On Multiplicities in Tuple-Based Coordination Languages: The Bach Family of Languages and Its Expressiveness Study

Denis Darquennes[1] , Jean-Marie Jacquet[1(✉)] , and Isabelle Linden[2]

[1] Faculty of Computer Science, University of Namur,
Rue Grandgagnage 21, 5000 Namur, Belgium
{denis.darquennes,jean-marie.jacquet}@unamur.be
[2] Business Administration Department, University of Namur,
Rempart de la Vierge 8, 5000 Namur, Belgium
isabelle.linden@unamur.be

Abstract. Building upon previous work by the authors, this paper reviews and proposes extensions of Linda-like languages aiming at coordinating data-intensive distributed systems. The languages manipulate tokens associated in different ways with a notion of multiplicity. Thanks to De Boer and Palamidessi's notion of modular embedding, we establish expressiveness hierarchies. We also discuss implementation issues and argue that the more expressive the language is the more expensive is its implementation.

1 Introduction

Technological evolutions over the last recent years have confirmed the upward trends in pervading our everyday environment by more and more numerical artifacts, mobile or not, injecting or retrieving an endless increasing amount of information. As a result, service-oriented applications have become more and more necessary and indeed have been developed at an increasing speed. Most of them are based on popularity and quality measures, with, as a key feature, the fact that these meseasures are not determined at specific points in time but rather continuously, as user experiences evolve.

Moreover, with the help of machine learning techniques, data tend to be more and more transformed in knowledge, which leads our daily life to be more and more mediated by knowledge systems. In this context coordinating systems relying on a huge amount of data appears to be a central task. Coordination languages and models have proved to be well suited to program the interaction of conventional distributed systems, and in particular to model service-oriented applications (see e.g. [7,23,31]). Recently, it has been shown in [28] how they can be used to code complex socio-technological systems based on the interaction of

knowledge intensive components. This paper aims at addressing a more fundamental issue in exploring how the addition of multiplicity information to tuples increases the expressiveness of Linda [18], the seminal coordination language, while being able to handle the above mentioned requirement of popularity and quality measures.

To do so, we shall start with a dialect of Linda developed at the University of Namur, named Bach. Following Linda, it 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 a non-deterministic fashion, a first extension was proposed in [20] in the aim of enriching traditional data-based coordination languages by a notion of multiplicity (historically named density) attached to tuples, thereby yielding a new coordination language, called Dense Bach. In a second extension we have proposed in [16] to consider lists of tuples among which densities are distributed. The resulting language has been named DBD-Bach. It turns out that its presentation can be made more elegant by using a variant, named VD-Bach, in which arguments of coordination primitives are composed of lists of so-called dense tokens.

Introducing variants of languages necessarily calls for a gain of expressiveness. Based on previous work by the authors, among others of [8,11,12,16,20,27], we shall employ de Boer and Palamidessi's modular embedding and show that Bach is less expressive than Dense Bach, which itself is less expressive than VD-Bach. VD-Bach being similar in essence to multiset rewriting, as introduced in Gamma [1,2], we shall also compare the two languages and prove that Gamma is actually more expressive than VD-Bach.

Since our purposes are essentially of a theoretical nature, for simplicity purposes, we shall consider in this paper simplified versions of the languages where tuples are taken in their simplest form of flat and unstructured tokens. Nevertheless, as we shall argue at the end of the paper, the resulting simplification of the matching process is orthogonal to our purposes and, consequently, our results can be directly extended to more general tuples. Consequently, our languages will subsequently be renamed with a T suffix, thus yielding BachT, DBD-BachT and VD-BachT.

Despite this simplification, we shall also discuss implementation issues and, show, without big surprise, that the more expressive a language is the more expensive is its implementation. This highlights from another perspective the expressiveness results: instead of directly using the more expressive language, it is of interest from an efficiency point of view to use the language just expressive enough for coding purposes under consideration.

The rest of the paper is organized as follows. Section 2 presents the languages studied in the paper and defines an operational semantics. Section 3 evidences the interest of these languages through the coding of some examples but also by showing how the newly introduced language VD-BachT can express the language DBD-Bach introduced in [16]. Section 4 provides a short presentation of

modular embedding and, on that basis, proceeds with an exhaustive comparison of the relative expressive power of the languages. Section 5 shows how these expressiveness results can be lifted to tuple-based languages. Section 6 discusses implementation issues. Finally, Sect. 7 compares our work with related work, draws our conclusions and presents expectations for future work.

2 Densed Tuple-Based Coordination Languages

2.1 Primitives

A. BachT and Dense BachT

Let us start by defining the BachT and Dense BachT languages [20] from which the languages under study in this paper are extensions. The following definition formalizes how we attach a multiplicity or 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 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 $\mathcal{T}_b$ of the token-based primitives as the set of primitives T_b generated by the following grammar:*

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

where t represents a token. Similarly, define the set of dense token-based primitives $\mathcal{T}_{db}$ as the set of primitives T_{db} generated by the following grammar:

$$T_{db} ::= tell(t(m)) \mid ask(t(m)) \mid get(t(m)) \mid nask(t(m))$$

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

The primitives of the BachT 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 $tell(t)$ primitive amounts to enriching the store by an occurrence of t. The $ask(t)$ and $get(t)$ primitives check whether t is present on the store with

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

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

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

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

Fig. 1. Transition rules for token-based primitives (BachT)

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

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

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

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

Fig. 2. Transition rules for dense token-based primitives (Dense BachT)

the latter removing one occurrence. Dually, $nask(t)$ tests whether t is absent from the store.

The primitives of the Dense BachT 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 Figs. 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 BachT are those of Dense BachT with a density of 1. Consequently, our explanation starts by the more general rules of Fig. 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 multiset 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 is sure that t does not occur in σ and consequently that there

$(\mathbf{T_v})$
$$\frac{m_1, \cdots, m_n \in \mathbb{N}_0}{\langle\, tell(t_1(m_1), \cdots, t_n(m_n)) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t_1(m_1), \cdots, t_n(m_n)\} \,\rangle}$$

$(\mathbf{A_v})$
$$\frac{m_1, \cdots, m_n \in \mathbb{N}_0}{\langle\, ask(t_1(m_1), \cdots, t_n(m_n)) \mid \sigma \cup \{t_1(m_1), \cdots, t_n(m_n)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t_1(m_1), \cdots, t_n(m_n)\} \,\rangle}$$

$(\mathbf{G_v})$
$$\frac{m_1, \cdots, m_n \in \mathbb{N}_0}{\langle\, get(t_1(m_1), \cdots, t_n(m_n)) \mid \sigma \cup \{t_1(m_1), \cdots, t_n(m_n)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$(\mathbf{N_v})$
$$\frac{m_1, \cdots, m_n \in \mathbb{N}_0, p_1 < m_1, \cdots, p_n < m_n}{\langle\, nask(t_1(m_1), \cdots, t_n(m_n)) \mid \sigma \uplus \{t_1(p_1), \cdots, t_n(p_n)\} \,\rangle \longrightarrow \langle\, E \mid \sigma \uplus \{t_1(p_1), \cdots, t_n(p_n)\} \,\rangle}$$

Fig. 3. Transition rules for vectorized dense token-based primitives (VD-BachT)

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 BachT language. As expected, they amount to the rules of Fig. 2 where the density m is taken to be 1.

B. Vectorized Dense BachT

A natural extension is to replace a dense token by a set of dense tokens in the primitives. For instance, the primitive $ask(t(1), u(2), v(3))$ would succeed on a store containing one occurrence of t, two of u and three of v. Dually, the computation of $tell(t(1), u(2), v(3))$ would result in adding one occurrence of t on the store, two of u and three of v.

The following definitions formalize this intuition. As seen above, to avoid using unnecessary brackets, we shall slightly abuse notations and use lists of dense tokens, which we shall subsequently designate as vectors of dense tokens, hence the name Vectorized Dense BachT or VD-BachT for short. The intuition remains however that of sets, with the order of the dense tokens being meaningless.

Definition 3. *Define a vector of dense tokens as a list $t_1(m_1), \cdots, t_n(m_n)$ of dense tokens. Such a vector is subsequently denoted as $\overrightarrow{t(m)}$. Define SVDtoken as the set of vectors of dense tokens.*

Definition 4. *Define the set of vectorized dense token-based primitives $\mathcal{T}_{vb}$ as the set of primitives T_{vb} generated by the following grammar:*

$$T_{vb} ::= tell(\overrightarrow{t(m)}) \mid ask(\overrightarrow{t(m)}) \mid get(\overrightarrow{t(m)}) \mid nask(\overrightarrow{t(m)})$$

where $\overrightarrow{t(m)}$ represents a vector of dense tokens.

The transition steps for these primitives are defined in Fig. 3. As suggested above, rule (T_v) asserts that telling a vector of dense tokens amounts to adding

$$(\mathbf{W_v}) \quad \frac{m_1, \cdots, m_n \in \mathbb{N}_0, \{t_1(p_1), \cdots, t_n(p_n)\} \nsubseteq \sigma}{\langle\, wnask(t_1(m_1), \cdots, t_n(m_n)) \mid \sigma\,\rangle \longrightarrow \langle\, E \mid \sigma\,\rangle}$$

Fig. 4. Transition rule for the weak nask

each of them with the corresponding density on the store. Similarly, rule (A_v) requires for an ask primitive to succeed the presence, for each token t_i, of at least m_i occurrences on the store. According to rule (G_v) the behavior of a get primitive performs such a test for presence but also removes m_i occurrences of t_i on the store. Finally, rule (N_v) requires, for each token t_i, the absence of m_i occurrences. It is here worth noting that, in contrast to BachT and Dense BachT, the behavior of the nask primitive is not the negation of that of the ask primitive. Indeed, this interpretation would have required for the nask primitive that, for some token t_i, less than m_i occurrences are present on the store. It will however be handful to have such a nask primitive. We thus introduce it, name it weak nask and denote it by wnask. It is formally defined by rule (W_w) of Fig. 4.

It is worth observing that with such a definition, $wnask(\overrightarrow{t(m)})$ succeeds whenever $nask(\overrightarrow{t(m)})$ succeeds. However, the converse is not true. Consider, for instance, the store composed of 2 occurrences of t and 4 of u. In that context, $nask(t(1), u(5))$ does not succeed since, although $4 < 5$ the inequality $2 < 1$ does not hold. However, $wnask(t(1), u(5))$ succeeds since, as multisets, $\{t(2), u(4)\} \nsubseteq \{t(1), u(5)\}$. Rephrased using the notation of rule (N_v), it is required for $nask$ that $p_1 < m_1 \wedge \cdots \wedge p_n < m_n$ whereas $wnask$ only requires that $p_1 < m_1 \vee \cdots \vee p_n < m_n$. In view of that, it is easy to verify that $wnask(t_1(m_1), \cdots, t_n(m_n))$ can be encoded as follows: $nask(t_1(m_1)) + \cdots + nask(t_n(m_n))$ where $+$ denotes the non-deterministic choice. As a result, although useful later, $wnask$ does not bring an increase of expressiveness.

C. MRT

The last language we shall consider is a Gamma-like language [1,2], based on the chemical reaction metaphor. It considers communication primitives as the rewriting of pre-condition multi-sets into post-condition multi-sets. Intuitively, the operational effect of a multi-set rewriting (*pre*, *post*) consists in inserting all the positive post-conditions, and in deleting all the negative post-conditions from the current store σ, provided that σ contains all positive pre-conditions and does not meet any of the negative pre-conditions. Formally, these rewritings are specified as follows.

Definition 5. *Define the set of multi-set rewriting primitives $\mathcal{T}_{MR}$ as the set of primitives T_{MR} generated by the following grammar:*

$$T_{MR} ::= (\{M\}, \{M\})$$
$$M ::= \lambda \mid +t \mid -t \mid M, M$$

where λ indicates an empty multi-set and where t denotes a token.

$$(CM) \qquad \frac{pre^+ \subseteq \sigma, \ pre^- \perp \sigma,}{\langle (pre, post) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma' \rangle}$$

Fig. 5. Transition rules for multi-set rewriting-based primitives (MRT)

It is worth observing that not all pairs of preconditions and postconditions correspond to reasonable computations. Indeed, as stated above, it is possible to require in a precondition that the same token is present and absent or to require in the postcondition the removal of a token which has not been tested for presence in the precondition. We subsequently define such reasonable pairs of pre- and post-conditions as respectively consistent and valid. To that end, we first introduce some notations.

Definition 6. *Given a multi-set rewriting pair $(Pre, Post)$, denote by Pre^+ the multi-set $\{t \mid + t \in Pre\}$ of tokens positively appearing in the precondition and by Pre^- the multi-set $\{t \mid - t \in Pre\}$ negatively appearing in it. Similarly, we shall denote by $Post^+$ and $Post^-$ the multiset of tokens appearing positively and negatively in the postcondition.*

A multi-set rewriting pair (Pre, Post) is said to be consistent if $Pre^+ \cap Pre^- = \varnothing$. It is said to be valid if $Post^- \subseteq Pre^+$.

A consequence of consistency and validity is that four basic pairs of pre- and post-conditions can be put forward: $(\{+t\}, \{\})$, $(\{-t\}, \{\})$, $(\{\}, \{+t\})$, $(\{+t\}, \{-t\})$. They correspond respectively to the *ask(t)*, *nask(t)*, *tell(t)* and *get(t)* of the BachT language.

It turns out that it is possible to define it by one rule. To express it, an auxiliary notion is however needed. It extends the notations of Definition 6 to capture the fact that, for each token, the tokens mentioned negatively in the definition are not with their multiplicity on the current store σ.

Definition 7. *For any token t, define $Pre^-[t]$ as the multiset of negatively marked tokens t in the precondition Pre:*

$$Pre^-[t] = \{t : -t \in Pre^-\}.$$

Given a precondition Pre and a store σ, we then define the non element-wise inclusion operator $\perp$ as follows:

$$Pre^- \perp \sigma \ \text{iff} \ Pre^-[t] \nsubseteq \sigma, \text{for any token t.}$$

With this notation, rule (CM) of Fig. 5 states that a multi-set rewriting $(Pre, Post)$ can be executed in a store σ if the multi-set Pre^+ is included in σ and if no negative pre-condition occurs with the required multiplicity in σ. Under these conditions, the effect of the rewriting is to delete from σ all the negative post-conditions and to add to σ all the positive post-conditions.

2.2   Languages

We are now in a position to formally define the languages we shall consider in the paper. 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 non-deterministic choice operator $+$, parallel operator (denoted by the $||$ symbol) and the sequential operator (denoted by the ; symbol). The formal definition is as follows.

Definition 8. *Define the BachT language $\mathcal{L}_B$ as the set of agents A generated by the following grammar:*

$$A ::= T_b \mid A\,;\,A \mid A \parallel A \mid A\,+\,A$$

where T_b represents a token-based primitive. Define similarly the Dense BachT language $\mathcal{L}_{DB}$, the VD-BachT language $\mathcal{L}_{VB}$, the MRT language $\mathcal{L}_{MR}$ by taking instead of the token-based primitive T_b, respectively the dense token-based primitives T_{db}, the list of token-based primitive T_{vb} and the multi-set rewriting primitive T_{MR}.

Moreover, subsequently, we shall consider sublanguages formed similarly but by considering only subsets of these primitives. In that case, if $\mathcal{H}$ denotes such a subset, then we shall write the induced sublanguages as $\mathcal{L}_B(\mathcal{H})$, $\mathcal{L}_{DB}(\mathcal{H})$, $\mathcal{L}_{VB}(\mathcal{H})$ and $\mathcal{L}_{MR}(\mathcal{H})$ 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 $(E; A)$, $(E \parallel A)$ and $(A \parallel E)$ as A. This is technically achieved by defining the extended sets of agents as $\mathcal{L}_B \cup \{E\}$, $\mathcal{L}_{DB} \cup \{E\}$, $\mathcal{L}_{VB} \cup \{E\}$ or $\mathcal{L}_{MR} \cup \{E\}$ and by justifying the simplifications by imposing a bimonoid structure.

The rules for the primitives of the languages have been given in Figs. 1, 2, 3, 4 and 5. Figure 6 details the usual rules for sequential composition, parallel composition, interpreted in an interleaving fashion, and non-deterministic 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 e.g. [10,11,24–26]),

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

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

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

Fig. 6. Transition rules for the operators

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).

Definition 9.

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 $\mathcal{L}_I$ of the languages $\mathcal{L}_B$, $\mathcal{L}_{DB}$, $\mathcal{L}_{VB}$, $\mathcal{L}_{MR}$, define the operational semantics $\mathcal{O}_I : \mathcal{L}_I \to \mathcal{P}(Shist)$ as the following function: for any agent $A \in \mathcal{L}$*

$$\mathcal{O}(A) = \{(\sigma, \delta^+) : \langle A|\varnothing \rangle \to^* \langle E|\sigma \rangle\}$$
$$\cup \{(\sigma, \delta^-) : \langle A|\varnothing \rangle \to^* \langle B|\sigma \rangle \nrightarrow, B \neq E\}$$

3 Applications

To evidence the interest of Dense BachT and VD-BachT, let us turn to some applications and see how they easily allow for their encodings.

3.1 A Simple Taxi Application

A typical service application inspired by Uber consists in a system allowing to select taxi drivers based on their reputation. To be operational, such a system needs on the one hand to allow users to express their satisfaction with regard to the service provided, and on the other hand, to test that a taxi driver is recognized at a sufficient level of satisfaction. For illustration purposes, we will assume that only positive marks are taken into account and that the service offered by a taxi driver can be evaluated as good or excellent, corresponding to a respective evaluation with numbers 1 and 2. We will then imagine that a level of satisfaction 100 is a minimal satisfaction mark for a reasonable driver.

Using Dense BachT for the first task, the satisfaction of a user can be registered by inserting the token `taxi_driver_id` once if the evaluation mark is good and twice if it is excellent. Technically, with `taxi_driver_id` being the identifier of the taxi driver, this amounts to respectively executing `tell(taxi_driver_id(1))` or `tell(taxi_driver_id(2))`. As regards the second task, making sure that a proposed driver, say identified by `id`, has reached a level of satisfaction of at least 100, can be simulated by executing the primitive `ask(id(100))`. Note that, as the number of matching tuples is only counted, such a satisfaction level may be reached thanks to the contribution of many users. Of course, different policies can be implemented in the application, for instance to forbid a user to mark a taxi driver more than once a day. It is also worth noting that thanks to the space and time decoupling between information producers and information consumers offered by coordination languages, it is very easy to introduce new users and new taxi drivers in the application.

3.2   The Dining Philosophers

Formulated by Edsger Dijkstra in 1965, the dining philosophers is a classical concurrency problem addressing the synchronisation of processes sharing resources. It is formulated as follows: N philosophers spend their time thinking and eating. To eat, they must sit on a round table in front of a dish and take the forks on their left and right sides. There is however only one fork between two dishes, which makes it impossible for all the philosophers to eat simultaneously.

The classical solution is to use semaphores, one for each fork and one to let only enter to the table a number of philosophers. Other solutions have been proposed by the coordination community, for instance, using Respect [29]. The Vector Dense BachT language proposes a very simple solution by associating each fork with a token and by simulating each philosopher taking his two forks by a get primitive on these tokens and dually each philosopher realising them by means of a tell primitive. For $N = 5$, the philosophers may then be coded as follows:

$$Phil_0 = get(f_0, f_1); tell(f_0, f_1); Phil_0$$
$$Phil_1 = get(f_1, f_2); tell(f_1, f_2); Phil_1$$
$$\dots$$
$$Phil_4 = get(f_4, f_0); tell(f_4, f_0); Phil_4$$

with the whole set of philosophers simulated by

$$Phil_0 \parallel Phil_1 \parallel Phil_2 \parallel Phil_3 \parallel Phil_4$$

3.3   An Online Shopping System

Let us now consider an online shopping system related to an European sporting goods store, present in five different European cities: Brussels, Paris, London, Berlin and Rome. All these shops propose the same articles. In order to manage

efficiently the number of orders that arrive through the online system, these are distributed on the different shops present in the five cities. Assume that a group of 50 orders arrive and has to be distributed equally between the different shops. This can be simulated through the execution of the following tell primitive:

```
tell(Brussels(10), Paris(10), London(10), Berlin(10), Rome(10)).
```

Assume now that the following maxima of orders to be processed have been imposed for the shops: 200 orders for Brussels, 75 for Paris, 50 for London, 150 for Berlin and 70 for Rome. A check whether these maxima have not been reached can be simulated by executing the following nask primitive:

```
nask(Brussels(200), Paris(75), London(50), Berlin(150), Rome(70)).
```

3.4   Distributed Density

In the online shopping problem, the arrival of 50 orders has been explicitly distributed on the shops. A natural extension is to let the execution of the primitive non-deterministically choose the distribution. We are then lead to consider a list of tokens together with a density and to distribute it on the tokens. The following definition formalizes such an association.

Definition 10. *Let Snlt denote the set of non-empty lists of tokens in which, for simplicity purposes, each token differs from the others. Such a list 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 list of tokens as a list of Snlt associated with a strictly positive integer. Such a dense list is typically represented as $L(m)$, with L the list of tokens and m an integer.*

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

Definition 11. *Define the distribution of tokens from dense lists of tokens to sets of tuples of dense tokens as follows:*

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

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 $\mathcal{D}i([t_1, \cdots, t_p](m))$ the distribution of m over $[t_1, \cdots, t_p]$.

The distribution of an integer m over a list of tokens L has the potential to express the behavior of the BachT primitives extended with dense lists of tokens as arguments. Indeed, telling a dense list amounts to telling atomically the $t_i[m_i]$'s of a tuple defined above. Asking or getting a dense list requires to check that a tuple of $\mathcal{D}i([t_1, \cdots, t_p](m))$ 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.

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

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

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

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

Fig. 7. Transition rules for list of token-based primitives (Dense BachT with distributed Density)

Definition 12. *Let m be a positive integer, $L = [t_1, \cdots, t_p]$ be a list of tokens and σ a store. We define $\mathcal{D}i(L(m)) \sqcap \sigma$ as the following set of tuples of dense tokens:*

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

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

Definition 13. *Define the set of dense lists primitives $\mathcal{T}_{dbd}$ as the set of primitives T_{dbd} generated by the following grammar:*

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

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

The transition steps for these primitives are defined in Fig. 7. As suggested above, rule (T_{dbd}) specifies that telling a dense list $L(m)$ of tokens amounts to atomically adding 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 BachT and Vectorized Dense BachT. Rule (A_{dbd}) states that asking for the dense list $L(m)$ 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.

We are now in a position to define the language Dense BachT with a Distribution of the density over a list of tokens by considering the statements of this language as defined from the tell, ask, get and nask primitives possibly combined by the non deterministic choice, parallel and sequential operators.

A further extension consists in equipping the tokens of a dense list with minimal and maximal numbers, as follows.

Definition 14. *Define the association of a token and two positive integers of $\mathbb{N}$ as a capacity dense token. Such a token is typically denoted as $t(m, n)$ where t is the token and m, n are the integers.*

Definition 15. *Let Snlct denote the set of non-empty lists of capacity dense tokens in which, for simplicity purposes, each token differs from the others. Such a list is typically denoted as $L = [t_1(m_1, n_1), \ldots, t_p(m_p, n_p)]$ and is thus such that $t_i \neq t_j$ for $i \neq j$. Define a dense list of capacity dense tokens as a list of Snlct associated with a strictly positive integer. Such a list is typically represented as $L(m)$, with L the list of capacity dense tokens and m an integer.*

The expected extended language is simply obtained by slightly modifying the notion of distribution introduced in Definition 11.

Definition 16. *Define the cardinality based distribution of tokens from dense lists of capacity tokens to sets of tuples of extended dense tokens as follows:*

$$\mathcal{D}c([t_1(m_1, n_1), \cdots, t_p(m_p, n_p)](q))$$
$$= \{(t_1(q_1), \cdots, t_p(q_p)) : q_1 + \cdots + q_p = q \text{ and } m_i \leq q_i \leq n_i \text{ for } i \in \{1, \cdots, p\}\}$$

Note that nothing guarantees that the above set is non empty. We shall subsequently called *coherent* those dense lists of capacity based tokens such that their cardinality based distribution is non empty and restrict ourselves to such coherent dense lists in the following.

To conclude this section, it is worth observing that it is straightforward to translate the positive version of DBD-BachT and its cardinality extension in terms of VD-BachT.

Indeed, as easily observed, one can code the tell, ask and get primitives of DBD-BachT as follows:

$$tell(L(m)) = \sum_{\boldsymbol{v} \in \mathcal{D}i(L(m))} tell(\boldsymbol{v}^{\,r})$$

$$ask(L(m)) = \sum_{\boldsymbol{v} \in \mathcal{D}i(L(m))} ask(\boldsymbol{v}^{\,r})$$

$$get(L(m)) = \sum_{\boldsymbol{v} \in \mathcal{D}i(L(m))} get(\boldsymbol{v}^{\,r})$$

where $\boldsymbol{v}^{\,r}$ denotes the vector $\boldsymbol{v}$ restricted to its strictly positive dense tokens.

Translating the nask primitive is slightly more complicated in requiring the parallel composition of the weak form of nask of vectors:

$$nask(L(m)) = ||_{\boldsymbol{v} \in \mathcal{D}i(L(m))} wnask(\boldsymbol{v}^{\,r})$$

Translating the DBD-BachT language with cardinality proceeds similarly by using $\mathcal{D}c(L(m))$ instead of $\mathcal{D}i(L(m))$.

4   Expressiveness Study

The translation just introduced evidences the need for a study of the expressiveness power of the languages introduced in Sect. 2, to which we now turn.

4.1   On the Expressiveness of Languages

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.
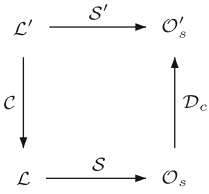
Fig. 8. Basic embedding.

According to this intuition, Shapiro introduced in [30] a first notion of embedding as follows. Consider two languages $\mathcal{L}$ and $\mathcal{L}'$. Assume given the semantics mappings (*Observation criteria*) $\mathcal{S} : \mathcal{L} \to \mathcal{O}_s$ and $\mathcal{S}' : \mathcal{L}' \to \mathcal{O}'_s$, where $\mathcal{O}_s$ and $\mathcal{O}'_s$ are on some suitable domains. Then $\mathcal{L}$ can *embed* $\mathcal{L}'$ if there exists a mapping $\mathcal{C}$ (coder) from the statements of $\mathcal{L}'$ to the statements of $\mathcal{L}$, and a mapping $\mathcal{D}_c$ (decoder) from $\mathcal{O}_s$ to $\mathcal{O}'_s$, such that the diagram of Fig. 8 commutes, namely such that for every statement $A \in \mathcal{L}' : \mathcal{D}_c(\mathcal{S}(\mathcal{C}(A))) = \mathcal{S}'(A)$.

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 [17] to add three constraints on the coder $\mathcal{C}$ and on the decoder $\mathcal{D}_c$ in order to obtain a notion of *modular* embedding usable for concurrent languages:

1. $\mathcal{D}_c$ should be defined in an element-wise way with respect to $\mathcal{O}_s$, namely for some appropriate mapping $\mathcal{D}_{el}$

$$\forall X \in \mathcal{O}_s : \ \mathcal{D}_c(X) = \{\mathcal{D}_{el}(x) \mid x \in X\} \tag{P_1}$$
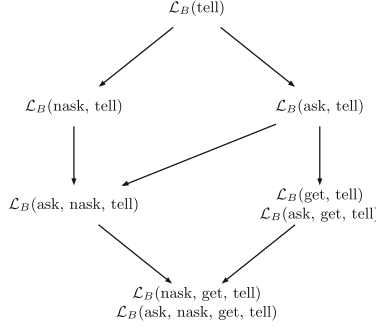
2. the coder $\mathcal{C}$ should be defined in a compositional way with respect to the sequential, parallel and choice operators:

$$\begin{aligned} \mathcal{C}(A \ ; \ B) &= \mathcal{C}(A) \ ; \ \mathcal{C}(B) \\ \mathcal{C}(A \parallel B) &= \mathcal{C}(A) \parallel \mathcal{C}(B) \\ \mathcal{C}(A \ + \ B) &= \mathcal{C}(A) \ + \ \mathcal{C}(B) \end{aligned} \tag{P_2}$$
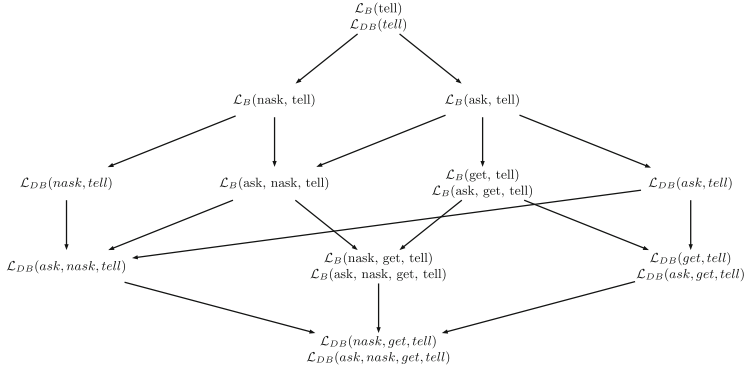
3. the embedding should preserve the behavior of the original processes with respect to deadlock, failure and success (*termination invariance*):

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

where *tm* and *tm'* extract the termination information from the observables of $\mathcal{L}$ and $\mathcal{L}'$, respectively.

$\mathcal{L}_B(\text{tell})$

$\mathcal{L}_B(\text{nask, tell})$ $\mathcal{L}_B(\text{ask, tell})$

$\mathcal{L}_B(\text{ask, nask, tell})$ $\mathcal{L}_B(\text{get, tell})$
$\mathcal{L}_B(\text{ask, get, tell})$

$\mathcal{L}_B(\text{nask, get, tell})$
$\mathcal{L}_B(\text{ask, nask, get, tell})$

Fig. 9. Embedding hierarchy of BachT Languages.

$\mathcal{L}_B(\text{tell})$
$\mathcal{L}_{DB}(\text{tell})$

$\mathcal{L}_B(\text{nask, tell})$ $\mathcal{L}_B(\text{ask, tell})$

$\mathcal{L}_{DB}(\text{nask, tell})$ $\mathcal{L}_B(\text{ask, nask, tell})$ $\mathcal{L}_B(\text{get, tell})$
$\mathcal{L}_B(\text{ask, get, tell})$ $\mathcal{L}_{DB}(\text{ask, tell})$

$\mathcal{L}_{DB}(\text{ask, nask, tell})$ $\mathcal{L}_B(\text{nask, get, tell})$
$\mathcal{L}_B(\text{ask, nask, get, tell})$ $\mathcal{L}_{DB}(\text{get, tell})$
$\mathcal{L}_{DB}(\text{ask, get, tell})$

$\mathcal{L}_{DB}(\text{nask, get, tell})$
$\mathcal{L}_{DB}(\text{ask, nask, get, tell})$

Fig. 10. Embedding hierarchy of BachT and Dense BachT

An embedding is then called *modular* if it satisfies properties P_1, P_2, and P_3. The existence of a modular embedding from $\mathcal{L}'$ into $\mathcal{L}$ is subsequently denoted by $\mathcal{L}' \leqslant \mathcal{L}$. It is easy to prove that $\leqslant$ is a pre-order relation. Moreover if $\mathcal{L}' \subseteq \mathcal{L}$ then $\mathcal{L}' \leqslant \mathcal{L}$ that is, any language embeds all its sublanguages. This property descends immediately from the definition of embedding, by setting $\mathcal{C}$ and $\mathcal{D}_c$ equal to the identity function.

4.2 Comparing BachT, Dense BachT and Vectorized Dense BachT

The expressive power of the different sublanguages of BachT has been studied in [9–11] from which the expressiveness hierarchy of Fig. 9 can be established. Building upon these results, the article [22] has established the embedding relations of Fig. 10.

In both figures, 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 $\mathcal{L}_1 \leqslant \mathcal{L}_2$. When an arrow from $\mathcal{L}_1$ to $\mathcal{L}_2$ has no counterpart from $\mathcal{L}_2$ to $\mathcal{L}_1$, then $\mathcal{L}_1$ is strictly less expressive than $\mathcal{L}_2$, that is $\mathcal{L}_1 < \mathcal{L}_2$. If $\mathcal{L}_1 \leqslant \mathcal{L}_2$ and $\mathcal{L}_2 \leqslant \mathcal{L}_1$ then $\mathcal{L}_1$ and $\mathcal{L}_2$ are equivalent, that is $\mathcal{L}_1 = \mathcal{L}_2$. In that
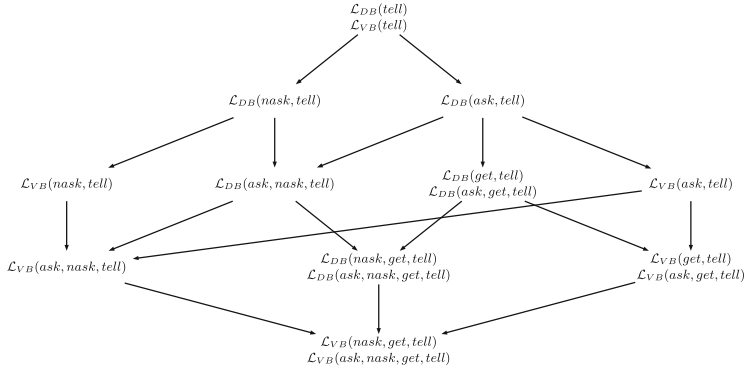
Fig. 11. Embedding hierarchy of Dense BachT and Vectorized Dense BachT languages.

case, they are depicted together. If $\mathcal{L}_1 \nleq \mathcal{L}_2$ and $\mathcal{L}_2 \nleq \mathcal{L}_1$ then $\mathcal{L}_1$ and $\mathcal{L}_2$ are not comparable with each other. This is subsequently denoted by $\mathcal{L}_1 \wr \mathcal{L}_2$. Thanks to the transitivity, both figures contain only a minimal amount of arrows. Apart from these induced relations, no other relation holds.

It is worth noting that the hierarchy relations presented in Fig. 9 appear in the center of Fig. 10. This reflects the fact that BachT is a special case of Dense BachT. Moreover, the hierarchy of the Dense BachT sublanguages resembles that of the BachT sublanguages. This intuitively results from the very nature of the ask, nask and get primitives, which are not altered by the density of tokens. Nevertheless, except for the sublanguage reduced to a tell primitive, it is worth observing that the dense sublanguages are strictly more expressive than their BachT counterparts. This highlights the fact that Dense BachT is an extension of BachT bringing more expressiveness.

Following the same reasonings as those published in [16] it is possible to establish similar embedding relations between Dense BachT and Vectorized BachT. Due to space limits, the proof are not reproduced here but the results are depicted in Fig. 11. To get a complete expressiveness picture, it remains to study the expressiveness of Vectorized Dense BachT and MRT. This is the purpose of the next subsection. Due to space limits, the key points are only given, the interested reader being referred to [15] where all the proofs are conducted in details.

4.3   Relating Vectorized Dense BachT and MRT

As a first observation, it is easy to establish that the VD-BachT sublanguages are embedded in the corresponding MRT sublanguages.

Proposition 1. $\mathcal{L}_{VB}(\chi) \leqslant \mathcal{L}_{MR}(\chi)$, for any subset of χ of primitives.

Proof. Immediate by defining the coder as follows:

$$\mathcal{C}(tell((t_1(m_1),\ldots,t_k(m_k)))) = (\{\}, \{\underbrace{+t_1,\cdots,+t_1}_{m_1\ times},\cdots,\underbrace{+t_k,\cdots,+t_k}_{m_k\ times}\})$$

$$\mathcal{C}(ask((t_1(m_1),\ldots,t_k(m_k)))) = (\{\underbrace{+t_1,\cdots,+t_1}_{m_1\ times},\cdots,\underbrace{+t_k,\cdots,+t_k}_{m_k\ times}\}, \{\})$$

$$\mathcal{C}(get((t_1(m_1),\ldots,t_k(m_k)))) = (\{\underbrace{+t_1,\cdots,+t_1}_{m_1\ times},\cdots,\underbrace{+t_k,\cdots,+t_k}_{m_k\ times}\},$$
$$\{\underbrace{-t_1,\cdots,-t_1}_{m_1\ times},\cdots,\underbrace{-t_k,\cdots,-t_k}_{m_k\ times}\})$$

$$\mathcal{C}(nask((t_1(m_1),\ldots,t_k(m_k)))) = (\{\underbrace{-t_1,\cdots,-t_1}_{m_1\ times},\cdots,\underbrace{-t_k,\cdots,-t_k}_{m_k\ times}\}, \{\})$$

and using the identity as decoder.

As for the results mentioned before, we shall only consider non trivial sublanguages, namely sublanguages containing at least the tell primitive. The store being fed with tokens, the second step is to provide the sublanguage with a possibility to question the store about the presence or the absence of tokens on it. Those two capacities result from the introduction of the ask and nask primitives. A third important property is then to allow the language to retrieve tokens from the store, by using the get primitive. Finally the last step studies the most complete language, combining the get and tell primitives with the nask and/or ask primitives.

A. Adding Tokens on the Store

When only constituted by the tell primitive the sublanguages are equivalent, namely $\mathcal{L}_{VB}(\text{tell})$ and $\mathcal{L}_{MR}(\text{tell})$ are equivalent.

Proposition 2. $\mathcal{L}_{MR}(tell)$ and $\mathcal{L}_{VB}(tell)$ are equivalent.

Proof. We have $\mathcal{L}_{VB}(\text{tell}) \leqslant \mathcal{L}_{MR}(\text{tell})$ by Proposition 1. Furthermore, $\mathcal{L}_{MR}(\text{tell}) \leqslant \mathcal{L}_{VB}(\text{tell})$ is established by coding any tell primitive of $\mathcal{L}_{MR}(\text{tell})$ as the composition of their dense versions: $\mathcal{C}(\{\}, \{\underbrace{+t_1,\cdots,+t_1}_{m_1\ times},\cdots,$

$\underbrace{+t_k,\cdots,+t_k}_{m_k\ times}\}) = tell((t_1(m_1),\cdots,t_k(m_k)))$.

B. Checking for Presence and/or Absence When Adding Tokens

In contrast to what is obtained in the comparison of Dense BachT and Vectorized BachT languages, $\mathcal{L}_{VB}(\text{ask,tell})$ is as expressive as $\mathcal{L}_{MR}(\text{ask,tell})$.

Proposition 3. $\mathcal{L}_{VB}(ask,tell) = \mathcal{L}_{MR}(ask,tell)$

Proof. **(i)** On the one hand, $\mathcal{L}_{VB}(\text{ask,tell}) \leqslant \mathcal{L}_{MR}(\text{ask,tell})$, by Proposition 1. **(ii)** On the other hand, $\mathcal{L}_{MR}(\text{ask,tell}) \leqslant \mathcal{L}_{VB}(\text{ask,tell})$ is established by noting that any agent of $\mathcal{L}_{MR}(\text{ask,tell})$ can be simulated by an agent of $\mathcal{L}_{MR}(\text{ask})$ followed by an agent of $\mathcal{L}_{MR}(\text{tell})$.

In contrast, $\mathcal{L}_{VB}(\text{nask,tell})$ is strictly less expressive than $\mathcal{L}_{MR}(\text{nask,tell})$.

Proposition 4. $\mathcal{L}_{VB}(\textit{nask,tell}) < \mathcal{L}_{MR}(\textit{nask,tell})$.

Proof. **(i)** On the one hand, $\mathcal{L}_{VB}(\text{nask,tell}) \leqslant \mathcal{L}_{MR}(\text{nask,tell})$ holds by Proposition 1. **(ii)** On the other hand, $\mathcal{L}_{MR}(\text{nask,tell}) \not\leqslant \mathcal{L}_{VB}(\text{nask,tell})$ is proved by considering agent $AB = (\{-a\}, \{+b\})$ and agent $BA = (\{-b\}, \{+a\})$, with $\mathcal{O}(AB \parallel BA) = \{(\varnothing, \delta^-)\}$. The proof proceeds by contradiction, by assuming the existence of a coder $\mathcal{C}$ with $\mathcal{C}(AB)$ in normal form [9], and thus written as $tell(\overrightarrow{t_1}); A_1 + \cdots + tell(\overrightarrow{t_p}); A_p + nask(\overrightarrow{u_1}); B_1 + \cdots + nask(\overrightarrow{u_q}); B_q$. In this expression we will establish that there is no alternative guarded by a $tell(\overrightarrow{t_i})$ operation and no alternative guarded by a $nask(\overrightarrow{u_j})$ operation either, which is impossible since $\mathcal{C}(AB)$ must contain at least one primitive. We notice that the coding of $\mathcal{C}(AB \parallel BA)$ can be written as $\mathcal{C}(AB) \parallel \mathcal{C}(BA)$ by P_2.

Let us first establish that there is no alternative guarded by a $tell(\overrightarrow{t_i})$ operation. Indeed if there is an alternative guarded, say by $tell(\overrightarrow{t_i})$, then $D = \langle \mathcal{C}(AB \parallel BA) | \varnothing \rangle \rightarrow \langle (A_i \parallel \mathcal{C}(BA)) | \{\overrightarrow{t_i}\} \rangle$ is a valid computation prefix of $\mathcal{C}(AB \parallel BA)$. It should deadlock afterwards since $\mathcal{O}(AB \parallel BA) = (\varnothing, \delta^-)$. However D is also a valid computation prefix of $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$. Hence, $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$ admits a failing computation which contradicts the fact that $\mathcal{O}((AB \parallel BA) + (\{\}, \{+a\})) = (\{a\}, \delta^+)$.

Secondly we establish that there is no alternative guarded by a $nask(\overrightarrow{u_j})$ operation. Indeed starting from the empty store, if there is an alternative guarded, say by $nask(\overrightarrow{u_j})$, then $D = \langle \mathcal{C}(AB \parallel BA) | \varnothing \rangle \rightarrow \langle (B_j \parallel \mathcal{C}(BA)) | \{\overrightarrow{t_i}\} \rangle$ is a valid computation prefix of $\mathcal{C}(AB \parallel BA)$. It should deadlock afterwards since $\mathcal{O}(AB \parallel BA) = (\varnothing, \delta^-)$. However D is also a valid computation prefix of $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$. Hence, $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$ admits a failing computation which contradicts the fact that $\mathcal{O}((AB \parallel BA) + (\{\}, \{+a\})) = (\{a\}, \delta^+)$.

$\mathcal{L}_{MR}(\text{ask,tell})$ and $\mathcal{L}_{VB}(\text{nask,tell})$ are not comparable with each other, and so are $\mathcal{L}_{MR}(\text{nask,tell})$ and $\mathcal{L}_{VB}(\text{ask,tell})$.

Proposition 5. $\mathcal{L}_{MR}(\textit{ask,tell}) \wr \mathcal{L}_{VB}(\textit{nask,tell})$

Proof. On the one hand, we have that $\mathcal{L}_{MR}(\text{ask,tell}) \not\leqslant \mathcal{L}_{VB}(\text{nask,tell})$. Otherwise, by non embedding by transitivity, we have $\mathcal{L}_{VB}(\text{ask,tell}) \leqslant \mathcal{L}_{VB}(\text{nask,tell})$ which has been proved impossible (see Fig. 11). On the other hand, $\mathcal{L}_{VB}(\text{nask,tell}) \not\leqslant \mathcal{L}_{MR}(\text{ask,tell})$ is established by contradiction. Indeed, assuming the relation holds, we would then have $\mathcal{L}_{B}(nask, tell) \leqslant \mathcal{L}_{MR}(\text{ask,tell})$, which has been proved impossible in [11].

Proposition 6. $\mathcal{L}_{MR}(\textit{nask,tell}) \wr \mathcal{L}_{VB}(\textit{ask,tell})$

Proof. On the one hand, $\mathcal{L}_{MR}(\text{nask,tell}) \not\leqslant \mathcal{L}_{VB}(\text{ask,tell})$ holds. Otherwise, by embedding by transitivity, we have $\mathcal{L}_{VB}(\text{nask,tell}) \leqslant \mathcal{L}_{VB}(\text{ask,tell})$ which is impossible (see Fig. 11). On the other hand, $\mathcal{L}_{VB}(\text{ask,tell}) \not\leqslant \mathcal{L}_{MR}(\text{nask,tell})$ is established by contradiction by considering $tell(t(1)) ; ask(t(1)))$ and by noting that $\mathcal{O}(tell(t(1)) ; ask(t(1))) = \{(\{t(1)\}, \delta^+)\}$.

We now prove that $\mathcal{L}_{VB}$(ask,nask,tell) and $\mathcal{L}_{MR}$(nask,tell) are not comparable with each other.

Proposition 7. $\mathcal{L}_{VB}(ask,nask,tell) \wr \mathcal{L}_{MR}(nask,tell)$

Proof. **(i)** We have that $\mathcal{L}_{VB}$(ask,nask,tell) $\not\leqslant \mathcal{L}_{MR}$(nask,tell). Otherwise, by embedding by transitivity, $\mathcal{L}_{VB}$(ask,tell) $\leqslant \mathcal{L}_{MR}$(nask,tell), which has been proved impossible in Proposition 6.

(ii) The proof of $\mathcal{L}_{MR}$(nask,tell) $\not\leqslant \mathcal{L}_{VB}$(ask,nask,tell) is an extension of the proof used in Proposition 4 with normal forms extended with *ask* primitives. It is established by considering agent AB = $(\{-a\}, \{+b\})$ and agent BA = $(\{-b\}, \{+a\})$, with $\mathcal{O}((\{-a\}, \{+b\}) \,\|\, (\{-b\}, \{+a\})) = \{(\varnothing, \delta^-)\}$.

Let us now prove that $\mathcal{L}_{MR}$(ask,tell) is strictly less expressive than $\mathcal{L}_{VB}$(ask,nask,tell).

Proposition 8. $\mathcal{L}_{MR}(ask,tell) < \mathcal{L}_{VB}(ask,nask,tell)$

Proof. **(i)** On the one hand, thanks to Proposition 3, $\mathcal{L}_{MR}$(ask,tell) = $\mathcal{L}_{VB}$(ask,tell) $\leqslant$ $\mathcal{L}_{VB}$(ask,nask,tell) and thus $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{VB}$(ask,nask,tell).

(ii) On the other hand, $\mathcal{L}_{VB}$(ask,nask,tell) $\not\leqslant \mathcal{L}_{MR}$(ask,tell), since otherwise, $\mathcal{L}_{VB}$(nask,tell) $\leqslant \mathcal{L}_{VB}$(ask,nask,tell) $\leqslant \mathcal{L}_{MR}$(ask,tell), which has been proved impossible in Proposition 5.

We now prove that $\mathcal{L}_{VB}$(ask,nask,tell) is stricly less expressive than $\mathcal{L}_{MR}$(ask,nask,tell).

Proposition 9. $\mathcal{L}_{VB}(ask,nask,tell) < \mathcal{L}_{MR}(ask,nask,tell)$

Proof. **(i)** On the one hand, the fact that $\mathcal{L}_{VB}$(ask,nask,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,tell) is immediate by Proposition 1. **(ii)** On the other hand, $\mathcal{L}_{MR}$(ask,nask,tell) $\not\leqslant \mathcal{L}_{VB}$(ask,nask,tell). Otherwise, by embedding by transitivity, from $\mathcal{L}_{MR}$(nask,tell) $\leqslant \mathcal{L}_{MR}$(ask,nask,tell), one would get $\mathcal{L}_{MR}$(nask,tell) $\leqslant \mathcal{L}_{VB}$(ask,nask,tell), which has been proved impossible in Proposition 7.

$\mathcal{L}_{VB}$(get,tell) is not comparable with $\mathcal{L}_{MR}$(ask,tell) nor with $\mathcal{L}_{MR}$(ask,nask,tell).

Proposition 10. $\mathcal{L}_{VB}(get,tell) \wr \mathcal{L}_{MR}(ask,tell)$

Proof. See [15].

Proposition 11. $\mathcal{L}_{VB}(get,tell) \wr \mathcal{L}_{MR}(ask,nask,tell)$

Proof. On the one hand, for $\mathcal{L}_{VB}$(get,tell) $\not\leqslant \mathcal{L}_{MR}$(ask,nask,tell), we refer the reader to [15]. On the other hand, $\mathcal{L}_{MR}$(ask,nask,tell) $\not\leqslant \mathcal{L}_{VB}$(get,tell). Otherwise, by transitivity of the embedding, one would have that $\mathcal{L}_{MR}$(ask,tell) $\leqslant \mathcal{L}_{MR}$(ask,nask,tell) $\leqslant \mathcal{L}_{VB}$(get,tell) which has been proved impossible in Proposition 10.

$\mathcal{L}_{MR}$(ask,tell) can be proved to be not comparable with $\mathcal{L}_{VB}$(nask,get,tell) nor is $\mathcal{L}_{MR}$(ask,tell).

Proposition 12. $\mathcal{L}_{MR}(ask,tell) \wr \mathcal{L}_{VB}(nask,get,tell) \wr \mathcal{L}_{MR}(nask,tell)$

Proof. See [15].

We are now in a position to establish that $\mathcal{L}_{VB}$(get,tell) is not comparable with $\mathcal{L}_{MR}$(nask,tell).

Proposition 13. $\mathcal{L}_{VB}(get,tell) \wr \mathcal{L}_{MR}(nask,tell)$

Proof. On the one hand, $\mathcal{L}_{VB}$(get,tell) $\nleqslant$ $\mathcal{L}_{MR}$(nask,tell). Otherwise, as $\mathcal{L}_{VB}$(ask,tell) $<$ $\mathcal{L}_{VB}$(get,tell), one would have $\mathcal{L}_{VB}$(ask,tell) $\leqslant$ $\mathcal{L}_{VB}$(get,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) which has been proved impossible in Proposition 6. On the other hand, $\mathcal{L}_{MR}$(nask,tell) $\nleqslant$ $\mathcal{L}_{VB}$(get,tell). Otherwise, we would have $\mathcal{L}_{MR}$(nask,tell) $\leqslant$ $\mathcal{L}_{VB}$(get,tell) $\leqslant$ $\mathcal{L}_{VB}$(nask,get,tell) which has been proved impossible in Proposition 12.

$\mathcal{L}_{VB}$(nask,get,tell) is not comparable with $\mathcal{L}_{MR}$(ask,nask,tell).

Proposition 14. $\mathcal{L}_{MR}(ask,nask,tell) \wr \mathcal{L}_{VB}(nask,get,tell)$

Proof. **(i)** On the one hand $\mathcal{L}_{MR}$(ask,nask,tell) $\nleqslant$ $\mathcal{L}_{VB}$(nask,get,tell). Otherwise, $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{VB}$(nask,get,tell) which contradicts Proposition 12. **(ii)** On the other hand, $\mathcal{L}_{VB}$(nask,get,tell) $\nleqslant$ $\mathcal{L}_{MR}$(ask,nask,tell). By contradiction, consider $tell(t(1))$; $get(t(1))$. $\mathcal{O}((tell(t(1))$; $get(t(1)))) = \{(\varnothing, \delta^+)\}$. Hence any computation of $\mathcal{C}(tell(t(1)))$; $\mathcal{C}(get(t(1)))$ is successful. Such a computation is composed of a computation for $\mathcal{C}(tell(t(1)))$ followed by a computation for $\mathcal{C}(get(t(1)))$. As $\mathcal{C}(get(t(1)))$ is composed of ask, nask, tell primitives which do not destroy elements on the store, the latter computation can be repeated step by step which yields successful computation for $\mathcal{C}(tell(t(1)))$; $(\mathcal{C}(get(t(1)))$ || $\mathcal{C}(get(t(1))))$. However, $\mathcal{O}(tell(t(1))$; $(get(t(1))$ || $get(t(1)))) = \{(\varnothing, \delta^-)\}$.

C. Retrieving Tokens from the Store

Proposition 15. $\mathcal{L}_{VB}(get,tell) < \mathcal{L}_{MR}(get,tell)$

Proof. See [15].

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

Proposition 16. $\mathcal{L}_{MR}(get,tell) \wr \mathcal{L}_{VB}(nask,tell)$

Proof. On the one hand, $\mathcal{L}_{MR}$(get,tell) $\nleqslant$ $\mathcal{L}_{VB}$(nask,tell). Otherwise, $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(nask,tell) which has been proved impossible in [11]. On the other hand, $\mathcal{L}_{VB}$(nask,tell) $\nleqslant$ $\mathcal{L}_{MR}$(get,tell) is established by contradiction, by considering $tell(t(1))$; $nask(t(1))$. Indeed, one has $\mathcal{O}(tell(t(1))$; $nask(t(1))) = \{(\{t(1)\}, \delta^-)\}$ whereas it is possible to establish that $\mathcal{C}(tell(t(1)))$; $\mathcal{C}(nask(t(1)))$ has a successful computation.

Proposition 17. $\mathcal{L}_{MR}(get,tell) \wr \mathcal{L}_{VB}(nask,get,tell)$

Proof. On the one hand, $\mathcal{L}_{MR}$(get,tell) $\nleqslant$ $\mathcal{L}_{VB}$(nask,get,tell). Otherwise, as $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell), we then have $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{VB}$(nask,get,tell) which has been proved impossible in Proposition 12. On the other hand, $\mathcal{L}_{VB}$(nask,get,tell) $\nleqslant \mathcal{L}_{MR}$(get,tell). Otherwise, we would have $\mathcal{L}_{VB}$(nask,tell) $\leqslant \mathcal{L}_{MR}$(get,tell) which has been proved impossible in Proposition 16.

Proposition 18. $\mathcal{L}_{MR}(get,tell) \wr \mathcal{L}_{VB}(ask,nask,tell)$

Proof. On the one hand, $\mathcal{L}_{MR}$(get,tell) $\nleqslant$ $\mathcal{L}_{VB}$(ask,nask,tell). Otherwise, one has $\mathcal{L}_{MR}$(ask,tell) $\leqslant$ $\mathcal{L}_{MR}$(get,tell) $\leqslant$ $\mathcal{L}_{VB}$(ask,nask,tell) which has been proved impossible in Proposition 8. On the other hand, $\mathcal{L}_{VB}$(ask,nask,tell) $\nleqslant \mathcal{L}_{MR}$(get,tell). Otherwise, one would have $\mathcal{L}_{MR}$(ask,tell) $\leqslant \mathcal{L}_{MR}$(ask,nask,tell) $\leqslant \mathcal{L}_{VB}$(get,tell) which has been proved impossible in Proposition 10.

## D. Checking for Presence and/or Absence When Adding and/or Retrieving Tokens

We finally prove that $\mathcal{L}_{VB}$(ask,nask,get,tell) is strictly less expressive than $\mathcal{L}_{MR}$(ask,nask,get,tell).

Proposition 19. $\mathcal{L}_{VB}(ask,nask,get,tell) < \mathcal{L}_{MR}(ask,nask,get,tell)$

Proof. On the one hand, $\mathcal{L}_{VB}$(ask,nask,get,tell) $\leqslant$ $\mathcal{L}_{MR}$(ask,nask,get,tell) is immediate by Proposition 1. On the other hand, $\mathcal{L}_{MR}$(ask,nask,get,tell) $\nleqslant$ $\mathcal{L}_{VB}$(ask,nask,get,tell) is established by contradiction. Indeed, assuming that $\mathcal{L}_{MR}$(ask,nask,get,tell) $\leqslant$ $\mathcal{L}_{VB}$(ask,nask,get,tell), as $\mathcal{L}_{VB}$(ask,nask,get,tell) $=$ $\mathcal{L}_{VB}$(nask,get,tell), one would have $\mathcal{L}_{MR}$(nask, tell) $\leqslant \mathcal{L}_{MR}$(ask,nask,get,tell) $\leqslant$ $\mathcal{L}_{VB}$(ask,nask,get,tell) $\leqslant \mathcal{L}_{VB}$(nask,get,tell) which has been proved impossible in Proposition 12.

E. Summary

Figure 12 provides a summary of the expressiveness results developed in this paper in a three dimensional perspective. It is worth observing that the Vectorized Dense BachT language obeys the same hierarchy as the BachT, Dense BachT and MRT languages. This is due to the nature of the tell, ask, nask and get primitives, which is preserved by the extension provided to the tokens. It is also worth noting that the sublanguage reduced to the tell primitive has the same power in all the languages. The other sublanguages obey the expressiveness studies already developed for BachT and Dense BachT. Finally, the Vectorized Dense Bach sublanguages appear to be strictly more expressive than their Dense BachT counterparts but are strictly less expressive than their MRT counterparts.
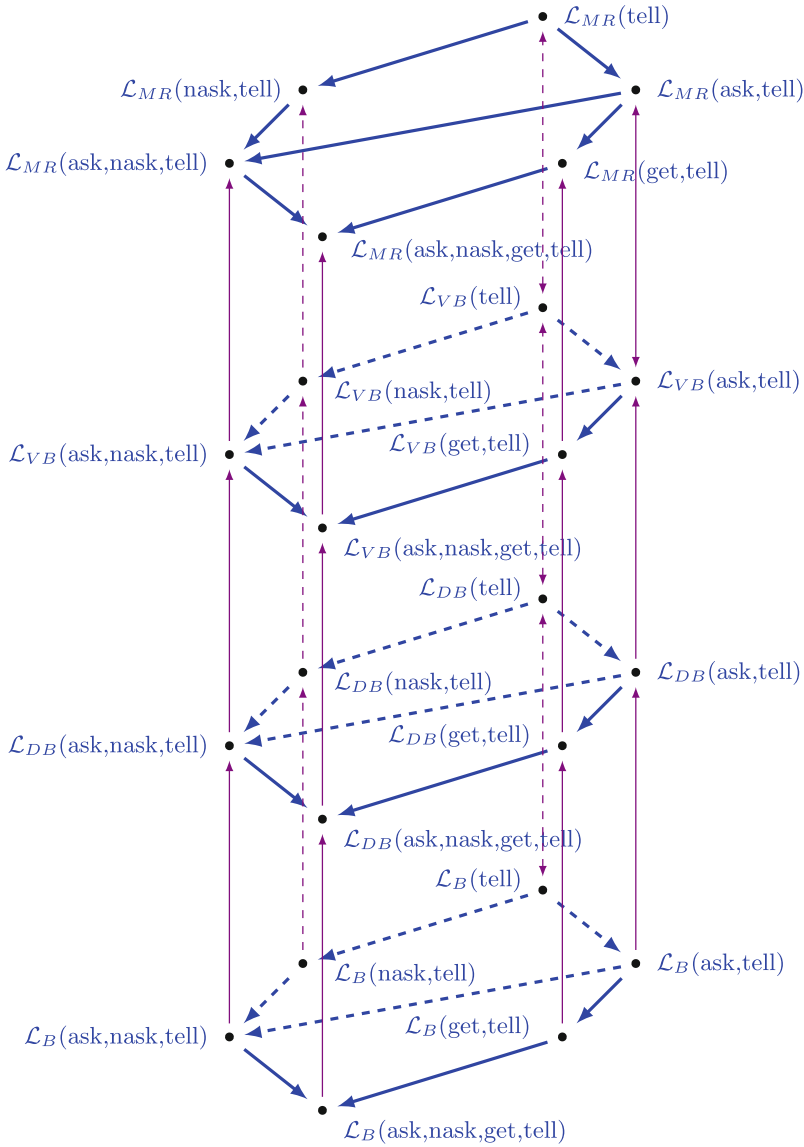
Fig. 12. Three-dimensional representation of the expressiveness relations between the different languages.

The notable exception is provided by the $\mathcal{L}_{VB}$(ask,tell) sublanguage, which is as expressive as the $\mathcal{L}_{MR}$(ask,tell) sublanguage.

Note that, in the picture, the dash arrows are drawn to suggest the three-dimensional perspective but have the same meaning as the plain arrows.

5   From Tokens to Tuples

As announced in the introduction, the paper has so far concentrated on token based versions of the languages. A natural question to ask is whether the expressiveness study performed in Sect. 4 can be extended to tuples and thereby can embrace Linda-like languages in their full version.

To that end, two issues need to be taken into account. On the one hand, structured pieces of information need to be tackled instead of flat tokens. Using our notations, this would lead to consider tuples of the form $\langle t_1, \cdots, t_n \rangle$ instead of t as arguments of *tell*, *ask*, *get* and *nask* primitives. On the other hand, it is desirable to introduce variables as arguments of the tuples in *ask*, *get* and *nask* primitives to retrieve values from the tuples stored on the tuple space[1]. The resulting tuples are classically called templates or anti-tuples.

It turns out that tackling these issues can be reduced to using flat tokens by assuming a very reasonable hypothesis: variables have to range over enumerable sets of values. If this is the case, then, as exemplified in process algebras like mCRL2 [19], any primitive containing a tuple with variables can be rewritten as a choice of that primitive with the variables instantiated to values. For instance, $ask(\langle 1, X : Int \rangle)$ can be rewritten as $\sum_{i \in Int} ask(\langle 1, i \rangle)$. As a result, assuming a general choice over enumerable sets in the language, we may reduce the language to primitives involving tuples without variables. As a further step, tuples having a finite number of arguments and the choices being on enumerable sets, these tuples range over enumerable unions of enumerable sets, namely over an enumerable set. We may thus associate any of these tuples to a token of *Stoken* and vice-versa. By doing so, we are back to the token-based languages studied before, provided that an interpretation is given to dense tuples. Two are actually possible. Consider for instance a store containing twice $t(1)$, three times $t(2)$ and one time $t(3)$. Then, in a first interpretation, the request $ask(t(X : Int)(3))$ can be satisfied if one may instantiate X to an integer, say i, such that the induced instance $t(i)$ appears at least three times on the store. In our example, this would be possible by giving to X the value 2. Under this interpretation, the translation just provided from tuple-based languages to token-based languages applies directly, which thus allows to lift the results obtained in Sect. 4 to tuple-based languages. In another interpretation, one may argue that one should find three occurrences of tuples which matches $t(X)$ by possibly instantiating X to different values. Under this interpretation, the request $ask(t(X : Int)(3))$ not only succeeds but also $ask(t(X : Int)(5))$ since $t(1)$ and $t(2)$ both match $t(X)$. However, this interpretation is exactly what is captured by DBD-BachT language. In our example, $ask(t(X : Int)(5))$ could indeed be reformulated as $ask([t(1), t(2), t(3), \cdots](5))$. Hence, under that second interpretation too, the results obtained in Sect. 4 can be lifted to tuple-based languages.

---

[1] In Linda, variables are also allowed in *tell* primitives to denote unknown attributes. However, as argued in [12], we believe that it is better to use ψ-terms in this case, which allows to keep the idea of structured information without the need for writing unknown arguments.

6   Implementation Issues

The expressiveness study has shown that BachT is strictly less expressive than Dense BachT, which itself is strictly less expressive than VD-BachT, which is finally less expressive than MRT. One may thus wonder about the interest of all the languages, except the most expressive one. The aim of this section is to convince the reader that implementing the more expressive languages comes with a higher cost and thus that one better selects the language just expressive enough for its coding purposes.

To start with, let us first detail how the token space (or more generally the tuple space[2]) may be implemented for Bach. Since our first implementation (see [4]), processes have been implemented as threads and the tuple space has been implemented as a token-indexed list. Per list element (token), we keep track of the number of identical tokens (*token counter*), and of the input primitives that are suspended on this token. The token list is stored in shared memory. The list is directly updated by the communication primitives, as one may guess: the tell primitive adding tokens and the get primitive consuming them. In order to guarantee exclusive access, the individual list elements are protected by a lock, which means that operations on different tokens can execute in parallel. This has turned out to generate interesting speed-ups over the naive implementation which would lock the entire tuple space for each primitive execution. More precisely, the following algorithms are employed for the primitives.

Performing an $nask$ primitive first checks whether the associated token is known to the tuple space (i.e. has already an element in the list of tokens). If not, the tuple space is locked, and a list element for the token is created. If the token counter equals zero, the $nask$-primitive succeeds. If not, it suspends, and is added to the list of suspended primitives, until the token counter reaches zero.

Asking a token t first checks whether at least one occurrence of t is present in the tuple space. If so, the primitive succeeds. Otherwise, the ask primitive is put in the associated list of waiting processes, until the token counter is positive.

Getting a token t proceeds similarly but decrements the token counter for t. If the token counter reaches zero, we check whether there are suspended $nask(t)$ primitives. If so, the process associated with the $nask$ primitive is resumed and is removed from the list of waiting primitives.

Finally, telling a token t proceeds dually. The list of waiting processes is first inspected to discover an ask or get primitive waiting for t. In case an ask primitive is discovered, it is resumed and the search continues. In case a get primitive is discovered the token t is consumed by that primitive and the corresponding process is resumed. If no waiting get primitives are encountered, then the token counter for t is incremented.

---

[2] In Bach, following [12], tuples are actually represented in the form of a functor name followed by a series of pairs, each consisting of an attribute associated with a value. Without entering into details, the functor names play the role of tokens and, in that manner, the implementation sketched in this section can be lifted from the tokens considered in this paper to more general tuples.

As the careful reader will have noticed, lifting the Bach implementation to Dense Bach is quite easy. One basically just needs to count the number of occurrences of the tokens for the ask and get primitives and to upgrade the number of tokens for the tell primitives. The case of general tuples is slightly more complicated in that pattern matching needs to be done in addition but only on the elements of the list associated with the functor taken as a token. However, the key property remains: the tuple space need not be blocked globally, locks are only put on the list associated with the considered token.

Moving to VD-Bach is more subtle since several locks need to be taken and hence the above key property cannot be met. Consider for instance a multi-get primitive which needs to consume tokens a, b and c. To evaluate it, one needs in principle to lock the lists associated with a, b and c. However, as other primitives may compete, for instance to get b, c and d, one actually needs to lock at once the three lists associated with a, b and c in order to prevent the system from deadlocks. In practice, an easy way to do so is to lock the whole tuple space. However, by using abstract interpretation techniques on a static code or by using declarations on the vectors employed and by employing the activator vectors of [21], one may slightly relax the global lock by adding locks for super sets of vectors, in the example above to a, b, c and d. This still allows for parallel computations, although to a lesser extend than for Dense Bach.

It is worth noting that the more the structure involves tokens the more constraining are the locks used. In particular, a similar technique can be used to implement MRT but, as one may expect, with a greater computation overhead.

As a conclusion, the more expressive the language is the more expensive is its implementation. Hence, there is obviously a trade-off to be made between the programming ease offered by the language expressiveness and the computation costs needed by its implementation.

7   Conclusion

This paper is written in the continuity of our previous research on the expressiveness of Linda-like languages. It has presented extensions of our Bach language aiming at handling multiplicities. In particular, as a novel piece of work, we have presented an extension of our Dense BachT language, that has promoted the interest of vectors of dense tokens. The new language, called Vectorized Dense BachT proposes to atomically perform multiple operations on dense tokens by introducing lists of dense tokens in the four classical primitives of our BachT language.

Our work thus builds upon our previous work [10,11,16,20,22,24–26]. 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 sub-languages of Dense BachT and Vectorized Dense BachT. Accordingly, we have established a gain of expressivity, namely that Vectorized Dense BachT is strictly more expressive than Dense BachT and, consequently, in view of the results of [20], strictly more expressive than the BachT and Linda languages. However the

structure of the hierarchies of the sublanguages of a family is kept, which shows that the very nature of the tell, ask, get and nask primitives is preserved. We have also compared Vectorized Dense BachT with a multiset rewriting language and showed that it is strictly less expressive. However, as shown in the paper, it is expressive enough to code interesting applications as well as Dense Bach with Distributed Density, a language we introduced in [16]. Moreover, the fact that Vectorized Dense BachT only provides atomic tell, ask, nask and get allows for more efficient implementations than MRT.

Our work has similarities but also differences with several work on the expressiveness of Linda-like languages. Compared to [33, 34], 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, 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 [3], nine variants of the $\mathcal{L}_B(ask, nask, get, tell)$ language are studied. They are obtained by varying both the nature of the shared data space and its structure. Rephrased in the setting of [17], 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 [14], 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 [13], 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 [3, 13, 14, 33, 34] do not deal with a notion of density attached to tuples. In contrast, [5, 6] 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 [5] 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 [5, 6] 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. As can be appreciated by the reader through the comparison of BachT, Dense BachT and Vectorized Dense BachT, 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. This will be the subject for future research.

In [32], Viroli and Casadei propose a stochastic extension of the Linda framework, with a notion of tuple concentration, similar to the weight of [5,6] 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 notions of multiplicity can be the basis of such probabilistic extensions.

References

1. Banâtre, J.-P., Le Métayer, D.: Programming by multiset transformation. Commun. ACM **36**(1), 98–111 (1993)
2. Banâtre, J.-P., Le Métayer, D.: Gamma and the chemical reaction model: ten years after. In: Coordination Programming, pp. 3–41. Imperial College Press, London (1996)
3. Bonsangue, M.M., Kok, J.N., Zavattaro, G.: Comparing coordination models based on shared distributed replicated data. In: ACM Symposium on Applied Computing, pp. 156–165 (1999)
4. De Bosschere, K., Jacquet, J.-M.: Multi-prolog: definition, operational semantics, and implementation. In: Warren, D.S. (ed.) Proceedings of the International Conference on Logic Programming, Budapest, Hongrie, pp. 299–314. The MIT Press (1993)
5. Bravetti, M., Gorrieri, R., Lucchi, R., Zavattaro, G.: Quantitative information in the tuple space coordination model. Theoret. Comput. Sci. **346**(1), 28–57 (2005)
6. Bravetti, M., Gorrieri, R., Lucchi, R., Zavattaro, G.: Probabilistic and prioritized data retrieval in the Linda coordination model. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 55–70. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24634-3_7
7. Bravetti, M., Zavattaro, G.: Service oriented computing from a process algebraic perspective. J. Logic Algebr. Program. **70**(1), 3–14 (2007)

8. Brogi, A., Jacquet, J.-M.: Modeling coordination via asynchronous communication. In: Garlan, D., Le Métayer, D. (eds.) COORDINATION 1997. LNCS, vol. 1282, pp. 238–255. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63383-9_84

9. Brogi, A., Jacquet, J.-M.: On the expressiveness of Linda-like concurrent languages. Electron. Not. Theoret. Comput. Sci. **16**(2), 61–82 (1998)

10. Brogi, A., Jacquet, J.-M.: On the expressiveness of coordination models. In: Ciancarini, P., Wolf, A.L. (eds.) COORDINATION 1999. LNCS, vol. 1594, pp. 134–149. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48919-3_11

11. Brogi, A., Jacquet, J.-M.: On the expressiveness of coordination via shared dataspaces. Sci. Comput. Program. **46**(1–2), 71–98 (2003)

12. Brogi, A., Jacquet, J.-M., Linden, I.: On modeling coordination via asynchronous communication and enhanced matching. Electron. Not. Theoret. Comput. Sci. **68**(3), 284–309 (2003)

13. Busi, N., Gorrieri, R., Zavattaro, G.: On the Turing equivalence of Linda coordination primitives. Electron. Not. Theoret. Comput. Sci. **7**, 75–75 (1997)

14. Busi, N., Gorrieri, R., Zavattaro, G.: A process algebraic view of Linda coordination primitives. Theoret. Comput. Sci. **192**, 167–199 (1998)

15. Darquennes, D.: On Multiplicities in Coordination Languages. Ph.D. thesis, Faculty of Computer Science, University of Namur, Namur, Belgium (2017)

16. Darquennes, D., Jacquet, J.-M., Linden, I.: On distributed density in tuple-based coordination languages. In: Cámara, J., Proença, J. (eds.) Proceedings of the 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems. EPTCS, vol. 175, pp. 36–53. Springer (2015)

17. de Boer, F.S., Palamidessi, C.: Embedding as a tool for language comparison. Inf. Comput. **108**(1), 128–157 (1994)

18. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985)

19. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)

20. Jacquet, J.-M., Linden, I., Darquennes, D.: On density in coordination languages. In: Canal, C., Villari, M. (eds.) ESOCC 2013. CCIS, vol. 393, pp. 189–203. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45364-9_16

21. Jacquet, J.-M., Linden, I., Staicu, M.-O.: Blackboard rules: from a declarative reading to its application for coordinating context-aware applications in mobile ad hoc networks. Sci. Comput. Program. **115–116**, 79–99 (2016)

22. Jacquet, J.M., Linden, I., Darquennes, D.: On the introduction of density in tuple-space coordination languages. Sci. Comput. Program. **115–116**, 149–176 (2016)

23. Jongmans, S.-S.T.Q., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Automatic code generation for the orchestration of web services with Reo. In: De Paoli, F., Pimentel, E., Zavattaro, G. (eds.) ESOCC 2012. LNCS, vol. 7592, pp. 1–16. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33427-6_1

24. Linden, I., Jacquet, J.-M.: On the expressiveness of absolute-time coordination languages. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 232–247. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24634-3_18

25. Linden, I., Jacquet, J.-M.: On the expressiveness of timed coordination via shared dataspaces. Electron. Not. Theoret. Comput. Sci. **180**(2), 71–89 (2007)

26. Linden, I., Jacquet, J.-M., De Bosschere, K., Brogi, A.: On the expressiveness of relative-timed coordination models. Electron. Not. Theoret. Comput. Sci. **97**, 125–153 (2004)

27. Linden, I., Jacquet, J.-M., De Bosschere, K., Brogi, A.: On the expressiveness of timed coordination models. Sci. Comput. Program. **61**(2), 152–187 (2006)
28. Mariani, S.: Coordination of Complex Sociotechnical Systems - Self-organisation of Knowledge in MoK. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47109-9
29. Omicini, A.: Formal ReSpecT in the A&A perspective. Electron. Not. Theoret. Comput. Sci. **175**(2), 97–117 (2007)
30. Shapiro, E.: Embeddings among concurrent programming languages. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 486–503. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0084811
31. Tolksdorf, R.: Laura - a service-based coordination language. Sci. Comput. Program. **31**(2–3), 359–381 (1998)
32. Viroli, M., Casadei, M.: Biochemical tuple spaces for self-organising coordination. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 143–162. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02053-7_8
33. Zavattaro, G.: On the incomparability of Gamma and Linda. In: Electronic Transactions on Numerical Analysis (1998)
34. Zavattaro, G.: Towards a hierarchy of negative test operators for generative communication. Electron. Not. Theoret. Comput. Sci. **16**, 154–170 (1998)