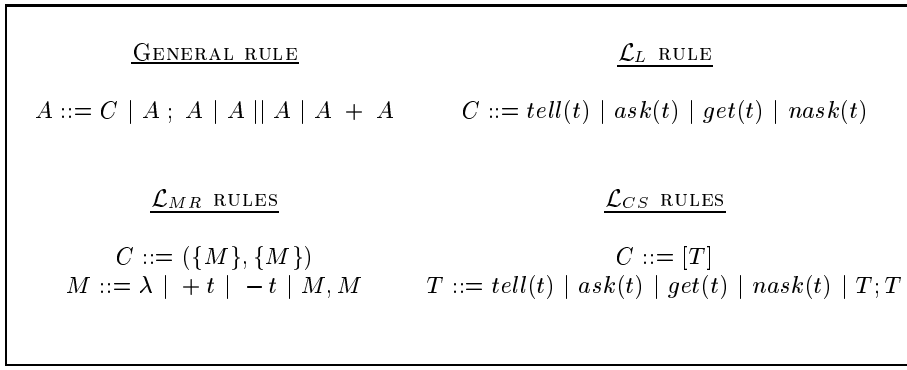


**Fig. 1.** Comparative syntax of the languages.

the results from a programming point of view. Figure 4 summarises the results presented in this section. Finally section 6 contains some concluding remarks.

## 2 The family of coordination languages

### 2.1 Common syntax and rules

We shall consider a family of languages $\mathcal{L}(\mathcal{X})$, parameterised w.r.t. the set of communication primitives $\mathcal{X}$. The set is in turn a subset of a general set of communication primitives *Slcom*, *Smcom*, or *Stcom* depending on the family under consideration. Assuming this general set, all the languages use sequential, parallel, and choice operators (see "General rule" in figure 1), whose meaning is defined by the usual rules (S), (P), and (C) in figure 2.

### 2.2 $\mathcal{L}_L$: Linda

The first family of languages is the Linda-like family of languages.

**Definition 1.** *Define the set of communication primitives Slcom as the set of C's generated by the $\mathcal{L}_L$ rule of figure 1. Moreover, for any subset $\mathcal{X}$ of Slcom, define the language $\mathcal{L}_L(\mathcal{X})$ as the set of agents A generated by the general rule of figure 1. The transition rules for these agents are the general ones of figure 2 together with rules (T), (A), (N), (G) of that figure.*

Rule (T) states that an atomic agent $tell(t)$ can be executed in any store $\sigma$, and that its execution results in adding the token $t$ to the store $\sigma$. ($E$ denotes the empty agent.) Rules (A) and (N) state respectively that the atomic agents $ask(t)$ and $nask(t)$ can be executed in any store containing the token $t$ and not containing $t$, and that their execution does not modify the current store. Rule (G) also states that an atomic agent $get(t)$ can be executed in any store containing an occurrence of $t$, but in the resulting store the occurrence of $t$ has been deleted. Note that the symbol $\cup$ actually denotes multiset union.

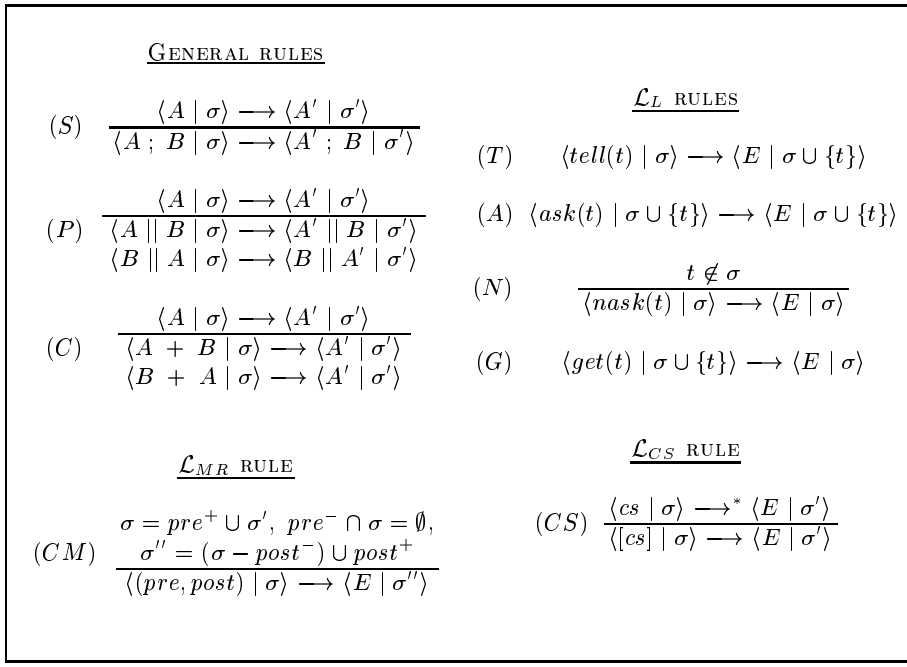


**Fig. 2.** Comparative semantics of the languages.

### 2.3 $\mathcal{L}_{MR}$: Multi-set rewriting

The transition rules $(T)$, $(A)$, $(N)$, and $(G)$ suggest an alternative view of Linda-like communication primitives in terms of which conditions the current store should obey to allow the transitions to occur and which modifications these transitions make on the store.

A natural dual view of communication primitives is then to consider them as the rewriting of pre-conditions into post-conditions. We shall consequently examine, as a second family, languages based on multi-set rewriting. It is here worth noting that this approach has already been taken in [3, 11, 21].

Each communication primitive thus consists of a multi-set of pre-conditions and of a multi-set of post-conditions. Pre- and post-conditions are (possibly empty) multi-sets of positive and negative tuples. Intuitively speaking, the operational effect of a multi-set rewriting $(pre, post)$ is to insert all positive post-conditions and to delete all negative post-conditions from the current dataspace $\sigma$, provided that $\sigma$ contains all positive pre-conditions and does not contain any of the negative pre-conditions. For instance, the operational effect of the multi-set rewriting $(\{+a, -b\}, \{+c, -d\})$ is to add $c$ and delete $d$ from the current dataspace $\sigma$ provided that $\sigma$ contains $a$ and does not contain $b$.

Given a multi-set rewriting $(pre, post)$ we shall denote by $pre^+$ the multi-set $\{t \mid +t \in pre\}$ and by $pre^-$ the multi-set $\{t \mid -t \in pre\}$. The denotations $post^+$ and $post^-$ are defined analogously.

A multi-set rewriting $(pre, post)$ is *consistent* iff $pre^+ \cap pre^- = \emptyset$. A multi-set rewriting $(pre, post)$ is *valid* if $post^- \subseteq pre^+$, where $\subseteq$ denotes multi-set inclusion.

**Definition 2.** *Define the set of multi-set communication primitives Smcom as the set of C's engendered by the $\mathcal{L}_{MR}$ rules of figure 1. Given a subset $\mathcal{X}$ of Smcom, define the language $\mathcal{L}_{MR}(\mathcal{X})$ as the set of A's generated by the general rule of figure 1.*

As a result of restricting to consistent and valid multi-set communication primitives, four basic pairs of pre and post-conditions are only possible: $(\{+t\}, \{\})$, $(\{-t\}, \{\})$, $(\{\}, \{+t\})$, $(\{+t\}, \{-t\})$. We shall respectively identify them to $ask(t)$, $nask(t)$, $tell(t)$, and $get(t)$.

For our comparison purposes, given $\mathcal{X}$ a subset of communication primitives of *Slcom*, we shall abuse notations and denote by $\mathcal{L}_{MR}(\mathcal{X})$ the language obtained by restricting multi-set rewriting pairs to component-wise multi-set unions of pairs associated with the communication primitives of $\mathcal{X}$. For instance, if $\mathcal{X} = \{ask, nask\}$, then the language $\mathcal{L}_{MR}(\mathcal{X})$ only involves pairs of the form $(Pre, \{\})$ where $Pre$ may contain positive and negative tokens. Similarly, if $\mathcal{X} = \{tell, get\}$ then $\mathcal{L}_{MR}(\mathcal{X})$ includes only pairs of the form $(Pre, Post)$ where $Pre$ contain positive tokens only provided that each one is associated with one negative counterpart in $Post$ and $Post$ contain negative tokens provided each one is associated to one positive token in $Pre$ as well as positive tokens (without restriction). Note that these notations fully agree with the one introduced in definition 2.

**Definition 3.** *Define the transition rules for the $\mathcal{L}_{MR}$ family of languages as the general rules of figure 2 together with rule (CM) of that figure.*

Rule (CM) states that a multi-set rewriting $(pre, post)$ can be executed in a store $\sigma$ if the multi-set $pre^+$ is included in $\sigma$ and if no negative pre-condition occurs in $\sigma$. If such conditions hold then the execution of the rewriting deletes from $\sigma$ all the negative post-conditions, and adds to $\sigma$ all the positive post-conditions.

### 2.4 $\mathcal{L}_{CS}$: Communication transactions

A natural further refinement is to impose an order on the test of pre-conditions and the evaluation of post-conditions, possibly mixing pre- and post-conditions. We are thus lead to sequences of elementary actions, which we will take, for clarity purposes, in the Linda form instead of the $+t$ and $-t$ of the $\mathcal{L}_{MR}$ family. These sequences will be called *communication transactions,* with the intuition that they are to be executed as a single "all-or-nothing" transaction. They have been employed in Shared-Prolog ([6]) and in PoliS [12]).

**Definition 4.** *Define the set of communication transactions Stcom as the set of C's engendered by the $\mathcal{L}_{CS}$ rules of figure 1. Moreover, for any subset $\mathcal{X}$ of Stcom, define the language $\mathcal{L}_L(\mathcal{X})$ as the set of agents A generated by the general rule of figure 1. The transition rules for these agents are the general ones of figure 2 together with rule (CS).*

## 3 Operational semantics

### 3.1 Observables

**Definition 5.**

1. *Let Stoken be a denumerable set, the elements of which are subsequently called* tokens *and are typically represented by the letters t and u. Define the set of stores Sstore as the set of finite multisets with elements from Stoken.*
2. *Let $\delta^+$ and $\delta^-$ be two fresh symbols denoting respectively success and failure. Define the set of histories Shist as the set $Sstore \times \{\delta^+, \delta^-\}$.*
3. *Define the* operational semantics $\mathcal{O} : Sagent \to \mathcal{P}(Shist)$ *as the following function: For any agent A,*

$$\begin{aligned}\mathcal{O}(A) = \; & \{(\sigma, \delta^+) : \langle A \mid \emptyset\rangle \to^* \langle E \mid \sigma\rangle\} \\ & \cup \\ & \{(\sigma, \delta^-) : \langle A \mid \emptyset\rangle \to^* \langle B \mid \sigma\rangle \not\to, B \neq E\}\end{aligned}$$

### 3.2 Normal form

A classical result of concurrency theory is that modelling parallel composition by interleaving, as we do, allows agents to be considered in a normal form. We first define what this actually means, and then state the proposition that agents and their normal forms are equivalent in the sense that they yield the same computations.

**Definition 6.** *Given a subset $\mathcal{X}$ of Slcom, Smcom, or Stcom, the set Snagent of agents in normal form is defined by the following rule, where N is an agent in normal form and c denotes a communication action of $\mathcal{X}$:*

$$N ::= c \mid c\,;\; N \mid N \;+\; N.$$

**Proposition 1.** *For any agent A, there is an agent N in normal form which has the same derivation sequences as A.*

## 4 Modular embedding

A natural way to compare the expressive power of two languages is to see whether all programs written in one language can be "easily" and "equivalently" translated into the other language, where equivalent is intended in the sense of the same observable behaviour.

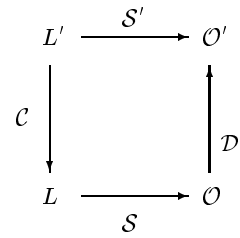


**Fig. 3.** Basic embedding.

The basic definition of embedding, given by Shapiro [25] is the following. Consider two languages $L$ and $L'$. Assume given the semantics mappings (*observation criteria*) $\mathcal{S} : L \to \mathcal{O}$ and $\mathcal{S}' : L' \to \mathcal{O}'$, where $\mathcal{O}$ and $\mathcal{O}'$ are some suitable domains. Then $L$ can *embed* $L'$ if there exists a mapping $\mathcal{C}$ (*coder*) from the statements of $L'$ to the statements of $L$, and a mapping $\mathcal{D}$ (*decoder*) from $\mathcal{O}$ to $\mathcal{O}'$, such that the diagram of figure 3 commutes, namely such that for every statement $A \in L'$: $\mathcal{D}(\mathcal{S}(\mathcal{C}(A))) = \mathcal{S}'(A)$.

The basic notion of embedding is too weak since, for instance, the above equation is satisfied by any pair of Turing-complete languages. De Boer and Palamidessi hence proposed in [15] to add three constraints on the coder $\mathcal{C}$ and on the decoder $\mathcal{D}$ in order to obtain a notion of *modular* embedding usable for concurrent languages:

1. $\mathcal{D}$ should be defined in an element-wise way w.r.t. $\mathcal{O}$:

$$\forall X \in \mathcal{O} : \ \mathcal{D}(X) = \{\mathcal{D}_{el}(x) \mid x \in X\} \qquad (P_1)$$

   for some appropriate mapping $\mathcal{D}_{el}$;
2. the compiler $\mathcal{C}$ should be defined in a compositional way w.r.t. the sequential, parallel and choice operators (actually, this is only required for the parallel and choice operators in [15]):

$$\begin{aligned} \mathcal{C}(A \ ; \ B) &= \mathcal{C}(A) \ ; \ \mathcal{C}(B) \\ \mathcal{C}(A \mid\mid B) &= \mathcal{C}(A) \mid\mid \mathcal{C}(B) \\ \mathcal{C}(A \ + \ B) &= \mathcal{C}(A) \ + \ \mathcal{C}(B) \end{aligned} \qquad (P_2)$$

3. the embedding should preserve the behaviour of the original processes w.r.t. deadlock, failure and success (*termination invariance*):

$$\forall X \in \mathcal{O}, \forall x \in X : \ tm'(\mathcal{D}_{el}(x)) = tm(x) \qquad (P_3)$$

   where $tm$ and $tm'$ extract the information on termination from the observables of $L$ and $L'$, respectively.

An embedding is then called *modular* if it satisfies properties $P_1$, $P_2$, and $P_3$.

The existence of a modular embedding from $L'$ into $L$ will be denoted by $L' \leq L$. It is easy to see that $\leq$ is a pre-order relation. Moreover if $L' \subseteq L$ then

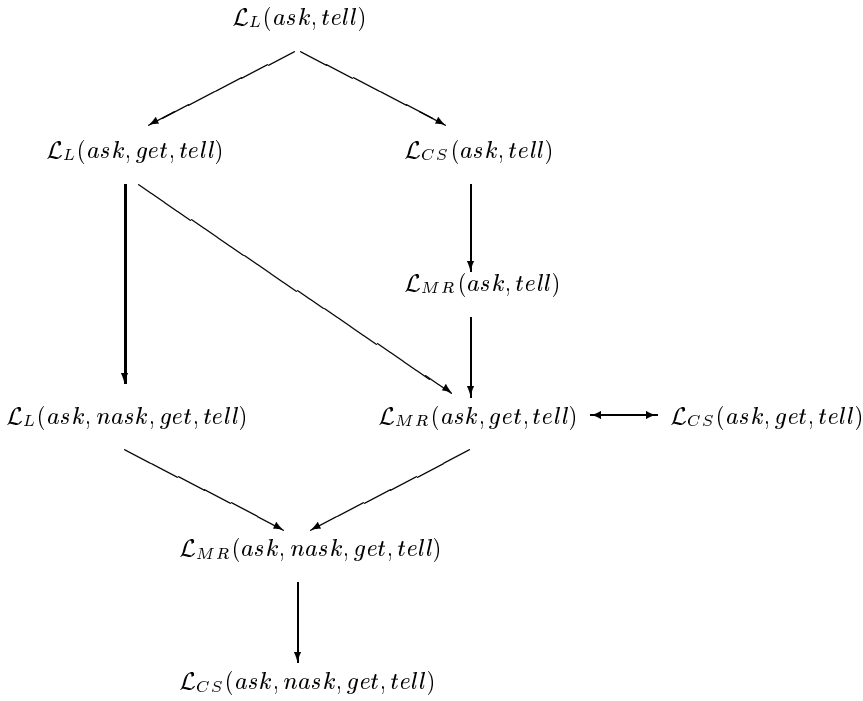


**Fig. 4.** Main results of the comparisons.

$L' \leq L$ that is, any language embeds all its sublanguages. This property descends immediately from the definition of embedding, by setting $\mathcal{C}$ and $\mathcal{D}$ equal to the identity function.

## 5 Comparisons

We now turn to an exhaustive comparison of the relative expressive power of the languages introduced in section 2.

We will consider nine different languages which are obtained by considering three different sets of communication primitives, namely $\mathcal{X} = \{ask, tell\}$, $\mathcal{X} = \{ask, get, tell\}$, and $\mathcal{X} = \{ask, nask, get, tell\}$, for each of the three parameterised languages $\mathcal{L}_L(\mathcal{X})$, $\mathcal{L}_{CS}(\mathcal{X})$, and $\mathcal{L}_{MR}(\mathcal{X})$.

The whole set of separation and equivalence results is summarised in figure 4, where an arrow from a language $\mathcal{L}_1$ to a language $\mathcal{L}_2$ means that $\mathcal{L}_2$ embeds $\mathcal{L}_1$, that is $\mathcal{L}_1 \leq \mathcal{L}_2$. Notice that, thanks to the transitivity of embedding, the figure contains only a minimal amount of arrows. However, apart from these induced relations, no other relation holds. In particular, when there is one arrow from $\mathcal{L}_1$ to $\mathcal{L}_2$ but there is no arrow from $\mathcal{L}_2$ to $\mathcal{L}_1$, then $\mathcal{L}_1$ is strictly less expressive than $\mathcal{L}_2$, that is $\mathcal{L}_1 < \mathcal{L}_2$.

The separation and equivalence results are presented in two steps. Section 5.1 first presents the intuition for these results whereas their formal statements are given in section 5.2.

### 5.1 Intuitive analysis of the results

Before presenting the proofs of the results illustrated in figure 4, we will try to analyse their intuitive meaning. More precisely, we shall try here to show informally how such formal separation results confirm the intuitive expectations form a programming viewpoint. Of course the intuitive explanation of a separation (or equivalence) result does not formally prove the validity of the result itself. One may indeed argue that even if there is no obvious encoding between the two languages, there may well be a non-trivial encoding that may yield the embedding. The non existence of such embeddings will be formally established by the propositions reported in section 5.2.

*Analysis for* $\mathcal{X} = \{ask, tell\}$
Let us first consider the case in which $\mathcal{X} = \{ask, tell\}$. It is easy to see that $\mathcal{L}_L(ask, tell)$ does not support a straightforward way of *atomically* testing the simultaneous presence of two resources $a$ and $b$ in the dataspace. Indeed the obvious coding $(ask(a); ask(b))$ will not be executed atomically and may not produce the desired behaviour for instance in: $(ask(a); ask(b); P) + (ask(a); ask(c); Q)$. The language $\mathcal{L}_{CS}(ask, tell)$ instead supports a straightforward way of atomically testing the presence of two resources in the dataspace, via the communication transaction $[ask(a); ask(b)]$, thus intuitively confirming the separation result $\mathcal{L}_L(ask, tell) < \mathcal{L}_{CS}(ask, tell)$.

It is easy to observe that the same kind of test can be naturally expressed also in $\mathcal{L}_{MR}(ask, tell)$ via the rewriting $(\{+a, +b\}, \{...\})$. Moreover the language $\mathcal{L}_{MR}(ask, tell)$ permits to express also tests of the form "if there are at least $n$ copies of a resource $a$ then". For instance the rewriting $(\{+a, +a\}, \{+b\})$ states that if there are at least two copies of resource $a$ then resource $b$ will be added to the dataspace. The same test cannot be easily expressed in $\mathcal{L}_{CS}(ask, tell)$ with $[ask(a); ask(a); tell(b)]$, since the two $ask$ operations may match the same instance of $a$ in the dataspace. The inability of $\mathcal{L}_{CS}(ask, tell)$ to atomically testing the presence of multiple copies of the same resource confirms intuitively the separation result $\mathcal{L}_{CS}(ask, tell) < \mathcal{L}_{MR}(ask, tell)$.

*Analysis for* $\mathcal{X} = \{ask, get, tell\}$
The addition of the *get* primitive to the set $\mathcal{X}$ gives to each of the former three languages the ability of deleting tuples, hence yielding a non-monotonic evolution of the dataspace. The three separation results $\mathcal{L}_L(ask, tell) < \mathcal{L}_L(ask, get, tell)$, $\mathcal{L}_{MR}(ask, tell) < \mathcal{L}_{MR}(ask, get, tell)$, and $\mathcal{L}_{CS}(ask, tell) < \mathcal{L}_{CS}(ask, get, tell)$ follow such intuition.

The separation result between the basic Linda calculus and the multi-set rewriting calculus continues to hold also after introducing the *get* operation, that is, $\mathcal{L}_L(ask, get, tell) < \mathcal{L}_{MR}(ask, get, tell)$. Indeed the addition of *get* does

not still allow $\mathcal{L}_L(ask, get, tell)$ to atomically test the simultaneous presence of two resources $a$ and $b$ in the dataspace.

On the other hand, the introduction of *get* removes the gap between communication sequences and multi-set rewriting, which have in this case the same expressive power, that is, $\mathcal{L}_{MR}(ask, get, tell) = \mathcal{L}_{CS}(ask, get, tell)$. For instance $\mathcal{L}_{CS}(ask, get, tell)$ can now express tests of the form "if there are at least two copies of resource $a$ then" via the transaction $[get(a); ask(a); tell(a)]$.

*Analysis for* $\mathcal{X} = \{ask, nask, get, tell\}$
The introduction of the *nask* primitive into the set $\mathcal{X}$ gives to each language the ability of testing the *absence* of data from the dataspace, and hence to express *if-then-else* conditions of the form "if resource $a$ belongs to the dataspace then do $P$ else do $Q$". For instance such test can be expressed in $\mathcal{L}_L(ask, nask, get, tell)$ as $(ask(a); P) + (nask(a); Q)$. The additional expressive power given by *nask* intuitively explains the separations result $\mathcal{L}(ask, get, tell) < \mathcal{L}(ask, nask, get, tell)$, which holds for $\mathcal{L}$ being either $\mathcal{L}_L$, $\mathcal{L}_{MR}$ or $\mathcal{L}_{CS}$.

Even after introducing *nask*, the basic Linda calculus is less expressive of both communication transactions and multi-set rewriting. Indeed $\mathcal{L}_L(ask, nask, get, tell)$ is still not able to atomically test the simultaneous presence of two resources $a$ and $b$ in the dataspace.

The introduction of negative tests instead reverses the relation between $\mathcal{L}_{MR}(ask, nask, get, tell)$ and $\mathcal{L}_{CS}(ask, nask, get, tell)$. Indeed the availability of *nask* allows $\mathcal{L}_{CS}(ask, nask, get, tell)$ to "count" the number of copies of a resource available in the dataspace. For instance $\mathcal{L}_{CS}(ask, nask, get, tell)$ can express tests of the form "if there are *exactly* two copies of a resource $a$ then do $P$" via the communication sequence $[get(a); get(a); nask(a); tell(a); tell(a)]; P$ while $\mathcal{L}_{MR}(ask, nask, get, tell)$ can only express test of the form "if there *at least* $n$ copies of resource then". This intuitively explains the last separation result $\mathcal{L}_{MR}(ask, nask, get, tell) < \mathcal{L}_{CS}(ask, nask, get, tell)$.

### 5.2 Formal results

We now formally state the results of the language comparisons that induce the hierarchy reported in figure 4. To illustrate the proof techniques used we sketch the proof of the first proposition. The other proofs are not included for the lack of space. We first establish some basic relations.

**Proposition 2.**

*i)* $\mathcal{L}_L(\mathcal{X}) \leq \mathcal{L}_{MR}(\mathcal{X})$ *for any set of communication primitives* $\mathcal{X}$
*ii)* $\mathcal{L}_{MR}(tell, ask, get) \leq \mathcal{L}_{CS}(tell, ask, get)$
*iii)* $\mathcal{L}_{MR}(tell, ask, nask, get) \leq \mathcal{L}_{CS}(tell, ask, nask, get)$
*iv)* $\mathcal{L}_{MR}(tell, ask) \not\leq \mathcal{L}_{CS}(tell, ask)$

*Proof.* i) Immediate by defining the coder as follows:

$$\mathcal{C}(tell(t)) = (\{\}, \{+t\}) \qquad \mathcal{C}(get(t)) = (\{+t\}, \{-t\})$$
$$\mathcal{C}(ask(t)) = (\{+t\}, \{\}) \qquad \mathcal{C}(nask(t)) = (\{-t\}, \{\})$$

ii) and iii) Indeed, the non-redundancy of multiple ask queries in $\mathcal{L}_{MR}$ can be reconstructed in $\mathcal{L}_{CS}$ by first getting the tokens and then telling them back. It is thus sufficient to code
$(\{+g_1, ..., +g_p, +a_1, ..., +a_q, -n_1, ..., -n_r, \}, \{+t_1, ..., +t_s, -g_1, ..., -g_p\})$ into

$$[get(g_1), \cdots, get(g_p), get(a_1), \cdots, get(a_q), tell(a_1), \cdots, tell(a_q),$$
$$nask(n_1), \cdots, nask(n_r), tell(t_1), \cdots, tell(t_s)]$$

iv) By contradiction, assume that there is a coder $\mathcal{C}$. Obviously, for any token $t$, the computation of $(\{\}, \{+t\})$ succeeds and so should that of $\mathcal{C}((\{\}, \{+t\}))$ by $P_3$. Let us call $\sigma$ the state resulting from one computation. As $\mathcal{L}_{CS}(tell, ask)$ contains no destructive operations and no negative tests, $\mathcal{C}((\{\}, \{+t\}))$ ; $\mathcal{C}((\{\}, \{+t\}))$ has a successful computation resulting in the store $\sigma \cup \sigma$. Now consider $\mathcal{C}((\{+t, +t\}, \{\}))$ in its normal form: $(a_1 \ ; \ A_1 + \cdots + a_m \ ; \ A_m)$. Since $(\{\}, \{+t\})$ ; $(\{\}, \{+t\})$ ; $(\{+t, +t\}, \{\})$ succeeds, by $(P_3)$, there should exist $i \in [1, m]$ such that $\langle \mathcal{C}((\{+t, +t\}, \{\})) \mid \sigma \cup \sigma \rangle \longrightarrow \langle A_i \mid \sigma \cup \sigma \cup \tau \rangle$, for some store $\tau$, Moreover $A_i$ computed from $\sigma \cup \sigma \cup \tau$ should only lead to success and thus, as $\mathcal{L}_{CS}(tell, ask)$ does not contain any destructive operation, $A_i$ started on $\sigma \cup \tau$ has only successful computations. It follows that $\langle \mathcal{C}((\{\}, \{+t\}))$ ; $\mathcal{C}((\{+t, +t\}, \{\})) \mid \emptyset \rangle \longrightarrow^* \langle \mathcal{C}((\{+t, +t\}, \{\})) \mid \sigma \rangle \longrightarrow \langle M_i \mid \sigma \cup \tau \rangle$ is a valid computation prefix for $\mathcal{C}((\{\}, \{+t\})$ ; $(\{+t, +t\}, \{\}))$ which can only be continued by successful computations. This contradicts by $P_3$ the fact that $(\{\}, \{+t\})$ ; $(\{+t, +t\}, \{\})$ has only one failing computation.

It is worth noting that while the $\mathcal{L}_L$ family of languages can be all embedded in the corresponding languages of the $\mathcal{L}_{MR}$ family, the converse does not hold.

**Proposition 3.**

*i)* $\mathcal{L}_{MR}(tell, ask) \not\leq \mathcal{L}_L(tell, ask)$
*ii)* $\mathcal{L}_{MR}(tell, ask, get) \not\leq \mathcal{L}_L(tell, ask, get)$
*iii)* $\mathcal{L}_{MR}(ask, nask, get, tell) \not\leq \mathcal{L}_L(ask, nask, get, tell)$

We now analyse some embeddings of the $\mathcal{L}_{CS}$ family in the $\mathcal{L}_{MR}$ family.

**Proposition 4.**

*i)* $\mathcal{L}_{CS}(tell, ask) \leq \mathcal{L}_{MR}(tell, ask)$
*ii)* $\mathcal{L}_{CS}(tell, ask, get) \leq \mathcal{L}_{MR}(tell, ask, get)$
*iii)* $\mathcal{L}_{CS}(tell, ask, get, nask) \not\leq \mathcal{L}_{MR}(tell, ask, get, nask)$

The separation results between members of the family $\mathcal{L}_L$ [7] extend to the members of $\mathcal{L}_{MR}$ and $\mathcal{L}_{CS}$, as shown by the following proposition.

**Proposition 5.**

*i)* $\mathcal{L}_L(ask, tell) < \mathcal{L}_L(ask, get, tell) < \mathcal{L}_L(ask, nask, get, tell)$
*ii)* $\mathcal{L}_{MR}(ask, tell) < \mathcal{L}_{MR}(ask, get, tell) < \mathcal{L}_{MR}(ask, nask, get, tell)$
*iii)* $\mathcal{L}_{CS}(ask, tell) < \mathcal{L}_{CS}(ask, get, tell) < \mathcal{L}_{CS}(ask, nask, get, tell)$

The following proposition completes the comparisons.

**Proposition 6.**

*i)* $\mathcal{L}_L(ask, tell) < \mathcal{L}_{CS}(ask, tell)$
*ii)* $\mathcal{L}_{CS}(ask, tell)$ *and* $\mathcal{L}_L(ask, get, tell)$ *are incomparable*
*iii)* $\mathcal{L}_{CS}(ask, tell)$ *and* $\mathcal{L}_L(ask, get, nask, tell)$ *are incomparable*
*iv)* $\mathcal{L}_{MR}(ask, tell)$ *and* $\mathcal{L}_L(ask, get, tell)$ *are incomparable*
*v)* $\mathcal{L}_{MR}(ask, tell)$ *and* $\mathcal{L}_L(ask, nask, get, tell)$ *are incomparable*
*vi)* $\mathcal{L}_{MR}(ask, get, tell)$ *and* $\mathcal{L}_L(ask, nask, get, tell)$ *are incomparable*

## 6 Concluding remarks

We have compared the expressive power of three families of coordination models based on shared dataspaces. The first class $\mathcal{L}_L$ relies on Linda's communication primitives, the second class $\mathcal{L}_{MR}$ relies on the more general notion of multiset rewriting, while the third class $\mathcal{L}_{CS}$ features communication transactions that consist of sequences of Linda-like operations to be executed atomically. For each family we have considered three different languages that differ from one another in the set $\mathcal{X}$ of communication primitives used, for $\mathcal{X}$ equal respectively to $\{ask, tell\}$, $\{ask, get, tell\}$ and $\{ask, nask, get, tell\}$.

It is worth mentioning that we have exploited the main proof techniques reported in this paper to perform a wider comparison of the languages, by considering also other sets $\mathcal{X}$ of primitives with $\mathcal{X} \subseteq \{ask, nask, get, tell\}$. Because of lack of space, we reported here only the main comparisons. It is also worth noting that the results presented extend directly to agents containing recursion.

As pointed out in the Introduction, the families $\mathcal{L}_L$, $\mathcal{L}_{MR}$, and $\mathcal{L}_{CS}$ are representative of a substantial amount of coordination languages. We believe that the comparison of the expressive power of different classes of coordination models provides useful insights for both the theory and the practice of coordination-based approaches. The resulting hierarchy highlights the equivalence of different models and indicates which extensions may be worth considering because of their additional expressive power.

## 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. J.M. Andreoli and R. Pareschi. Linear Objects: logical processes with builtin inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.
3. J. Banatre and D. LeMetayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1991.
4. E. Best, F.S. de Boer and C. Palamidessi. Partial Order and SOS Semantics for Linear Constraint Programs. In [18].
5. M. Bonsangue, J. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. To appear in proc. of SAC'99.

6. A. Brogi and P. Ciancarini. The Concurrent Language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, January 1991.
7. A. Brogi and J.-M. Jacquet. On the Expressiveness of Linda-like Concurrent Languages. *Electronic Notes in Theoretical Computer Science*, 1998.
8. N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing Equivalence of Linda Coordination Primitives. *Electronic Notes in Theoretical Computer Science*, 7, 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. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.
11. N. Carriero, D. Gelernter and L. Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz and A. Yonezawa (editors) *Object based models and languages for concurrent systems*, LNCS 924, pages 66–76, Springer-Verlag, 1994.
12. P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12(3):251–284, 1994.
13. P. Ciancarini and C. Hankin (editors). Coordination'96: First International Conference on Coordination Models and Languages. LNCS 1061. Springer-Verlag, 1996.
14. P. Ciancarini and D. Rossi. Jada: coordination and communication for Java agents. In *Second Int. Workshop on mobile object systems*, LNCS 1222, pages 213–228, Springer-Verlag, 1996.
15. F.S. de Boer and C. Palamidessi. Embedding as a Tool for Language Comparison. *Information and Computation*, 108(1):128–157, 1994.
16. R. De Nicola, G. Ferrari and R. Pugliese. KLAIM: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 1998.
17. M. Felleisen. On the Expressive Power of Programming Languages. In N. Jones, editor, *Proc. ESOP'90*, LNCS 432, pages 134–151. Springer-Verlag, 1990.
18. D. Garlan and D. Le Metayer (editors). Coordination'97: Second International Conference on Coordination Models and Languages. LNCS. Springer-Verlag, 1997.
19. D. Gelernter and N. Carriero. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, 1992.
20. J.M. Jacquet and K. De Bosschere. On the Semantics of muLog. *Future Generation Computer Systems Journal*:10, pages 93–135, Elsevier, 1994.
21. J.-M. Jacquet and L. Monteiro. Towards Resource Handling in Logic Programming: the PPL Framework and its Semantics. *Computer Languages*, 22(2/3):51–77, 1996.
22. G.A. Papadopolous and F. Arbab. Coordination models and languages. *Advances in Computers*, 48, Academic-Press, 1998.
23. A. Rowstron and A. Wood. BONITA: A set of tuple space primitives for distributed coordination. In *30th Hawaii Int. Conf. on System Sciences*, IEEE Press, Vol. 1, pages 379–388, 1997.
24. V.A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.
25. E.Y. Shapiro. Embeddings among Concurrent Programming Languages. In W.R. Cleaveland, editor, *Proc. of CONCUR'92*, pages 486–503. Springer-Verlag, 1992.
26. R. Tolksdorf. Coordinating services in open distributed systems with LAURA. In [13], pages 386–402.
27. P. Wegner. Why Interaction Is More Powerful Than Algorithms. *Communications of the ACM*, 1997.
28. G. Zavattaro. On the incomparability of Gamma and Linda. Technical report SEN-R9827, Department of Software Engineering, CWI, 1998.
29. G. Zavattaro. Towards a hierarchy of negative test operators for generative communication. *Electronic Notes in Theoretical Computer Science*, 16(2):83–100, 1998.