On Distributed Density in Tuple-based Coordination Languages

Denis Darquennes Jean-Marie Jacquet
Faculty of Computer Science
University of Namur
Namur, Belgium
denis.darquennes@unamur.be jean-marie.jacquet@unamur.be

Isabelle Linden
Department of Business Administration
Univeristy of Namur
Namur, Belgium
isabelle.linden@unamur.be

Inspired by the chemical metaphor, this paper proposes an extension of Linda-like languages in the aim of modeling the coordination of complex distributed systems. The new language manipulates finite sets of tuples and distributes a density among them. This new concept adds to the non-determinism inherent in the selection of matched tuples a non-determinism to the tell, ask and get primitives on the consideration of different tuples. Furthermore, thanks to de Boer and Palamidessi's notion of modular embedding, we establish that this new language strictly increases the expressiveness of the Dense Bach language introduced earlier and, consequently, Linda-like languages.

1 Introduction

The technological evolutions over the last recent years confirm the upward trends in a pervading of our everyday environments by new mobile devices injecting or retrieving information from very dynamic and dense networks. In order to garantee robustness and continuity of the services they propose, the global structure must be tolerant regarding any modification in topology, device technology or creation of new services. These constraints will be fulfilled if full self-organisation is incorporated as an inherent property of the coordination models. Coordination languages based on tuple spaces offer an elegant response to such constraints. The Bach language - a dialect of Linda developed at the University of Namur - is one of them, and permits to model in an elegant way the interaction between different components through the deposit and retrieval of tuples in a shared space. As its basic form only allows the manipulation of one tuple at a time and since the selection between several tuples matching a required one is provided in an non-deterministic fashion, a first extension was first proposed in [12] in the aim of enriching traditional data-based coordination languages by a notion of density attached to tuples, thereby yielding a new coordination language, called Dense Bach.

To illustrate the use of the dense tuples, let us consider the context of service oriented computing. Let us imagine a situation where a group of n researchers planing their presence to a conference want to book rooms in a hotel. Their query could naturally consist in getting n rooms in the tuple space of free rooms in any hotel, and to effectivelly book them only if all the n rooms are available, or if the number of rooms combined with their number of beds meets the required n amount. In a more scientific context, we could consider the chemical reactions between three elements N (nitrogen), O (oxygen) and S (sulphur) present in a reactor. Following the distribution - understood here in the chemical sense - of the global density

Javier Cámara and José Proença (Eds.):
13th International Workshop on Foundations of
Coordination Languages and Self-Adaptive Systems (FOCLASA'14)
EPTCS 175, 2015, pp. 36–53, doi:10.4204/EPTCS.175.3

between those three different reactants, some reactions will be more facilitated and others minder, and the concentration of the resulting products could alternate more in favour of one solution (like NO_2) or another (like SO_2).

Considering those two previous examples, we then propose in this paper as a next natural step to consider a set of tuples, among which the density is distributed. The new abstract language resulting from that extension strictly increases the expressiveness of Linda-like languages. It is still built upon the four primitives tell, ask, get and nask, accessing a tuplespace, also named subsequently store. However, it enhances it with a non-deterministic behavior of the tell, ask and get primitives with regard to which tuples from the set are considered.

Our purposes remain of a theoretical nature and, hence, for simplicity purposes, we shall consider in this paper a simplified version where tuples are taken in their simplest form of flat and unstructured tokens. Nevertheless, the resulting simplification of the matching process is orthogonal to our purposes and, consequently, our results can be directly extended to more general tuples.

This paper fits into the continuity of previous work done by the authors, among others of [5, 8, 9, 12, 16]. As a result, our approach follows the same lines of research, and employs de boer and Palamidessi's modular embedding to test the expressiveness of languages. The rest of this paper is consequently organized as follows. Section 2 presents our extension of the Dense Bach language, called Dense Bach with Distributed Density and, after the definition of the distribution of density on a list of tokens, defines an operational semantics. Section 3 provides a short presentation of modular embedding and, on that basis, proceeds with an exhaustive comparison of the relative expressive power of the languages Dense Bach and Dense Bach with Distributed Density. Finally, section 4 compares our work with related work, draws our conclusions and presents expectations for future work.

2 Densed Tuple-based Coordination Languages

This section exposes in four points the different densed tuple-based coordination languages, firstly by presenting their primitives, and secondly the different languages. The two last points present an operational semantic based on transition systems.

2.1 Primitives

We start by defining the Bach and Dense Bach languages [12] from which the language under study in this paper is an extension.

2.1.1 Bach and Dense Bach

The following definition formalizes how we attach a density to them.

Definition 1. *Let* Stoken *be a enumerable set, the elements of which are subsequently called tokens and are typically represented by the letters t and u. Define the association of a token t and a positive integer $n \in \mathbb{N}$ as a dense token. Such an association is typically denoted as $t(n)$. Define then the set of dense tokens as the set SDtoken. Note that since Stoken and $\mathbb{N}$ are both enumerable, the set SDtoken is also enumerable.*

Intuitively, a dense token $t(m)$ represents the simultaneous presence of m occurrences of t. As a result, $\{t(m)\}$ is subsequently used to represent the multiset $\{t, \cdots, t\}$ composed of these m occurrences. Moreover, given two multisets of tokens σ and τ, we shall use $\sigma \cup \tau$ to denote the multiset union of

$$(\mathbf{T}) \qquad \langle\, tell(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t\} \,\rangle$$

$$(\mathbf{A}) \quad \langle\, ask(t) \mid \sigma \cup \{t\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t\} \,\rangle$$

$$(\mathbf{G}) \qquad \langle\, get(t) \mid \sigma \cup \{t\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle$$

$$(\mathbf{N}) \qquad \frac{t \notin \sigma}{\langle\, nask(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 1: Transition rules for token-based primitives (Bach)

elements of σ and τ. As a particular case, by slightly abusing the syntax in writing $\{t(m),t(n)\}$, we have $\{t(m)\} \cup \{t(n)\} = \{t(m),t(n)\} = \{t(m+n)\}$. Finally, we shall use $\sigma \uplus \{t(m)\}$ to denote, on the one hand, the multiset union of σ and $\{t(m)\}$, and, on the other hand, the fact that t does not belong to σ.

Definition 2. *Define the set $\mathscr{T}$ of the token-based primitives as the set of primitives T generated by the following grammar:*

$$T \quad ::= \quad tell(t) \mid ask(t) \mid get(t) \mid nask(t)$$

where t represents a token.

Definition 3. *Define the set of dense token-based primitives $\mathscr{T}_d$ as the set of primitives T_d generated by the following grammar:*

$$T_d \quad ::= \quad tell(t(m)) \mid ask(t(m)) \mid get(t(m)) \mid nask(t(m))$$

where t represents a token and m a positive natural number.

The primitives of the Bach language are essentially the Linda ones rephrased in a constraint-like setting. As a result, by calling *store* a multiset of tokens aiming at representing the current content of the tuple space, the execution of the $tell(t)$ primitives amounts to enrich the store by an occurrence of t. The $ask(t)$ and $get(t)$ primitives check whether t is present on the store with the latter removing one occurrence. Dually, $nask(t)$ tests whether t is absent from the store.

The primitives of the dense Bach language extend these primitives by simultaneously handling multiple occurrences. Accordingly, $tell(t(m))$ atomically puts m occurrences of t on the store and $ask(t(m))$ together with $get(t(m))$ require the presence of at least m occurrences of t with the latter removing m of them. Moreover, $nask(t(m))$ verifies that there are less than m occurrences of t.

These executions can be formalized by the transition steps of figures 1 and 2, where configurations are pairs of instructions, for the moment reduced to simple primitives, coupled to the contents of a store. Note that E is used to denote a terminated computation. As can be seen by the above description, the primitives of Bach are those of Dense Bach with a density of 1. Consequently, our explanation starts by the more general rules of figure 2. Rule (T_d) states that for any store σ and any token t with density m, the effect of the tell primitive is to enrich the current set of tokens by m occurrences of token t. Note that $\cup$ denotes multi-set union. Rules (A_d) and (G_d) specify the effect of ask and get primitives, both requiring the presence of at least m occurrences of t, but the latter also consuming them. Rule (N_d) defines the nask primitive, which tests for the absence of m occurrences of t. Note that there might be some provided there are less than m. It is also worth observing that thanks to the notation $\sigma \uplus \{t(n)\}$ one

$$(\mathbf{T_d}) \quad \frac{m \in \mathbb{N}_0}{\langle\, tell(t(m)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t(m)\} \,\rangle}$$

$$(\mathbf{A_d}) \quad \frac{m \in \mathbb{N}_0}{\langle\, ask(t(m)) \mid \sigma \cup \{t(m)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t(m)\} \,\rangle}$$

$$(\mathbf{G_d}) \quad \frac{m \in \mathbb{N}_0}{\langle\, get(t(m)) \mid \sigma \cup \{t(m)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$(\mathbf{N_d}) \quad \frac{n < m}{\langle\, nask(t(m)) \mid \sigma \uplus \{t(n)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \uplus \{t(n)\} \,\rangle}$$

Figure 2: Transition rules for dense token-based primitives (Dense Bach)

is sure that t does not occur in σ and consequently that there are exactly n occurrences of t. This does not apply for rules (A_d) and (G_d) for which it is sufficient to assume the presence of at least m occurrences, allowing σ to contain others.

Figure 1 specifies the transition rules for the primitives of the Bach language. As expected, they amount to the rules of Figure 2 where the density m is taken to be 1 and the union symbol is interpreted on multi-sets.

2.1.2 Dense Bach with distributed density

A natural extension is to replace a token by a set of tokens and to distribute the density requirements on tokens. For instance, the primitive $ask([t,u,v](6))$ succeeds on a store containing one occurrence of t, two of u and three of v. Dually, the computation of $tell([t,u,v](6))$ may result in adding two occurrences of t on the store, three of u and one of v. The following definitions formalize this intuition.

Definition 4. *Let Snlt denote the set of non-empty sets of tokens in which, for simplicity purposes, each token differs from the others. Such a set is typically denoted as $L = [t_1, \ldots, t_p]$ and is thus such that $t_i \neq t_j$ for $i \neq j$. Define a dense set of tokens as a set of Snlt associated with a positive integer. Such a dense set is typically represented as $L(m)$, with L the set of tokens and m an integer.*

The distribution of the density over a set of tokens is formalized through the following distribution function.

Definition 5. *Define the distribution of tokens from dense sets of tokens to sets of tuples of dense tokens as follows:*

$$\mathscr{D}([t_1, \cdots, t_p](m)) = \{(t_1(m_1), \cdots, t_p(m_p)) : m_1 + \cdots + m_p = m\}$$

Note that, thanks to the definition of dense tokens, we assume above that the m_i's are positive integers. For the sake of simplicity, we shall call the set $\mathscr{D}([t_1, \cdots, t_p](m))$ the distribution of m over $[t_1, \cdots, t_p]$.

The distribution of an integer m over a set of tokens L has the potential to express the behavior of the extended primitives. Indeed, telling a dense set amounts to telling atomically the $t_i(m_i)$'s of a tuple defined above. Asking or getting a dense set requires to check that a tuple of $\mathscr{D}([t_1, \cdots, t_p](m))$ is present on the considered store. For the negative ask, the requirement is that none of the tuple is present. For the ease of writing and to make this latter concept clear, we introduce the following concept of intersection.

Definition 6. *Let m be a positive integer, $L = [t_1, \cdots, t_p]$ be a set of tokens and σ a store. We define $\mathscr{D}(L(m)) \sqcap \sigma$ as the following set of tuples of dense tokens :*

$$\mathscr{D}(L(m)) \sqcap \sigma = \{(t_1(m_1), \ldots, t_p(m_p)) \in \mathscr{D}(L(m)) : \forall i \{t_i(m_i)\} \subseteq \sigma\}$$

From an implementation point of view, it is worth observing that one may give a syntactical characterization of the emptyness of such an intersection.

Definition 7. *Given a store σ and a dense set $L(m)$, with $L = [t_1, \cdots, t_p]$, we denote by $Max(\sigma, L(m))$ the tuple $(t_1(m_1), \ldots, t_p(m_p))$ where the m_i's denote the number of occurrences of t_i in σ. Moreover, we denote by $SMax(\sigma, L(m))$ the sum $m_1 + \cdots + m_p$*

It is easy to establish the following proposition.

Theorem 1. *For any dense set of tokens $L(m)$ and any store σ, one has $\mathscr{D}(L(m)) \sqcap \sigma = \emptyset$ iff $SMax(\sigma, L(m)) < m$.*

Proof. Simple verification. □

We are now in a position to specify the language extension handling dense sets of tokens.

Definition 8. *Define the set of dense sets primitives $\mathscr{T}_{dbd}$ as the set of primitives T_{dbd} generated by the following grammar:*

$$T_{dbd} \quad ::= \quad tell(L(m)) \mid ask(L(m)) \mid get(L(m)) \mid nask(L(m))$$

where $L(m)$ represents a dense set of tokens.

The transition steps for these primitives are defined in figure 3. As suggested above, rule (T_{dbd}) specifies that telling a dense set $L(m)$ of tokens amounts to atomically add the multiple occurrences $t_i(m_i)$'s of the tokens of a tuple of the distribution of m over L. Note that the selected tuple is chosen non-deterministically, which gives to a tell primitive a non-deterministic behavior as opposed to the tell primitives of Bach and Dense Bach. Rule (A_{dbd}) states that asking for the dense set $L(m)$ amounts to testing that a tuple of the distribution of m over L is in the store, which is technically stated through the non-emptyness of the intersection of the distribution and the store. Rule (G_{dbd}) requires that the tokens of the tuples are removed in the considered multiplicity. Finally, rule (N_{dbd}) specifies that negatively asking $L(m)$ succeeds if m is strictly positive and no tuple of the distribution of m over L is present on the current store.

2.2 Languages

We are now in a position to define the languages we shall consider. The statements of these languages, also called *agents*, are defined from the tell, ask, get and nask primitives by possibly combining them by the classical choice operator $+$, used among others in CCS, parallel operator (denoted by the $||$ symbol) and the sequential operator (denoted by the $;$ symbol). The formal definition is as follows.

Definition 9. *Define the Bach language $\mathscr{L}_B$ as the set of agents A generated by the following grammar:*

$$A \quad ::= \quad T \mid A \,;\, A \mid A \,||\, A \mid A + A$$

where T represents a token-based primitive. Define the Dense Bach language $\mathscr{L}_{DB}$ similarly but by taking dense token-based primitives T_d:

$$A_d \quad ::= \quad T_d \mid A_d \,;\, A_d \mid A_d \,||\, A_d \mid A_d \,+\, A_d$$

$$(\mathbf{T_{dbd}}) \quad \frac{(t_1(m_1),\cdots,t_p(m_p)) \in \mathscr{D}(L(m))}{\langle\, tell(L(m)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t_1(m_1),\cdots,t_p(m_p)\} \,\rangle}$$

$$(\mathbf{A_{dbd}}) \quad \frac{\mathscr{D}(L(m)) \sqcap \sigma \neq \emptyset}{\langle\, ask(L(m)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$(\mathbf{G_{dbd}}) \quad \frac{(t_1(m_1),\cdots,t_p(m_p)) \in \mathscr{D}(L(m))}{\langle\, get(L(m)) \mid \sigma \cup \{t_1(m_1),\cdots,t_p(m_p)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$(\mathbf{N_{dbd}}) \quad \frac{m > 0 \text{ and } \mathscr{D}(L(m)) \sqcap \sigma = \emptyset}{\langle\, nask(L(m)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 3: Transition rules for set of token-based primitives (Dense Bach with distributed Density)

Define the Dense Bach with distributed Density language $\mathscr{L}_{DBD}$ similarly but by taking lists of token-based primitives T_{dbd}:

$$A_{dbd} \quad ::= \quad T_{dbd} \mid A_{dbd} \,;\, A_{dbd} \mid A_{dbd} \,||\, A_{dbd} \mid A_{dbd} \,+\, A_{dbd}$$

Subsequently, we shall consider sublanguages formed similarly but by considering only subsets of these primitives. In that case, if $\mathscr{H}$ denotes such a subset, then we shall write the induced sublanguages as $\mathscr{L}(\mathscr{H})$, $\mathscr{L}_{DB}(\mathscr{H})$, and $\mathscr{L}_{DBD}(\mathscr{H})$ respectively. Note that for the latter sublanguages, the tell, ask, nask and get primitives are associated with the basic pairs described above.

2.3 Transition system

To study the expressiveness of the languages, a semantics needs to be defined. As suggested in the previous subsections, we shall use an operational one, based on transition systems. For each transition system, the configuration consists of agents (summarizing the current state of the agents running on the store) and a multi-set of tokens (denoting the current state of the store). In order to express the termination of the computation of an agent, we extend the set of agents by adding a special terminating symbol E that can be seen as a completely computed agent. For uniformity purpose, we abuse the language by qualifying E as an agent. To meet the intuition, we shall always rewrite agents of the form $(E;A)$, $(E \,||\, A)$ and $(A \,||\, E)$ as A. This is technically achieved by defining the extended sets of agents as $\mathscr{L}_B \cup \{E\}$, $\mathscr{L}_{DB} \cup \{E\}$ or $\mathscr{L}_{DBD} \cup \{E\}$ and by justifying the simplifications by imposing a bimonoid structure.

The rules for the primitives of the languages have been given in Figures 1 to 3. Figure 4 details the usual rules for sequential composition, parallel composition, interpreted in an interleaving fashion, and CCS-like choice.

2.4 Observables and operational semantics

We are now in a position to define what we want to observe from the computations. Following previous work by some of the authors (see eg [7, 8, 13–15]), we shall actually take an operational semantics recording the final state of the computations, this being understood as the final store coupled to a mark indicating whether the considered computation is successful or not. Such marks are respectively denoted as δ^+ (for the successful computations) and δ^- (for failed computations).

$$(\mathbf{S}) \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\langle A\,;\,B \mid \sigma\rangle \longrightarrow \langle A'\,;\,B \mid \sigma'\rangle}$$

$$(\mathbf{P}) \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A \,||\, B \mid \sigma\rangle \longrightarrow \langle A' \,||\, B \mid \sigma'\rangle \\ \langle B \,||\, A \mid \sigma\rangle \longrightarrow \langle B \,||\, A' \mid \sigma'\rangle\end{array}}$$

$$(\mathbf{C}) \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A + B \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle \\ \langle B + A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle\end{array}}$$

Figure 4: Transition rules for the operators

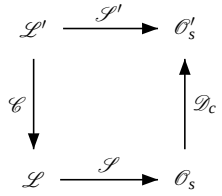

Figure 5: Basic embedding.

Definition 10.

1. *Define the set of stores Sstore as the set of finite multisets with elements from Stoken.*
2. *Let δ^+ and δ^- be two fresh symbols denoting respectively success and failure. Define the set of histories Shist as the cartesian product Sstore $\times \{\delta^+, \delta^-\}$.*
3. *For each language $\mathscr{L}_I$ of the languages $\mathscr{L}_B$, $\mathscr{L}_{DB}$, $\mathscr{L}_{DBD}$, define the operational semantics $\mathscr{O}_I : \mathscr{L}_I \to \mathscr{P}(Shist)$ as the following function: for any agent $A \in \mathscr{L}$*

$$\begin{array}{rcl}\mathscr{O}(A) & = & \{(\sigma, \delta^+) : \langle A|\emptyset\rangle \to^* \langle E|\sigma\rangle\} \\ & & \cup \{(\sigma, \delta^-) : \langle A|\emptyset\rangle \to^* \langle B|\sigma\rangle \not\to, B \neq E\}\end{array}$$

3 Comparison of Dense Bach and Dense Bach with Distributed Density

This section focusses on the comparison between the Dense Bach language and the newly introduced Dense Bach with Distrbuted Density language.

3.1 Modular embedding

A natural way to compare the expressive power of two languages is to determine 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 conserving the same observable behaviors.

According to this intuition, Shapiro introduced in [17] a first notion of embedding as follows. Consider two languages $\mathscr{L}$ and $\mathscr{L}'$. Assume given the semantics mappings (*Observation criteria*) $\mathscr{S} : \mathscr{L} \to \mathscr{O}_s$ and $\mathscr{S}' : \mathscr{L}' \to \mathscr{O}'_s$, where $\mathscr{O}_s$ and $\mathscr{O}'_s$ are on some suitable domains. Then $\mathscr{L}$ can *embed* $\mathscr{L}'$ if there exists a mapping $\mathscr{C}$ (coder) from the statements of $\mathscr{L}'$ to the statements of $\mathscr{L}$, and a mapping

$\mathscr{D}_c$ (decoder) from $\mathscr{O}_s$ to $\mathscr{O}'_s$, such that the diagram of Figure 5 commutes, namely such that for every statement $A \in \mathscr{L}' : \mathscr{D}_c(\mathscr{S}(\mathscr{C}(A))) = \mathscr{S}'(A)$.

This basic notion of embedding turns out however to be too weak since, for instance, the above equation is satisfied by any pair of Turing-complete languages. de Boer and Palamidessi hence proposed in [1] to add three constraints on the coder $\mathscr{C}$ and on the decoder $\mathscr{D}_c$ in order to obtain a notion of *modular* embedding usable for concurrent languages:

1. $\mathscr{D}_c$ should be defined in an element-wise way with respect to $\mathscr{O}_s$, namely for some appropriate mapping $\mathscr{D}_{el}$
$$\forall X \in \mathscr{O}_s : \ \mathscr{D}_c(X) = \{\mathscr{D}_{el}(x) \mid x \in X\} \qquad (P_1)$$

2. the coder $\mathscr{C}$ should be defined in a compositional way with respect to the sequential, parallel and choice operators:
$$\begin{array}{c} \mathscr{C}(A\,;\,B) = \mathscr{C}(A)\,;\,\mathscr{C}(B) \\ \mathscr{C}(A \,||\, B) = \mathscr{C}(A) \,||\, \mathscr{C}(B) \\ \mathscr{C}(A \,+\, B) = \mathscr{C}(A) \,+\, \mathscr{C}(B) \end{array} \qquad (P_2)$$

3. the embedding should preserve the behavior of the original processes with respect to deadlock, failure and success (*termination invariance*):
$$\forall X \in \mathscr{O}_s, \forall x \in X : \ tm'(\mathscr{D}_{el}(x)) = tm(x) \qquad (P_3)$$
where *tm* and *tm'* extract the termination information from the observables of $\mathscr{L}$ and $\mathscr{L}'$, 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 $\mathscr{L}'$ into $\mathscr{L}$ is subsequently denoted by $\mathscr{L}' \leq \mathscr{L}$. It is easy to prove that $\leq$ is a pre-order relation. Moreover if $\mathscr{L}' \subseteq \mathscr{L}$ then $\mathscr{L}' \leq \mathscr{L}$ that is, any language embeds all its sublanguages. This property descends immediately from the definition of embedding, by setting $\mathscr{C}$ and $\mathscr{D}_c$ equal to the identity function.

3.2 Formal propositions and proofs

Let us now turn to the formal proofs. As a first result, thanks to the fact that any language contains its sublanguages, a number of modular embeddings are directly established. In subsequent proofs, this is referred to by *language inclusion*.

Proposition 1. *$\mathscr{L}_{DBD}(\psi) \leq \mathscr{L}_{DBD}(\chi)$, for any subsets of ψ, χ of primitives such that $\psi \subseteq \chi$.*

A second observation is that Dense Bach primitives are deduced from the primitives of Dense Bach with Distributed Density by taking dense sets with only one token, the density being the same. As a result Dense Bach sublanguages are embedded in the corresponding Dense Bach with Distributed Density sublanguages.

Proposition 2. *$\mathscr{L}_{DB}(\chi) \leq \mathscr{L}_{DBD}(\chi)$, for any subset of χ of primitives.*

Proof. Immediate by defining the coder as follows:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathscr{C}(tell(t(m))) & = & tell([t](m)) & \mathscr{C}(get(t(m))) & = & get([t](m)) \\ \mathscr{C}(ask(t(m))) & = & ask([t](m)) & \mathscr{C}(nask(t(m))) & = & nask([t](m)) \end{array}$$

□

Proposition 3. *$\mathscr{L}_{DB}(tell)$ and $\mathscr{L}_{DBD}(tell)$ are equivalent.*

Proof. Indeed, thanks to proposition 2, $\mathscr{L}_{DB}(\text{tell}) \leq \mathscr{L}_{DBD}(\text{tell})$. Furthermore, as to each distribution of tokens from dense set of tokens $\mathscr{D}([t_1,\cdots,t_p](m))$ is associated a finite set of tuple of dense tokens $\{(t_1(m_1),\cdots,t_p(m_p)) : m_1+\cdots+m_p = m\}$, by coding any $tell(L(m))$ primitive as $tell(t_1(m_1))$; $\cdots$; $tell(t_p(m_p))$ primitives, and by using the identity as decoder, one establishes that $\mathscr{L}_{DBD}(\text{tell}) \leq \mathscr{L}_{DB}(\text{tell})$. □

As a result of the expressiveness hierarchy of [12], it also comes that both languages $\mathscr{L}_{DB}$(ask,tell) and $\mathscr{L}_{DB}$(nask,tell) are strictly more expressive than $\mathscr{L}_{DBD}$(tell) since both have been established strictly more expressive than $\mathscr{L}_{DB}$(tell).

Let us now compare $\mathscr{L}_{DB}$(ask,tell) with its distributed dense counterpart.

Proposition 4. *$\mathscr{L}_{DB}(ask,tell) < \mathscr{L}_{DBD}(ask,tell)$*

Proof. On the one hand, $\mathscr{L}_{DB}(\text{ask,tell}) \leq \mathscr{L}_{DBD}(\text{ask,tell})$, by proposition 2. On the other hand, $\mathscr{L}_{DBD}(\text{ask,tell}) \not\leq \mathscr{L}_{DB}(\text{ask,tell})$ may be established by contradiction. The proof proceeds by exploiting the inability of $\mathscr{L}_{DB}$(ask,tell) to atomically test the presence of two distinct tokens a and b. Assume thus the existence of a coder $\mathscr{C} : \mathscr{L}_{DBD}(\text{ask,tell}) \to \mathscr{L}_{DB}(\text{ask,tell})$ and consider $AB = ask([a,b](2))$. Let us prove that its coder is empty, which is absurd since, by definition 9, it should contain at least one primitive. To that end, one may assume that $\mathscr{C}(AB)$ is in normal form [6] and thus is written as $tell(\overline{t_1});A_1+\cdots+tell(\overline{t_p});A_p+ask(\overline{u_1});B_1+\cdots+ask(\overline{u_q});B_q$, where $\overline{t_i}$ and $\overline{u_j}$ denote the token t_i and u_j associated to a density. In this expression, we will establish that there is no alternative guarded by a $tell(\overline{t_i})$ operation, and no alternative guarded by a $ask(\overline{u_j})$ operation either, in which case $\mathscr{C}(AB)$ is empty.

Let us first establish by contradiction that there is no alternative guarded by a $tell(\overline{t_i})$ operation. Assume there is one, say guarded by $tell(\overline{t_i})$. Then $D = \langle \mathscr{C}(AB)|\emptyset\rangle \to \langle A_i|\overline{t_i}\rangle$ is a valid computation prefix of $\mathscr{C}(AB)$. It should deadlocks afterwards since $\mathscr{O}(AB) = (\emptyset,\delta^-)$. However D is also a valid computation prefix of $\mathscr{C}(AB+tell([a](1)))$. Hence, $\mathscr{C}(AB+tell([a](1)))$ admits a failing computation which contradicts the fact that $\mathscr{O}(AB+tell([a](1))) = (\{a\},\delta^+)$.

Secondly we establish that there is also no alternative guarded by an $ask(\overline{u_j})$ operation. To that end, let us first consider two auxiliary computations: as $\mathscr{O}(tell([a](1))) = (\{a\},\delta^+)$, any computation of $\mathscr{C}(tell([a](1)))$ starting in the empty store succeeds. Let $\langle \mathscr{C}(tell([a](1)))|\emptyset\rangle \to \cdots \to \langle E|\{a_1,\ldots,a_m\}\rangle$ be such a computation. Similarly, let $\langle \mathscr{C}(tell([b](1)))|\emptyset\rangle \to \cdots \to \langle E|\{b_1,\ldots,b_n\}\rangle$ be one computation of $\mathscr{C}(tell([b](1)))$. The proof of the claim proceeds in two steps. First let us prove that none of the u_i's belong to $\{a_1,\ldots,a_m\}$. By contradiction, assume that $u_i = a_k$ for some k and that d is the density associated to u_i, namely, $\overline{u_i} = u_i(d)$. Let us observe that, since it is in $\mathscr{L}_{DB}$(ask,tell), the considered computation of $\mathscr{C}(tell([a](1)))$ can be repeated sequentially, as many times as needed. As a result, by using A^d to denote the sequential composition of d instances of A, the sequence $D' = \langle \mathscr{C}(tell([a](1))^d;AB)|\emptyset\rangle \to \cdots \to \langle AB|\{a_1^d,\ldots,a_m^d\}\rangle \to \langle B_j|\{a_1^d,\ldots,a_m^d\}\rangle$ is a valid computation prefix of $\mathscr{C}(tell([a](1))^d;AB)$, which can only be continued by failing suffixes. However D' induces the following computation prefix D'' for $tell([a](1))^d;(AB+ask([a](1)))$ which admits only successful computations: $D'' = \langle \mathscr{C}(tell([a](1))^d;(AB+ask([a](1))))|\emptyset\rangle \to \cdots \to \langle AB+ask([a](1))|\{a_1^d,\ldots,a_m^d\}\rangle \to \langle B_j|\{a_1^d,\ldots,a_m^d\}\rangle$. The proof proceeds similarly in the case $u_j \in b_1,\ldots,b_n$ for some $j \in 1,\ldots,q$ by then considering $tell([b](1))^d;AB$ and $tell([b](1))^d;(AB+ask([b](1)))$.

Finally, the fact that the $u_i's$ do not belong to $\{a_1,\ldots,a_m\} \cup \{b_1,\ldots,b_n\}$ induces a contradiction. Indeed, if this is the case then $\langle \mathscr{C}(tell([a](1));tell([b](1));AB)|\emptyset\rangle \to \cdots \to$

$\langle tell([b](1));AB)|\{a_1,\ldots,a_m\}\rangle \rightarrow \cdots \rightarrow \langle AB|\{a_1,\ldots,a_m,b_1,\ldots,b_n\}\rangle \not\rightarrow$ is a valid failing computation prefix of $\mathscr{C}(tell([a](1));tell([b](1));AB)$ whereas $tell([a](1));tell([b](1));AB$ has only one successful computation. As a conclusion, $\mathscr{C}(AB)$ is equivalent to an empty statement, which is absurd by definition 9. □

Symmetrically, $\mathscr{L}_{DB}$(nask,tell) is strictly less expressive than $\mathscr{L}_{DBD}$(nask,tell).

Proposition 5. $\mathscr{L}_{DB}(nask,tell) < \mathscr{L}_{DBD}(nask,tell)$.

Proof. On the one hand, $\mathscr{L}_{DB}$(nask,tell) $\leq \mathscr{L}_{DBD}$(nask,tell) holds by proposition 2. On the other hand, $\mathscr{L}_{DBD}$(nask,tell) $\not\leq \mathscr{L}_{DB}$(nask,tell) is proved by contradiction, assuming the existence of a coder $\mathscr{C}$. The proof proceeds as in proposition 4 but this time by exploiting the inability of $\mathscr{L}_{DB}$(nask,tell) to atomically test the absence of two distinct tokens *a* and *b*.

□

$\mathscr{L}_{DBD}$(nask,tell) and $\mathscr{L}_{DB}$(ask,tell) are not comparable with each other, as well as $\mathscr{L}_{DBD}$(ask,tell) with regards to $\mathscr{L}_{DB}$(nask,tell).

Proposition 6.

(i) $\mathscr{L}_{DBD}(nask,tell) \not\leq \mathscr{L}_{DB}(ask,tell)$
(ii) $\mathscr{L}_{DB}(ask,tell) \not\leq \mathscr{L}_{DBD}(nask,tell)$
(iii) $\mathscr{L}_{DBD}(ask,tell) \not\leq \mathscr{L}_{DB}(nask,tell)$
(iv) $\mathscr{L}_{DB}(nask,tell) \not\leq \mathscr{L}_{DBD}(ask,tell)$

Proof. **(i)** Otherwise we have $\mathscr{L}_{DB}$(nask,tell) $\leq \mathscr{L}_{DB}$(ask,tell) which has been proved impossible in [12]. **(ii)** By contradiction, consider $A = tell(t(1))\ ;\ ask(t(1)))$. One has $\mathscr{O}(A) = \{(\{t(1)\},\delta^+)\}$. Hence, by P_3, $\mathscr{C}(A)$ succeeds whereas we shall establish that it has failing computations. Indeed, since $\mathscr{O}(ask(t(1))) = \{(\emptyset,\delta^-)\}$, any computation of $\mathscr{C}(ask(t(1)))$ starting on the empty store fails. As $\mathscr{C}(ask(t(1)))$ is composed of nask and tell primitives, this can only occur by having a nask primitive preceded by a tell primitive. As enriching the initial content of the store leads to the same result, any computation starting on any (arbitrary) store fails. As a consequence, even if $\mathscr{C}(tell(t(1)))$ has a successful computation, this computation cannot be continued by a successful computation of $\mathscr{C}(ask(t(1)))$. Consequently any computation of $\mathscr{C}(tell(t(1));ask(t(1)))$ fails, which produces a contradiction. **(iii)** Otherwise we would have $\mathscr{L}_{DB}$(ask,tell) $\leq \mathscr{L}_{DB}$(nask,tell) which has been proved impossible in [12]. **(iv)** By contradiction, consider $A = tell(t(1))\ ;\ nask(t(1)))$. One has $\mathscr{O}(A) = \{(\{t\},\delta^-)\}$. By P_3, $\mathscr{C}(A)$ fails, whereas we shall establish that it has a successful computation. Indeed, since $\mathscr{O}(tell(t(1))) = \{(\{t(1)\},\delta^+)\}$, any computation of $\mathscr{C}(tell(t(1)))$ starting on the empty store is successful. Similarly, it follows from $\mathscr{O}(nask(t(1))) = \{(\emptyset,\delta^+)\}$ that any computation of $\mathscr{C}(nask(t(1)))$ starting on the empty store is successful, and, consequently, is any computation starting from any store, since $\mathscr{C}(nask(t(1)))$ is composed of ask and tell primitives. Summing up, any (successful) computation of $\mathscr{C}(tell(t))$ starting on the empty store can be continued by a (successful) computation of $\mathscr{C}(nask(t))$, which leads to the contradiction. □

$\mathscr{L}_{DBD}$(nask,tell) and $\mathscr{L}_{DBD}$(ask,tell) are not comparable with each other, as well as $\mathscr{L}_{DBD}$(nask,tell) with regards to $\mathscr{L}_{DB}$(ask,nask,tell).

Proposition 7.

(i) $\mathscr{L}_{DBD}(nask,tell) \not\leq \mathscr{L}_{DBD}(ask,tell)$
(ii) $\mathscr{L}_{DBD}(ask,tell) \not\leq \mathscr{L}_{DBD}(nask,tell)$
(iii) $\mathscr{L}_{DB}(ask,nask,tell) \not\leq \mathscr{L}_{DBD}(nask,tell)$
(iv) $\mathscr{L}_{DBD}(nask,tell) \not\leq \mathscr{L}_{DB}(ask,nask,tell)$

Proof. **(i)** Otherwise $\mathscr{L}_{DB}(\text{nask,tell}) \leq \mathscr{L}_{DBD}(\text{ask,tell})$, which contradicts proposition 6(iv). **(ii)** and **(iii)** Otherwise $\mathscr{L}_{DB}(\text{ask,tell}) \leq \mathscr{L}_{DBD}(\text{nask,tell})$, which contradicts proposition 6(ii). **(iv)** The proof proceeds as in proposition 5(ii). The presence of the ask primitive in $\mathscr{L}_{DB}$ does not modify the reasoning, as it does not destroy elements and so does not modify the state of the store σ. □

Symmetrically, $\mathscr{L}_{DB}(\text{get,tell})$ and $\mathscr{L}_{DBD}(\text{ask,tell})$ are not comparable with each other, as $\mathscr{L}_{DB}(\text{get,tell})$ and $\mathscr{L}_{DBD}(\text{nask,tell})$ are not comparable with each other.

Proposition 8.

(i) $\mathscr{L}_{DB}(get,tell) \not\leq \mathscr{L}_{DBD}(ask,tell)$ (iii) $\mathscr{L}_{DB}(get,tell) \not\leq \mathscr{L}_{DBD}(nask,tell)$

(ii) $\mathscr{L}_{DBD}(ask,tell) \not\leq \mathscr{L}_{DB}(get,tell)$ (iv) $\mathscr{L}_{DBD}(nask,tell) \not\leq \mathscr{L}_{DB}(get,tell)$

Proof. **(i)** By contradiction, consider $tell(t(1))$; $get(t(1))$. One has $\mathscr{O}(tell(t(1)) \,;\, get(t(1))) = \{(\emptyset, \delta^+)\}$. By P_2 and P_3, any computation of $\mathscr{O}(\mathscr{C}(tell(t(1))) \,;\, \mathscr{C}(get(t(1))))$ is thus successful. Since $\mathscr{C}(get(t(1)))$ is composed of ask and tell primitives only and since ask and tell primitives do not destroy elements, at least one computation of $\mathscr{O}(\mathscr{C}(tell(t(1))) \,;\, \mathscr{C}(get(t(1))) \,;\, \mathscr{C}(get(t(1))))$ is successful. However, $\mathscr{O}(tell(t(1)) \,;\, get(t(1)) \,;\, get(t(1))) = \{(\emptyset, \delta^-)\}$, which provides the contradiction.

(ii) The proof is established by contradiction. Intuitively, $\mathscr{L}_{DB}(\text{get,tell})$ is unable to atomically test the presence of a and b. Let us thus consider $AB = ask([a,b](2))$ and prove that its coder has a successful computation. This leads to a contradiction since AB has just one failing computation. To that end, one may assume that $\mathscr{C}(AB)$ is in normal form (see [6]) and thus is written as $tell(\overline{t_1}); A_1 + \cdots + tell(\overline{t_p}); A_p + get(\overline{u_1}); B_1 + \cdots + get(\overline{u_q}); B_q$, where $\overline{t_i}$ and $\overline{u_j}$ denote the token t_i and u_j associated to a density.

The proof proceeds by establishing that (I) there is no alternative guarded by a $tell(\overline{t_i})$ operation, and (II) there is no alternative guarded by a $get(\overline{u_j})$ operation. In which case, $\mathscr{C}(AB)$ is equivalent to an empty statement, which is not possible in view of definition 9.

CASE I: there is no alternative guarded by a $tell(\overline{t_i})$ operation. Otherwise, $D = \langle \mathscr{C}(AB) | \emptyset \rangle \rightarrow \langle A_i | \{\overline{t_i}\} \rangle$ would be a valid computation prefix of $\mathscr{C}(AB)$ which should deadlocks afterwards since $\mathscr{O}(AB) = \{(\emptyset, \delta^-)\}$. However D is also a valid computation prefix of $\mathscr{C}(AB + tell([a](1)))$. Hence, $\mathscr{C}(AB + tell([a](1)))$ admits a failing computation which contradicts the fact that $\mathscr{O}(AB + tell([a](1))) = (\{a\}, \delta^+)$.

CASE II: there is no alternative guarded by a $get(\overline{u_j})$ operation. To that end, let us first consider two auxiliary computations: as $\mathscr{O}(tell([a](1))) = \{(\{a\}, \delta^+)\}$, any computation of $\mathscr{C}(tell([a](1)))$ starting in the empty store succeeds. Let $\langle \mathscr{C}(tell([a](1))) | \emptyset \rangle \rightarrow \cdots \rightarrow \langle E | \{a_1, \ldots, a_m\} \rangle$ be such a computation. Similarly, let $\langle (tell([b](1))) | \emptyset \rangle \rightarrow \cdots \rightarrow \langle E | \{b_1, \ldots, b_n\} \rangle$ be one computation of $\mathscr{C}(tell([b](1)))$. As these two computations start by assuming no token on the store and since $\mathscr{L}_{DB}(\text{get,tell})$ does not contain negative tests, it is easy to verify that they can be put sequentially so as to establish the following computations:

$$\begin{array}{lll} \langle \mathscr{C}(tell([a](1)); tell([b](1))) | \emptyset \rangle & \rightarrow \cdots \rightarrow & \langle \mathscr{C}(tell([b](1))) | \{a_1, \ldots, a_m\} \rangle \\ & \rightarrow \cdots \rightarrow & \langle E | \{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_n\} \rangle \\ \langle \mathscr{C}(tell([b](1)); tell([a](1))) | \emptyset \rangle & \rightarrow \cdots \rightarrow & \langle \mathscr{C}(tell([a](1))) | \{b_1, \ldots, b_n\} \rangle \\ & \rightarrow \cdots \rightarrow & \langle E | \{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_n\} \rangle \end{array}$$

As $\mathscr{C}(tell([a](1)); tell([b](1)); AB)$ has a successful computation, one of the $get(\overline{u_i})$ succeeds, and, consequently, one has $\{\overline{u_j}\} \subseteq \{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_n\}$ for some j. Assume $\overline{u_j} = a_k$ for k and let d be the density associated to u_j, namely, $\overline{u_j} = a_k(d)$. Then

$$D' = \langle \mathscr{C}(tell([a](1)); AB) | \emptyset \rangle \rightarrow \cdots \rightarrow \langle \mathscr{C}(AB) | \{a_1, \ldots, a_m\} \rangle \rightarrow \langle B_j | \{a_1, \ldots, a_m\} \setminus \{\overline{u_j}\} \rangle$$

is a valid computation prefix of $\mathscr{C}(tell([a](1));AB)$. It can only be continued by failing suffixes since $tell([a](1));AB$ fails. However, this induces the following computation prefix D'' for $\mathscr{C}(tell([a](1));(AB+ask([a](1))))$ and thus a failing computation whereas $tell([a](1));(AB+ask([a](1)))$ only admits a successful computation:

$$\begin{array}{rcl} D'' = \langle \mathscr{C}(tell([a](1));(AB+ask([a](1))))|\emptyset\rangle & \to \cdots \to & \langle AB+ask([a](1))|\{a_1,\ldots,a_m\}\rangle \\ & \to & \langle B_j|\{a_1,\ldots,a_m\}\setminus\{\overline{u_j}\}\rangle. \end{array}$$

The proof proceeds similarly in the case $u_j = b_k$ for some k by then considering $tell([b](1));AB$ and $tell([b](1));(AB+ask([b](1)))$. **(iii)** Otherwise, $\mathscr{L}_{DB}$(nask,tell) would be embedded in $\mathscr{L}_{DB}$(get,tell), which has been prooved impossible in proposition 12(iv) of [12]. **(iv)** Otherwise, $\mathscr{L}_{DB}$(ask,tell) would be embedded in $\mathscr{L}_{DBD}$(nask,tell) which contradicts proposition 6(ii).

□

Let us now prove that $\mathscr{L}_{DB}$(get,tell) is not comparable with $\mathscr{L}_{DBD}$(ask,nask,tell).

Proposition 9.

(i) $\mathscr{L}_{DB}(get,tell) \not\leq \mathscr{L}_{DBD}(ask,nask,tell)$
(ii) $\mathscr{L}_{DBD}(ask,nask,tell) \not\leq \mathscr{L}_{DB}(get,tell)$

Proof. **(i)** By contradiction, consider $tell(t(1))$; $get(t(1))$, for which $\mathscr{O}((tell(t(1)) ; get(t(1))) = \{(\emptyset,\delta^+)\}$. Hence, by P_2 and P_3, any computation of $\mathscr{C}(tell(t(1)))$; $\mathscr{C}(get(t(1)))$ is successful. Such a computation is composed of a computation for $\mathscr{C}(tell(t(1)))$ followed by a computation for $\mathscr{C}(get(t(1)))$. As the latter is composed of ask, nask, and tell primitives which do not destroy elements on the store, the latter computation can be repeated step by step which yields a successful computation for $\mathscr{C}(tell(t(1)))$; $(\mathscr{C}(get(t(1))) \,||\, \mathscr{C}(get(t(1))))$. However, $\mathscr{O}(tell(t(1)) ; (get(t(1)) \,||\, get(t(1))) = \{(\emptyset,\delta^-)\}$, which produces the announced contradiction. **(ii)** Otherwise, $\mathscr{L}_{DB}$(nask,tell) would be embedded in $\mathscr{L}_{DB}$(get,tell) which has been prooved impossible in proposition 12(iv) of [12]. □

Let us now include the get primitive in the Dense Bach with Distributed Density language. We first prove that $\mathscr{L}_{DBD}$(get,tell) is embedded in $\mathscr{L}_{DBD}$(ask,get,tell), but is not embedded in $\mathscr{L}_{DBD}$(ask,tell).

Proposition 10. *$\mathscr{L}_{DBD}(get,tell) \leq \mathscr{L}_{DBD}(ask,get,tell)$ and $\mathscr{L}_{DBD}(get,tell) \not\leq \mathscr{L}_{DBD}(ask,tell)$*

Proof. **(i)** One has $\mathscr{L}_{DBD}$(get,tell) $\leq \mathscr{L}_{DBD}$(ask,get,tell) by language inclusion. **(ii)** By contradiction, consider A = $tell([t](1))$; $get([t](1))$. One has $\mathscr{O}(A) = \{(\emptyset,\delta^+)\}$. By P_2 and P_3, any computation of $\mathscr{O}(\mathscr{C}(tell([t](1))) ; \mathscr{C}(get([t](1))))$ is thus successful. Such a computation is composed of a computation for $\mathscr{C}(tell([t](1)))$ followed by a computation for $\mathscr{C}(get([t](1)))$. As $\mathscr{C}(get([t](1)))$ is composed of ask and tell primitives and since ask and tell primitives do not destroy elements, this latter computation can be repeated, which yields successful computations for $\mathscr{O}(\mathscr{C}(tell([t](1))) ; \mathscr{C}(get([t](1))) ; \mathscr{C}(get([t](1))))$. However, $\mathscr{O}(tell([t](1)) ; get([t](1)) ; get([t](1))) = \{(\emptyset,\delta^-)\}$, which leads to the contradiction.

□

Let us now establish that $\mathscr{L}_{DB}$(get,tell) is strictly less expressive than $\mathscr{L}_{DBD}$(get,tell).

Proposition 11. *$\mathscr{L}_{DB}(get,tell) < \mathscr{L}_{DBD}(get,tell)$*

Proof. On the one hand, $\mathscr{L}_{DB}$(get,tell) $\leq \mathscr{L}_{DBD}$(get,tell) holds by proposition 2. On the other hand, $\mathscr{L}_{DBD}$(get,tell) $\not\leq \mathscr{L}_{DB}$(get,tell) may be proved exactly as in proposition 8(ii), where we replace any occurrence of ask([a,b](2)) by get([a,b](2)). □

In order to use once more the reasoning of proposition 8(ii), we now prove that $\mathscr{L}_{DBD}$(ask,tell) is not comparable with $\mathscr{L}_{DB}$(nask,get,tell).

Proposition 12.

(i) $\mathscr{L}_{DBD}(ask,tell) \not\leq \mathscr{L}_{DB}(nask,get,tell)$
(ii) $\mathscr{L}_{DB}(nask,get,tell) \not\leq \mathscr{L}_{DBD}(ask,tell)$

Proof. **(i)** The proof proceeds as in proposition 8(ii), by constructing a successful coded computation for the same failing agent $ask([a,b](2))$ with the alternatives guarded by a nask primitive of the normal form of the coded version treated as the alternatives guarded by a tell primitive. **(ii)** Otherwise $\mathscr{L}_{DB}$(nask,tell) $\leq \mathscr{L}_{DBD}$(ask,tell) which contradicts proposition 6(iv). □

We can now prove that $\mathscr{L}_{DBD}$(get,tell) is not comparable with respectively $\mathscr{L}_{DB}$(nask,tell), $\mathscr{L}_{DBD}$(nask,tell), $\mathscr{L}_{DB}$(nask,get,tell), $\mathscr{L}_{DBD}$(ask,nask,tell) and $\mathscr{L}_{DB}$(ask,nask,tell).

Proposition 13.

(i) $\mathscr{L}_{DBD}(get,tell) \not\leq \mathscr{L}_{DB}(nask,tell)$
(ii) $\mathscr{L}_{DB}(nask,tell) \not\leq \mathscr{L}_{DBD}(get,tell)$
(iii) $\mathscr{L}_{DBD}(get,tell) \not\leq \mathscr{L}_{DBD}(nask,tell)$
(iv) $\mathscr{L}_{DBD}(nask,tell) \not\leq \mathscr{L}_{DBD}(get,tell)$
(v) $\mathscr{L}_{DBD}(get,tell) \not\leq \mathscr{L}_{DB}(nask,get,tell)$
(vi) $\mathscr{L}_{DB}(nask,get,tell) \not\leq \mathscr{L}_{DBD}(get,tell)$
(vii) $\mathscr{L}_{DBD}(get,tell) \not\leq \mathscr{L}_{DBD}(ask,nask,tell)$
$(viii)$ $\mathscr{L}_{DBD}(ask,nask,tell) \not\leq \mathscr{L}_{DBD}(get,tell)$
(ix) $\mathscr{L}_{DBD}(get,tell) \not\leq \mathscr{L}_{DB}(ask,nask,tell)$
(x) $\mathscr{L}_{DB}(ask,nask,tell) \not\leq \mathscr{L}_{DBD}(get,tell)$

Proof. **(i)** Indeed, otherwise we have $\mathscr{L}_{DB}$(ask,tell) $\leq \mathscr{L}_{DB}$(nask,tell) which has been proved impossible in [12]. **(ii)** By contradiction, consider $A = tell(t(1))$; $nask(t(1))$, for which $\mathscr{O}(A) = \{(\{t(1)\}, \delta^-)\}$. Then, by P_2 and P_3, any computation of $\mathscr{C}(tell(t(1)))$; $\mathscr{C}(nask(t(1)))$ must fail whereas we shall establish that $\mathscr{C}(tell(t(1)))$; $\mathscr{C}(nask(t(1)))$ has a successful computation. Indeed, let us observe that $\mathscr{O}(tell(t(1))) = \{(\{t(1)\}, \delta^+)\}$ and $\mathscr{O}(nask(t(1))) = \{(\emptyset, \delta^+)\}$. For both cases, by P_3, any computation of $\mathscr{C}(tell(t(1)))$ and $\mathscr{C}(nask(t(1)))$ starting on the empty store is successful. Consequently, since $\mathscr{C}(tell(t(1)))$ and $\mathscr{C}(nask(t(1)))$ are composed of get and tell primitives, so are all of their computations starting from any store. Therefore, any (successful) computation of $\mathscr{C}(tell(t(1)))$ starting on the empty store can be continued by a (successful) computation of $\mathscr{C}(nask(t(1)))$, which leads to the contradiction. **(iii)** Otherwise we have $\mathscr{L}_{DB}$(ask,tell) $\leq \mathscr{L}_{DBD}$(nask,tell) which contradicts proposition 6(ii). **(iv)** Otherwise we have $\mathscr{L}_{DB}$(nask,tell) $\leq \mathscr{L}_{DBD}$(get,tell) which contradicts (ii) above. **(v)** Similar to proposition 11. **(vi)** Otherwise $\mathscr{L}_{DB}$(nask,tell) $\leq \mathscr{L}_{DBD}$(get,tell) which contradicts (ii) above. **(vii)** By contradiction. Let us first observe that $\mathscr{O}(tell([t](1)) \; ; \; get([t](1))) = \{(\emptyset, \delta^+)\}$. By P_2 and P_3 any computation of $(\mathscr{C}(tell([t](1))) \; ; \; \mathscr{C}(get([t](1))))$ starting in the empty store is thus successful. By repeating step by step the computation of $\mathscr{C}(get([t](1)))$, this leads to a successful computation of $(\mathscr{C}(tell([t](1))) \; ; \; (\mathscr{C}(get([t](1))) \;||\; \mathscr{C}(get([t](1)))))$ starting in the empty store. However, $\mathscr{O}(tell([t](1)) \; ; \; (get([t](1)) \;||\; get([t](1)))) = \{(\emptyset, \delta^-)\}$, which leads to the contradiction. **(viii)** Otherwise $\mathscr{L}_{DBD}$(nask,tell) $\leq \mathscr{L}_{DBD}$(get,tell) which contradicts proposition (iv) above. **(ix)** Otherwise $\mathscr{L}_{DBD}$(get,tell) $\leq \mathscr{L}_{DBD}$(ask,nask,tell) which contradicts (vii) above. **(x)** Otherwise $\mathscr{L}_{DB}$(nask,tell) $\leq$ $\mathscr{L}_{DBD}$(get,tell) which contradicts (ii) above. □

Let us now establish that $\mathscr{L}_{DBD}$(nask,tell) and $\mathscr{L}_{DB}$(ask,nask,tell) are strictly less expressive than $\mathscr{L}_{DBD}$(ask,nask,tell).

Proposition 14.

(i) $\mathscr{L}_{DBD}(nask,tell) < \mathscr{L}_{DBD}(ask,nask,tell)$
(ii) $\mathscr{L}_{DB}(ask,nask,tell) < \mathscr{L}_{DBD}(ask,nask,tell)$

Proof. **(i)** By sublanguage inclusion, one has $\mathscr{L}_{DBD}(\text{nask,tell}) \leq \mathscr{L}_{DBD}(\text{ask,nask,tell})$. Moreover, if we had $\mathscr{L}_{DBD}(\text{ask,nask,tell}) \leq \mathscr{L}_{DBD}(\text{nask,tell})$, then we would have $\mathscr{L}_{DBD}(\text{ask,tell}) \leq \mathscr{L}_{DBD}(\text{nask,tell})$, which contradicts proposition 7(ii).

(ii) Let us thus proceed by contradiction by assuming the existence of a coder $\mathscr{C}$ from $\mathscr{L}_{DBD}(\text{ask,nask,tell})$ to $\mathscr{L}_{DB}(\text{ask,nask,tell})$. Let n be the cumulative occurrences of tokens in *nask* primitives of $\mathscr{C}(tell([a](1)))$.

As $\mathscr{C}(tell([a](1)))$ has only successful computations, let S_a be the store resulting from one of them. Moreover, as a matter of notation, let the construction $A^{||q}$ denote the parallel composition of q copies of A. As $(tell([b](1)))^{||(n+2)}$; $tell([a](1)))$ succeeds as well, let S'_b denote the store resulting from one successful computation of its coding. Consider finally $ABs = ask([a,b](n+4))$ with the intuitive aim of requesting one a with $n+3$ copies of b. Consider $\mathscr{C}(ABs)$ in its normal form:

$$\begin{array}{l} tell(\overline{t_1})\ ;\ A_1 + \cdots + tell(\overline{t_p})\ ;\ A_p \\ + ask(\overline{u_1})\ ;\ B_1 + \cdots + ask(\overline{u_q})\ ;\ B_q \\ + nask(\overline{v_1})\ ;\ C_1 + \cdots + nask(\overline{v_r})\ ;\ C_r \end{array}$$

As in proposition 5(ii), it is possible to establish that there are no alternatives guarded by $tell(\overline{t_i})$ and $nask(\overline{v_j})$ primitives.

Let us prove that $\{u_1,\cdots,u_q\} \cap (S_a \cup S'_b) = \emptyset$. This is done in two steps by establishing that (1) $\{u_1,\cdots,u_q\} \cap S_a = \emptyset$, and that (2) $\{u_1,\cdots,u_q\} \cap S'_b = \emptyset$.

First let us prove that $\{u_1,\cdots,u_q\} \cap S_a = \emptyset$. Assume $u_i \in S_a$ and let d be the density associated to u_i, namely, $\overline{u_i} = u_i(d)$. Let us observe that each step of the considered computation of $\mathscr{C}(tell([a](1)))$ can be repeated in turn, in as many parallel occurences of it as needed, so that

$$\begin{array}{rl} P = \langle \mathscr{C}(tell([a](1))^{||d}\ ;\ ABs) | \emptyset \rangle & \\ \rightarrow \cdots \rightarrow & \langle ABs | \cup_{k=1}^{d} S_a \rangle \\ \rightarrow & \langle B_i | (\cup_{k=1}^{d} S_a) \rangle \end{array}$$

is a valid computation prefix of $\mathscr{C}(tell([a](1))^{||q}\ ;\ ABs)$, which can only be continued by failing suffixes. However P induces the following computation prefix P' for $\mathscr{C}(tell([a](1))^{||q}\ ;\ (ABs + tell([a](1))))$ which admits only successful computations:

$$\begin{array}{rl} P' = \langle \mathscr{C}(tell([a](1))^{||d}\ ;\ (ABs + tell([a](1)))) | \emptyset \rangle & \\ \rightarrow \cdots \rightarrow & \langle \mathscr{C}(ABs + tell([a](1))) | \cup_{k=1}^{d} S_a \rangle \\ \rightarrow & \langle B_i | (\cup_{k=1}^{d} S_a) \rangle \end{array}$$

leading to the contradiction.

Secondly, the proof that $\{u_1,\cdots,u_q\} \cap S'_b = \emptyset$ is established similarly by considering S'_b instead of S_a and $tell([b](1))$ instead of $tell([a](1))$. □

$\mathscr{L}_{DBD}(\text{ask,tell})$ is strictly less expressive than $\mathscr{L}_{DBD}(\text{ask,nask,tell})$.

Proposition 15. $\mathscr{L}_{DBD}(ask,tell) < \mathscr{L}_{DBD}(ask,nask,tell)$

Proof. On the one hand, $\mathscr{L}_{DBD}$(ask,tell) $\leq \mathscr{L}_{DBD}$(ask,nask,tell) results from language inclusion. On the other hand, one has $\mathscr{L}_{DBD}$(ask,nask,tell) $\not\leq \mathscr{L}_{DBD}$(ask,tell) since otherwise $\mathscr{L}_{DBD}$(nask,tell) $\leq$ $\mathscr{L}_{DBD}$(ask,tell), which contradicts proposition 7(i). □

Symmetrically to proposition 12(i) and 12(ii), $\mathscr{L}_{DB}$(nask,get,tell) is not comparable with $\mathscr{L}_{DBD}$(nask,tell).

Proposition 16.

(i) $\mathscr{L}_{DB}(nask,get,tell) \not\leq \mathscr{L}_{DBD}(nask,tell)$
(ii) $\mathscr{L}_{DBD}(nask,tell) \not\leq \mathscr{L}_{DB}(nask,get,tell)$

Proof. **(i)** Otherwise, $\mathscr{L}_{DB}$(ask,tell) $\leq \mathscr{L}_{DBD}$(nask,tell) which contradicts proposition 6(ii).

(ii) The proof proceeds by contradiction, similarly to the proofs of $\mathscr{L}_{DBD}$(nask,tell) $\not\leq$ $\mathscr{L}_{DB}$(ask,nask,tell) of proposition 7(iv), which itself extends that of $\mathscr{L}_{DBD}$(nask,tell) $\not\leq \mathscr{L}_{DB}$(nask,tell) of proposition 5(ii).

□

$\mathscr{L}_{DB}$(nask,get,tell) is not comparable with $\mathscr{L}_{DBD}$(ask,nask,tell).

Proposition 17.

(i) $\mathscr{L}_{DBD}(ask,nask,tell) \not\leq \mathscr{L}_{DB}(nask,get,tell)$
(ii) $\mathscr{L}_{DB}(nask,get,tell) \not\leq \mathscr{L}_{DBD}(ask,nask,tell)$

Proof. **(i)** Otherwise, $\mathscr{L}_{DBD}$(ask,tell) $\leq \mathscr{L}_{DB}$(nask,get,tell) which contradicts proposition 12(i). **(ii)** Resulting from proposition 9(i). □

$\mathscr{L}_{DBD}$(ask,nask,tell) is strictly less expressive than $\mathscr{L}_{DBD}$(ask,nask,get,tell).

Proposition 18. $\mathscr{L}_{DBD}(ask,nask,tell) < \mathscr{L}_{DBD}(ask,nask,get,tell)$

Proof. On the one hand, $\mathscr{L}_{DBD}$(ask,nask,tell) $\leq \mathscr{L}_{DBD}$(ask,nask,get,tell) results from language inclusion. On the other hand, $\mathscr{L}_{DBD}$(ask,nask,get,tell) $\not\leq \mathscr{L}_{DBD}$(ask,nask,tell). Otherwise, $\mathscr{L}_{DBD}$(get,tell) $\leq$ $\mathscr{L}_{DBD}$(ask,nask,tell), which contradicts proposition 13(vii). □

$\mathscr{L}_{DBD}$(get,tell) is strictly less expressive than $\mathscr{L}_{DBD}$(nask,get,tell).

Proposition 19. $\mathscr{L}_{DBD}(get,tell) < \mathscr{L}_{DBD}(nask,get,tell)$

Proof. On the one hand, $\mathscr{L}_{DBD}$(get,tell) $\leq \mathscr{L}_{DBD}$(nask,get,tell) results from language inclusion. On the other hand, $\mathscr{L}_{DBD}$(nask,get,tell) $\not\leq \mathscr{L}_{DBD}$(get,tell) is established by contradiction. Consider $tell([t](1))$; $nask([t](1))$, for which $\mathscr{O}(tell([t](1))$; $nask([t](1)) = \{(\{t(1)\}, \delta^-)\}$. Hence, by P_2 and P_3, $\mathscr{C}(tell([t](1)))$; $\mathscr{C}(nask([t](1)))$ fails. The contradiction comes then from the fact that at least one computation of $\mathscr{C}(tell([t](1)))$; $\mathscr{C}(nask([t](1)))$ starting on the empty store is successful. Indeed, as $\mathscr{O}(tell([t](1))) = \{(\{t(1)\}, \delta^+)\}$, any computation of $\mathscr{C}(tell([t](1)))$ starting on the empty store succeeds. Similarly, any computation of $\mathscr{C}(nask([t](1)))$ starting on the empty store succeeds. Moreover, as $\mathscr{C}(nask([t](1)))$ is composed of get and tell primitives only, for any store σ, $\mathscr{C}(nask([t](1)))$ admits at least one successful computation starting on σ. It follows that any computation of $\mathscr{C}(tell([t](1)))$ starting on the empty store can be continued by a (successful) computation of $\mathscr{C}(nask([t](1)))$, which leads to the announced contradiction. □

Finally, $\mathscr{L}_{DB}$(ask,nask,get,tell) can be proved strictly less expressive than $\mathscr{L}_{DBD}$(ask,nask,get,tell).

Proposition 20. $\mathscr{L}_{DB}(ask,nask,get,tell) < \mathscr{L}_{DBD}(ask,nask,get,tell)$

Proof. On the one hand, $\mathscr{L}_{DB}$(ask,nask,get,tell) $\leq \mathscr{L}_{DBD}$(ask,nask,get,tell) is directly deduced from proposition 2. On the other hand, if one had $\mathscr{L}_{DBD}$(ask,nask,get,tell) $\leq \mathscr{L}_{DB}$(ask,nask,get,tell) then $\mathscr{L}_{DBD}$(get,tell) $\leq \mathscr{L}_{DB}$(nask,get,tell) would hold, which contradicts proposition 13(v). □

4 Conclusion

This paper is written in the continuity of our previous research on the expressiveness of Linda-like languages. It has presented an extension of our Dense Bach language, that had promoted the notions of density and dense tokens. The new language, called Dense Bach with Distributed Density, proposes to distribute the density on a finite list of tokens manipulated by the four classical primitives of the language. Technically this is achieved by associating a positive number, called density, to a finite list of tokens and to distribute this density among the tokens of the list.

Our work builds upon previous work by some of the authors [7, 8, 13–15]. We have essentially followed the same lines and in particular have used de Boer and Palamidessi's notion of modular embedding to compare the families of sublanguages of Dense Bach and Dense Bach with Distributed Density. Accordingly, we have established a gain of expressivity, namely that Dense Bach with Distributed Density is strictly more expressive than Dense Bach and, consequently, in view of the results of [12], strictly more expressive than the Bach and Linda languages.

Our work has similarities but also differences with several work on the expressiveness of Linda-like languages. Compared to [19] and [20], it is worth observing that a different comparison criteria is used to compare the expressiveness of languages. Indeed, in these pieces of work, the comparison is performed on (i) the compositionality of the encoding with respect to parallel composition, (ii) the preservation of divergence and deadlock, and (iii) a symmetry condition. Moreover, as will be observed by the careful reader, we have taken a more liberal view 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 $\sigma \cup \sigma$ (where $\cup$ denotes multi-set union) if this is so for the stores resulting from the agents themselves.

In [2], nine variants of the $\mathscr{L}$(ask,nask,get,tell) language are studied. They are obtained by varying both the nature of the shared data space and its structure. Rephrased in the setting of [1], this amounts to considering different operational semantics. In contrast, in our work we fix an operational semantics and compare different languages on the basis of this semantics. In [11], a process algebraic treatment of a family of Linda-like concurrent languages is presented. Again, different semantics are considered whereas we have sticked to one semantics and have compared languages on this basis.

In [10], a study of the absolute expressive power of different variants of Linda-like languages has been made, whereas we study the relative expressive power of different variants of such languages (using modular embedding as a yard-stick and the ordered interpretation of tell).

It is worth observing that [2, 10, 11, 19, 20] do not deal with a notion of density attached to tuples. In contrast, [3] and [4] decorate tuples with an extra field in order to investigate how probabilities and priorities can be introduced in the Linda coordination model. Different expressiveness results are established in [3] but on an absolute level with respect to Turing expressiveness and the possibility to encode

the Leader Election Problem. Our work contrasts in several aspects. First, we have established relative expressiveness results by comparing the sublanguages of two families. Moreover, some of these sublanguages incorporate the *nask* primitives, which strictly increases the expressiveness. Finally, the introduction of density resembles but is not identical to the association of weights to tuples. Indeed, in contrast to [3, 4] we do not modify the tuples on the store and do not modify the matching function so as to retrieve the tuple with the highest weight. In contrast, we modify the tuple primitives so as to be able to atomically put several occurrences of a tuple on the store and check for the presence or absence of a number of occurrences. We have also introduced a distribution of a density among the tokens of a set, which results in adding a new non-deterministic behavior to the tell, ask and get primitives. As can be appreciated by the reader through the comparison of Bach, Dense Bach and Dense Bach with Distributed Density, this facility of handling atomically several occurrences produces a real increase of expressiveness. One may however naturally think of encoding the number of occurrences of a tuple as an additional weight-like parameter. It is nevertheless not clear how our primitives tackling at once several occurrences can be rephrased in Linda-like primitives and how the induced encoding would still fulfills the requirements of modularity. Moreover, in contrast to Linda-like language, due to the non-determinism of the get and tell primitives, it is not clear how to code ask primitives by get and tell ones in our distributed density framework. This will be the subject for future research.

In [18], Viroli and Casadei propose a stochastic extension of the Linda framework, with a notion of tuple concentration, similar to the weight of [3] and [4] and our notion of density. The syntax of this tuple space is modeled by means of a calculus, with an operational semantics given as an hybrid CTMC/DTMC model. This operational semantics describes the behavior of tell, ask and get like primitives but does not consider a nask like primitive. Moreover, no expressiveness results are established and there is no counterpart for non-determinism arising from the distribution of density on tokens.

These three last pieces of work tackle probabilistic extensions of Linda-like languages. As a further and natural step in our research, we aim at studying how our notion of density can be the basis of such probabilistic extensions. As our work also relies on the possibility to atomically put several occurrences of tokens and test for their presence or absence, we will also examine in future work how Dense Bach with Distributed Density compares with the Gamma language.

References

[1] F.S. de Boer & C. Palamidessi (1994): *Embedding as a Tool for Language Comparison*. *Information and Computation* 108(1), pp. 128–157, doi:10.1006/inco.1994.1004.

[2] Marcello M. Bonsangue, Joost N. Kok & Gianluigi Zavattaro (1999): *Comparing coordination models based on shared distributed replicated data*. In: *ACM Symposium on Applied Computing*, pp. 156–165, doi:10.1145/298151.298226.

[3] M. Bravetti, R. Gorrieri, R. Lucchi & G.Zavattaro (2005): *Quantitative Information in the Tuple Space Coordination Model*. *Theoretical Computer Science* 346(1), pp. 28–57, doi:10.1016/j.tcs.2005.08.004.

[4] M. Bravetti, R. Gorrieri, R. Lucchi & G. Zavattaro (2004): *Probabilistic and Prioritized Data Retrieval in the Linda Coordination Model*. In R. De Nicola, G.L. Ferrari & G. Meredith, editors: *Proceedings of the 6th International Conference on Coordination Models and Languages, Lecture Notes in Computer Science* 2949, Springer, pp. 55–70, doi:10.1007/978-3-540-24634-3_7.

[5] A. Brogi & J.-M. Jacquet (1997): *Modeling Coordination via Asynchronous Communication*. In D. Garlan & D. Le Métayer, editors: *Proceedings of the Second International Conference on Coordination Lan-*

guages and Models, Lecture Notes in Computer Science 1282, Berlin, Germany, pp. 238–255, doi:10.1007/3-540-63383-9_84.

[6] A. Brogi & J.-M. Jacquet (1998): *On the Expressiveness of Linda-like Concurrent Languages*. *Electronic Notes in Theoretical Computer Science* 16(2), pp. 61–82, doi:10.1016/S1571-0661(04)00118-5.

[7] A. Brogi & J.-M. Jacquet (1999): *On the Expressiveness of Coordination Models*. In C. Ciancarini & A. Wolf, editors: *Proceedings of the Third International Conference on Coordination Languages and Models, Lecture Notes in Computer Science* 1594, Springer-Verlag, pp. 134–149, doi:10.1007/3-540-48919-3_11.

[8] A. Brogi & J.-M. Jacquet (2003): *On the Expressiveness of Coordination via Shared Dataspaces*. *Science of Computer Programming* 46(1–2), pp. 71 – 98, doi:10.1016/S0167-6423(02)00087-4.

[9] A. Brogi, J.-M. Jacquet & I. Linden (2003): *On Modeling Coordination via Asynchronous Communication and Enhanced Matching*. *Electronic Notes in Theoretical Computer Science* 68(3), doi:10.1016/S1571-0661(05)82568-X.

[10] Nadia Busi, Roberto Gorrieri & Gianluigi Zavattaro (1997): *On the Turing equivalence of Linda coordination primitives*. *Electronic Notes in Theoretical Computer Science* 7, pp. 75–75, doi:10.1016/S1571-0661(05)80467-0.

[11] Nadia Busi, Roberto Gorrieri & Gianluigi Zavattaro (1998): *A Process Algebraic View of Linda Coordination Primitives*. *Theoretical Computer Science* 192, pp. 167–199, doi:10.1016/S0304-3975(97)00149-7.

[12] J.-M. Jacquet, I. Linden & D. Darquennes (2013): *On Density in Coordination Languages*. In C. Canal & M. Villari, editors: *Proceedings of the European Conference on Service Oriented and Cloud Computing 2013, Communications in Computer and Information Science* 393, Springer, pp. 189–203, doi:10.1007/978-3-642-45364-9_16.

[13] I. Linden & J.-M. Jacquet (2004): *On the Expressiveness of Absolute-Time Coordination Languages*. In R. De Nicola, G.L. Ferrari & G. Meredith, editors: *Proc. 6th International Conference on Coordination Models and Languages*, *Lecture Notes in Computer Science* 2949, Springer, pp. 232–247, doi:10.1007/978-3-540-24634-3_18.

[14] I. Linden & J.-M. Jacquet (2007): *On the Expressiveness of Timed Coordination via Shared Dataspaces*. *Electronical Notes in Theoretical Computer Science* 180(2), pp. 71–89, doi:10.1016/j.entcs.2006.10.047.

[15] I. Linden, J.-M. Jacquet, K. De Bosschere & A. Brogi (2004): *On the Expressiveness of Relative-Timed Coordination Models*. *Electronical Notes in Theoretical Computer Science* 97, pp. 125–153, doi:10.1016/j.entcs.2004.04.034.

[16] I. Linden, J.-M. Jacquet, K. De Bosschere & A. Brogi (2006): *On the Expressiveness of Timed Coordination Models*. *Science of Computer Programming* 61(2), pp. 152–187, doi:10.1016/j.scico.2005.10.011.

[17] E.Y. Shapiro (1992): *Embeddings Among Concurrent Programming Languages*. In W.R. Cleaveland, editor: *Proceedings of Concur 1992*, Lecture Notes in Computer Science, Springer, pp. 486–503, doi:10.1007/BFb0084811.

[18] M. Viroli & M. Casadei (2009): *Biochemical Tuple Spaces for Self-organising Coordination*. In J. Field & V. T. Vasconcelos, editors: *Proceedings of 11th International Conference on Coordination Models and Languages*, *Lecture Notes in Computer Science* 5521, Springer, pp. 143–162, doi:10.1007/978-3-642-02053-7_8.

[19] G. Zavattaro (1998): *On the incomparability of Gamma and Linda*. *Electronic Transactions on Numerical Analysis.*

[20] Gianluigi Zavattaro (1998): *Towards a Hierarchy of Negative Test Operators for Generative Communication*. *Electronic Notes in Theoretical Computer Science* 16, pp. 154–170, doi:10.1016/S1571-0661(04)00125-2.