Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 97 (2004) 125–153
www.elsevier.com/locate/entcs

On the Expressiveness of Relative-Timed Coordination Models

I. Linden and J.-M. Jacquet[1]

Institute of Informatics
University of Namur
Namur, Belgium

K. De Bosschere[2]

ELIS
Ghent University
Ghent, Belgium

A. Brogi[3]

Department of Computer Science
University of Pisa
Pisa, Italy

Abstract

Although very simple and elegant, Linda-style coordination models lack the notion of time, and are therefore not able to precisely model real-life coordination applications. Nevertheless, industrial proposals such as TSpaces and JavaSpaces, inspired from Linda, have incorporated time constructs. This paper aims at a systematic study of the introduction of relative time in coordination models. It builds upon previous work to study the expressiveness of Linda, Linda extended with a delay mechanism and Linda primitives extended to support the duration of tuples and of the suspension of communication operations.

Keywords: Temporal coordination languages, semantics, expressiveness

[1] Email: ili,jmj@info.fundp.ac.be
[2] Email: kdb@elis.rug.ac.be
[3] Email: brogi@di.unipi.it

doi:10.1016/j.entcs.2004.04.034

1 Introduction

As motivated by the constant expansion of computer networks and illustrated by the development of distributed applications, the design of modern software systems centers on re-using and integrating software components. This induces a paradigm shift from stand-alone applications to interacting distributed systems, which, in turn, naturally calls for well-defined methodologies and tools aiming at integrating heterogeneous software components.

In this context, a clear separation between the *interactional* and the *computational* aspects of software components has been advocated by Gelernter and Carriero in [15]. Their claim has been supported by the design of a model, Linda ([9]), originally presented as a set of inter-agent communication primitives which may be added to almost any programming language. Besides process creation, this set includes primitives for adding, deleting, and testing the presence/absence of data in a shared dataspace.

A number of other models, now referred to as coordination models, have been proposed afterwards. Some of them extend Linda in different ways, for instance by introducing multiple dataspaces and meta-level control rules (e.g., Bauhaus Linda [22], Bonita [26], μLog [18], PoliS [11], Shared Prolog [5]), by addressing open distributed systems (e.g., Laura [33]), middleware web-based environments (e.g., Jada [12]), or mobility (e.g., KLAIM [23]). A number of other coordination models rely on a notion of shared dataspace, e.g., Concurrent Constraint Programming [29], Gamma [2] and Linear Objects [1], to cite only a few. A comprehensive survey of these and other coordination models and languages has been recently reported in [25].

However, the coding of applications reveals that data rarely has an eternal life and that services have to be provided in a bounded amount of time. For instance, a request for information on the web has to be satisfied in a reasonable amount of time. More crucial is even the request for an ambulance which, not only has to be answered eventually but within a critical period of time. The list could also be continued with software in the areas of air-traffic control, manufacturing plants and telecommunication switches, which are inherently reactive and, for which, interaction must occur in "real-time".

Although there is an obvious application need, the introduction of time has not been deeply studied in the context of coordination languages and models, the notable exceptions being [4,24,27,28], yet proposed in the context of concurrent constraint programming, and [7,8].

This paper aims at contributing to the study of time in coordination languages and models. More precisely, it builds upon [19] to perform a systematic and exhaustive study of two extensions proposed there. These extensions

adopt the classical *two-phase functioning* approach to real-time systems illustrated by languages such as Lustre ([10]), Esterel ([3]) and Statecharts ([16]). This approach may be described as follows. In a first phase, elementary actions of statements are executed. They are assumed to be atomic in the sense that they take no time. Similarly, composition operators are assumed to be executed at no cost. In a second phase, when no actions can be reduced or when all the components encounter a special timed action, time progresses by one unit.

Although simple, this approach has been proved to be effective for modelling reactive systems. For instance, in many reactive systems, timed actions determine instants at which inputs are sampled or output is produced. In the coordination context, it still leaves room for several variants:

(i) time may be introduced in the form of delays, stating that a communication primitives should only be processed after some units of time;

(ii) time may also be introduced by stating that tuples on the tuple space are only valid for some units of time; similarly, requests for tuples cannot be postponed indefinitely;

(iii) time may finally be introduced by specifying intervals of time in which actions should be processed.

In this paper, we consider only the first two extensions. These extensions were introduced in [19] with some expressiveness results. However, as will be appreciated by the reader, this paper aims at a much deeper study of these expressiveness results. In particular, all the results of sections 3 and 4.2 are original with respect to [19] while the comparisons of sections 4.1 and 4.3 have lead to 8 new results with respect to [19].

We postpone the comparison with other work to section 5 when technical notions and results necessary for the discussion will have been introduced.

The rest of this paper is structured as follows. Section 2 introduces the families of languages under study in the paper. All of them rest on common sequential, parallel and choice operators. The Linda-like languages are first modelled as the $\mathcal{L}$ family. Relative delays are then introduced and relative timing primitives are defined thereafter. The expressiveness hierarchy of each family of languages, considered in isolation, is studied in section 3. The inter-family comparison is discussed in section 4. In order to keep the size of this paper within reasonable limits, only the most difficult proofs have been done. We refer the interested reader to [20] where all the results are demonstrated. Finally, in section 5 we draw our conclusion and discuss related work.

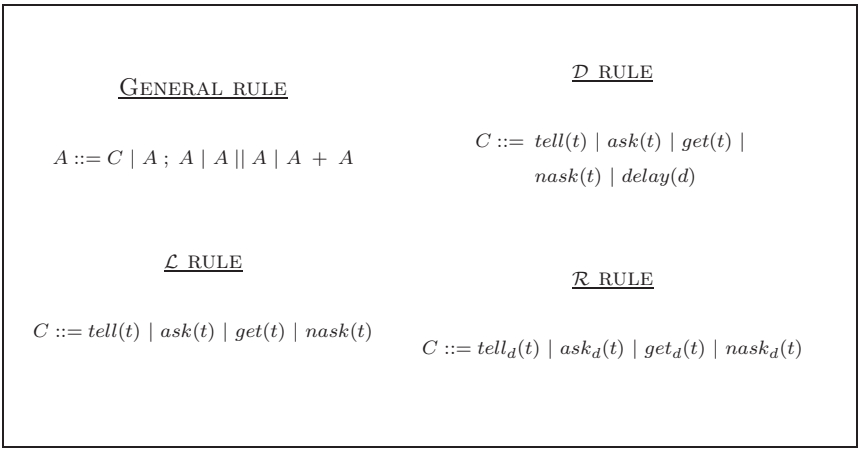

Fig. 1. Comparative syntax of the languages.

2 The families of languages

2.1 Common syntax and rules

All languages considered in this paper contain sequential, parallel and choice operators. They differ only in the set of communication primitives they embody. As a result, assuming such a set, the syntax of a statement, subsequently called agent, is defined by the "general rule" of figure 1 and its semantics is provided by rules (S), (P), and (C) of figure 2. There, configurations are of the form $\langle A \mid \sigma \rangle$ where A represents the agent under consideration and σ represents a memory, to be specified for each family of languages.

Note that, for simplicity of presentation, only finite processes are treated here, under the observation that infinite processes can be handled by extending the results of this paper in the classical way, as exemplified for instance in [17].

2.2 The family of Linda-like concurrent languages

To start with, consider the family of languages $\mathcal{L}(\mathcal{X})$, parameterized on the set of Linda-like communication primitives $\mathcal{X}$. This set $\mathcal{X}$ consists of the basic Linda primitives `out`, `in`, and `rd`, for putting an object in a shared dataspace, getting it and checking for its presence, respectively, together with a primitive testing the absence of an object from the dataspace. Formally, the language is defined as follows.

Definition 2.1 Let *Stoken* be an enumerable set, the elements of which are subsequently called *tokens* and are typically represented by the letters t and u. Define the set of communication actions *Scom* as the set generated by the

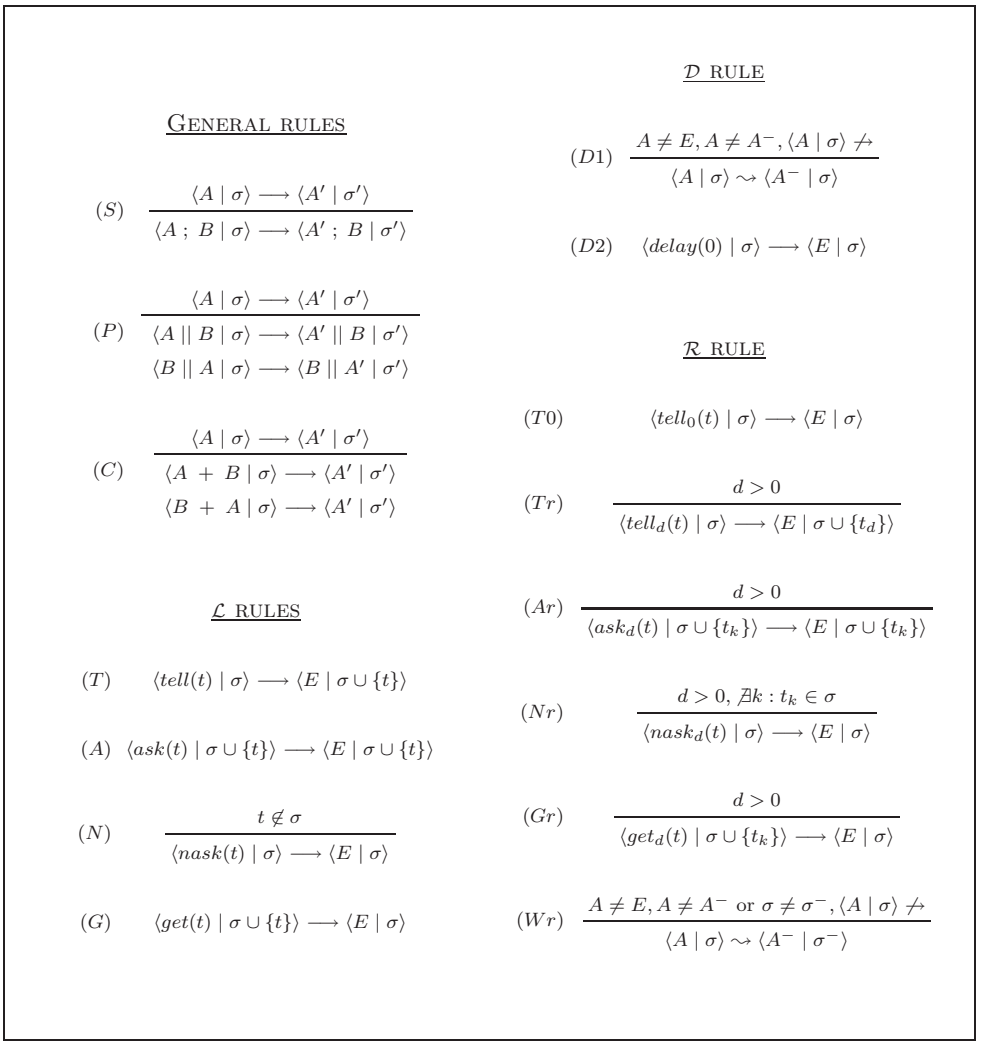

Fig. 2. Comparative semantics of the languages.

$\mathcal{L}$ rule of figure 1. Moreover, for any subset $\mathcal{X}$ of $Scom$, define the language $\mathcal{L}(\mathcal{X})$ as the set of agents A generated by the general rule of figure 1.

For any $\mathcal{X}$, computations in $\mathcal{L}(\mathcal{X})$ may be modelled by a transition system written in Plotkin's style. Following the intuition, most of the configurations consist of an agent together with a multi-set of tokens denoting the tokens currently available for the computation. To easily express termination, we shall introduce particular configurations composed of a special terminating symbol E together with a multi-set of tokens. For uniformity, we shall abuse

language and qualify E as an agent. However, to meet the intuitive expectaion, we shall always rewrite agents of the form $(E\ ;\ A)$, $(E \mid\mid A)$, and $(A \mid\mid E)$ as A. This is technically achieved by defining the extended set of agents as follows, and through simplifications derived by imposing a bimonoid structure.

Definition 2.2 Define the extended set of agents $Seagent$ by the following grammar

$$Ae ::= E \mid C \mid A\ ;\ A \mid A \mid\mid A \mid A\ +\ A$$

Moreover, we shall subsequently assert that the structure $(Seagent, E, \ ;\ ,\ \mid\mid\ )$ is a bimonoid and simplify elements of $Seagent$ accordingly.

Definition 2.3 Define the set of stores $Sstore$ as the set of finite multisets with elements from Stoken.

Definition 2.4 Define the set of configurations $Sconf$ as $Seagent \times Sstore$. Configurations are denoted as $\langle A \mid \sigma \rangle$, where A is an (extended) agent and σ is a multi-set of tokens.

Definition 2.5 The transition rules for the $\mathcal{L}$ agents are the general ones of figure 2 together with rules (T), (A), (N), (G) of that figure, where σ denotes a multi-set of tokens.

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

We are now in a position to define the operational semantics.

Definition 2.6

(i) Let δ^+ and δ^- be two fresh symbols denoting respectively success and failure. Define the set of final states $Sfstate$ as the set $Sstore \times \{\delta^+, \delta^-\}$.

(ii) Define the *operational semantics* $\mathcal{O} : Sagent \to \mathcal{P}(Sfstate)$ as the following function: For any agent A,

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

2.3 Normal form

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

Definition 2.7 Given a subset $\mathcal{X}$ of *Scom*, the set *Snagent* of agents in normal form is defined by the following rule, where N is an agent in normal form and c denotes a communication action of $\mathcal{X}$.

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

Proposition 2.8 *For any agent A, there is an agent in normal form N such that $\mathcal{O}(A) = \mathcal{O}(N)$.*

2.4 The family of Linda-like concurrent languages with delay

One way of introducing time in coordination languages is to postpone the execution of the primitives for some period of time. This amounts to introducing a special delay primitive.

Definition 2.9 Let *Stime* be the set of positive integers. Define the set *Sdcom* as the set generated by the $\mathcal{D}$ rule of figure 1, where $t \in Stoken$ and $d \in Stime$. Moreover, for any subset $\mathcal{X}$ of $Sdcom \setminus \{delay\}$, define the language $\mathcal{D}(\mathcal{X})$ as the set of agents generated by the general rule of figure 1 for $C \in \mathcal{X} \cup \{delay\}$.

The configurations to be considered here are similar to those used for the $\mathcal{L}$ family. However, time needs to be taken into account explicitly in the transitions. This done in two ways. First, by the introduction of a new rule (D1), which defines a new transition relation $\rightsquigarrow$ to express the progress of time by one unit. In fact, the $\rightarrow$ reduction is used to model the first phase of the two-phase functioning approach to real-time while the $\rightsquigarrow$ relation is used to model the second phase of this approach.

Second, as a result of the progress of time, delays under reduction, must be decreased by one unit. This is achieved by the A^- construct. Note that, to avoid that the computation infinitely tries to decrease blocked non-delay primitives, rule (D1) requires A^- to express some progress, namely to be different than A.

Finally, rule (D2) is introduced to reduce a delay of 0 unit of time to E.

Summing up, the transitions to be considered are defined as follows.

Definition 2.10 Define the set of configurations $Sdconf$ as $Seagent' \times Sstore$, where $Seagent'$ is the set of extended agents defined as in definition 2.2 but by taking $C \in Sdcom$ instead of $C \in Scom$.

Definition 2.11 Given an agent $A \in \mathcal{D}(\mathcal{X})$, we denote by A^- the agent defined inductively as follows where $d > 0$

$$\begin{array}{lll} tell(t)^- = tell(t) & get(t)^- = get(t) & (B\ ;\ C)^- = B^-\ ;\ C \\ ask(t)^- = ask(t) & delay(0)^- = delay(0) & (B \mid\mid C)^- = B^- \mid\mid C^- \\ nask(t)^- = nask(t) & delay(d)^- = delay(d-1) & (B\ +\ C)^- = B^-\ +\ C^- \end{array}$$

Definition 2.12 Define the transition rules for the $\mathcal{D}$ agents as the general ones of figure 2 and rules (T), (A), (N), (G), (D1) and (D2) of that figure.

The operational semantics is defined by integrating the two phase-relations in one relation.

Definition 2.13

(i) Let $\longmapsto$ be the relation defined by $\langle A \mid \sigma\rangle \longmapsto \langle B \mid \tau\rangle$ iff $\langle A \mid \sigma\rangle \rightarrow \langle B \mid \tau\rangle$ or $\langle A \mid \sigma\rangle \rightsquigarrow \langle B \mid \tau\rangle$.

(ii) Define the *operational semantics* $\mathcal{O}_d : \mathcal{D}(Sdcom) \rightarrow \mathcal{P}(Sfstate)$ as the following function: For any timed agent A,

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

2.5 The family of Linda-like concurrent languages with relative durations

A second way of introducing time in the family $\mathcal{L}(\mathcal{X})$ consists of enriching the primitives ask, nask, get, and tell themselves by durations. Formally, the new family of languages $\mathcal{R}(\mathcal{X})$ is defined as follows.

Definition 2.14 Define the set $Stcom$ of timed communication primitives as the one generated by the $\mathcal{R}$ rule of figure 1, where $t \in Stoken$ and $d \in Stime \cup \{\infty\}$. For any subset $\mathcal{X}$ of Stcom, define the language $\mathcal{R}(\mathcal{X})$ as the set of agents generated by the general rule of figure 1.

The configurations to be considered for the family $\mathcal{R}(\mathcal{X})$ are similar to those used for the family $\mathcal{L}(\mathcal{X})$. The introduction of time induces here the following adaptations:

(i) The intuition behind the construct $tell_d(t)$ is that t is added to the store but for d units of time only. To capture this fact, the tokens of the store

have associated durations.

(ii) As another consequence, this duration has to be updated after each tick of the clock. This motivates the introduction of the $-$ operator acting on the store.

(iii) Similarly, the intuition behind the $ask_d(t)$, $nask_d(t)$, and $get_d(t)$ primitives is that, if needed, suspension may occur only up to d units of time. As a result, a similar operator, also denoted $-$, has to be introduced to decrease the period of suspension after each tick of the clock.

This intuition leads to the following definitions.

Definition 2.15

(i) Given an agent $A \in \mathcal{R}(\mathcal{X})$, we denote by A^- the agent defined inductively as follows: [4]

$$
\begin{aligned}
tell_d(t)^- &= tell_d(t) \\
ask_d(t)^- &= ask_{max\{0,d-1\}}(t) \qquad & (B\ ;\ C)^- &= B^-\ ;\ C \\
nask_d(t)^- &= nask_{max\{0,d-1\}}(t) \qquad & (B\ ||\ C)^- &= B^-\ ||\ C^- \\
get_d(t)^- &= get_{max\{0,d-1\}}(t) \qquad & (B\ +\ C)^- &= B^-\ +\ C^-
\end{aligned}
$$

(ii) Define the set of timed stores $Ststore$ as the set of multisets of elements of the form t_d where t is a token and d is a duration. Given a timed store σ, we denote by σ^- the new store obtained by decreasing the duration associated with the tokens by one unit and by removing those associated in σ with 1 unit of time: precisely, if all the notations are understood to relate to multi-sets: $\sigma^- = \{t_{d-1} : t_d \in \sigma, d > 1\}$

(iii) Define the set of configurations $Sconf$ as $Seagent \times Ststore$. Configurations are denoted as $\langle A \mid \sigma \rangle$, where A is an (extended) timed agent and σ is a timed store.

Due to the introduction of time, the operational semantics is defined by means of the transition relations $\rightarrow$ and $\rightsquigarrow$ describing the two phase approach. They basically adapt the relations defined for the $\mathcal{L}$ family. Accordingly, rules (Tr), (Ar), (Nr), and (Gr) adapt respectively rules (T), (A), (N), (G) in the obvious way by requiring that communication primitives be executed only for a strictly positive duration. Moreover, rule (T0) states that telling a token for a zero duration succeeds by not updating the store. Rule (Wr) is the analogue of rule (D1).

[4] We extend classical arithmetic on natural numbers by $\infty - 1 = \infty$.

Definition 2.16 Define the transition rules for the $\mathcal{R}$ agents as rules (S), (P), (C), (T0), (Tr), (Ar), (Nr), (Gr), (Wr) of figure 2.

The operational semantics is defined by using an auxiliary relation $\longmapsto$, defined in a similar way as in the previous subsection. We shall subsequently write this semantics as $\mathcal{O}_r$.

Definition 2.17 Define the *operational semantics* $\mathcal{O}_r : \mathcal{R}(Srcom) \to \mathcal{P}(Sfstate)$ as the following function: For any timed agent A,

$$\mathcal{O}_r(A) = \{(\sigma^*, \delta^+) : \langle A \mid \emptyset\rangle \longmapsto^* \langle E \mid \sigma\rangle\} \\ \cup \{(\sigma^*, \delta^-) : \langle A \mid \emptyset\rangle \longmapsto^* \langle B \mid \sigma\rangle \not\longmapsto, B \neq E\}$$

where σ^* denotes the multiset of the tokens present in σ without their duration.

3 Intra-family comparison

3.1 Modular embedding

A natural question to ask is whether the time extensions we just introduced strictly increase the expressivness of the Linda language and, if so, whether some of the timed primitives may be expressed in terms of others.

A basic approach to answer that question has been given by Shapiro in [30] as follows. Consider two languages L and L'. Assume given the semantic mappings (*observation criteria*) $\mathcal{S} : L \to \mathcal{O}$ and $\mathcal{S}' : L' \to \mathcal{O}'$, where $\mathcal{O}$ and $\mathcal{O}'$ are some suitable domains. Then, according to [30], L can *embed* L' if there exists a mapping $\mathcal{C}$ (*coder*) from the statements of L' to the statements of L, and a mapping $\mathcal{D}^e$ (*decoder*) from $\mathcal{O}$ to $\mathcal{O}'$, such that $\mathcal{D}^e(\mathcal{S}(\mathcal{C}(A))) = \mathcal{S}'(A)$, for every statement $A \in L'$.

This approach is however too weak since, for instance, the above equation is satisfied by any pair of Turing-complete languages. To circumvent this problem, De Boer and Palamidessi have proposed in [13] to add three constraints on the coder $\mathcal{C}$ and on the decoder $\mathcal{D}^e$. First, $\mathcal{D}^e$ should be defined in an element-wise way w.r.t. $\mathcal{O}$:

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

for some appropriate mapping $\mathcal{D}^e{}_{el}$. Second, the coder $\mathcal{C}$ should be defined in

a compositional way w.r.t. the sequential, parallel and choice operators: [5]

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

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

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

where tm and tm' extract the information on termination from the observables of L and L', respectively. An embedding satisfying these properties (P_1, P_2, P_3) is said to be *modular*.

The existence of a modular embedding from L' into L is subsequently denoted by $L' \leq L$. It is easy to see that $\leq$ is a pre-order relation. Moreover if $L' \subseteq L$ then $L' \leq L$, that is, any language embeds all its sublanguages. This property descends immediately from the definition of embedding, by setting $\mathcal{C}$ and $\mathcal{D}^e$ equal to the identity function.

When two languages L and L' embed each other, they are said to be equivalent. This is denoted as $L \equiv L'$. Finally, we write $L' < L$ when $L' \leq L$ but $L \not\leq L'$

The study of the embedding in the $\mathcal{L}(X)$ family has been done in [6]. We can thus limit our exploration to the $\mathcal{D}$ and $\mathcal{R}$ families of languages.

3.2 The hierarchy of the languages with delay

We now turn to the $\mathcal{D}$ family of languages. A first result is that any language embeds all its sublanguages.

Proposition 3.1 *For all subsets X and Y of $\{ask, nask, get, tell\}$ such that $X \subseteq Y$, one has $\mathcal{D}(X) \leq \mathcal{D}(Y)$*

The primitive *nask* alone has no more power than *delay*.

Proposition 3.2 $\mathcal{D}(\emptyset) \equiv \mathcal{D}(nask)$.

The primitives *tell* and *ask* introduce new forms of computations, the first one by modifying the store and the second by introducing failures. While $\mathcal{D}(tell)$ and $\mathcal{D}(ask)$ are both strictly more powerful than $\mathcal{D}(\emptyset)$, they are not comparable to one another.

[5] Actually, this is only required for the parallel and choice operators in [13].

Proposition 3.3 *$\mathcal{D}(tell) \not\leq \mathcal{D}(\emptyset)$, $\mathcal{D}(ask) \not\leq \mathcal{D}(\emptyset)$, $\mathcal{D}(ask) \not\leq \mathcal{D}(tell)$, and $\mathcal{D}(tell) \not\leq \mathcal{D}(ask)$.*

Without the *tell* primitive, the store stays empty and the *get* and *nask* do not provide more power than the *ask* an *delay* primitives.

Proposition 3.4 *$\mathcal{D}(ask) \equiv \mathcal{D}(get) \equiv \mathcal{D}(ask, get) \equiv \mathcal{D}(ask, nask)$ $\equiv \mathcal{D}(nask, get) \equiv \mathcal{D}(ask, nask, get)$.*

We now consider the languages $\mathcal{D}(ask, tell)$ and $\mathcal{D}(nask, tell)$ obtained by extending $\mathcal{D}(tell)$ with the ability of checking the presence and the absence of data, respectively, in the dataspace. It is easy to establish that both $\mathcal{D}(ask, tell)$ and $\mathcal{D}(nask, tell)$ are strictly more expressive than $\mathcal{D}(tell)$.

Proposition 3.5 *$\mathcal{D}(ask, tell) \not\leq \mathcal{D}(tell)$ and $\mathcal{D}(nask, tell) \not\leq \mathcal{D}(tell)$.*

While $\mathcal{D}(ask, tell)$ extends stricly $\mathcal{D}(ask)$, $\mathcal{D}(nask, tell)$ is not comparable with $\mathcal{D}(ask)$.

Proposition 3.6 *For any $X \subseteq \{nask, get, tell\}$, one has*

(i) $\mathcal{D}(ask, tell) \not\leq \mathcal{D}(ask)$

(ii) $\mathcal{D}(nask, tell) \not\leq \mathcal{D}(ask)$

(iii) $\mathcal{D}(ask, X) \not\leq \mathcal{D}(nask, tell)$

Proof. Cases (i) and (ii) are easily proved by contradiction. For case (iii), let us proceed also by contradiction and assume that $\mathcal{D}(ask, X) \leq \mathcal{D}(nask, tell)$ and that the coder $\mathcal{C}$ and decoder $\mathcal{D}^e$ satisfy properties $P1$ to $P3$. The proof is based on the examination of the normal form of the coding of the primitives *delay* and *ask*.

First, consider $\mathcal{C}(delay(i))$ for $i > 0$. Since $\mathcal{C}(delay(i))$ is in $\mathcal{D}(nask, tell)$, its normal form can be written as

$$\begin{aligned}\mathcal{C}(delay(i)) = &(delay(j_1)\ ;\ A_1)\ +\ \ldots\ +\ (delay(j_n)\ ;\ A_n)\\ &+ (nask(t_1)\ ;\ B_1)\ +\ \ldots\ +\ (nask(t_m)\ ;\ B_m)\\ &+ (tell(s_1)\ ;\ C_1)\ +\ \ldots\ +\ (tell(s_l)\ ;\ C_l)\end{aligned}$$

for some times j_i's and tokens t_i's and s_i's, with $n, m, l \geq 0$. Our first observation is that the coding can not contain any choice starting with a *nask*, a *tell* or a $delay(0)$ primitive, i.e. $m = 0$, $l = 0$ and $j_k > 0 (1 \leq j \leq n)$. Indeed, if there is one choice starting with a *nask* primitive, then the coding of the agent $delay(i)\ +\ (delay(0)\ ;\ ask(t))$ accepts the following derivation

$$\langle \mathcal{C}(delay(i)\ +\ (delay(0)\ ;\ ask(t)) \mid \emptyset \rangle \rightarrow \langle B_1 \mid \emptyset \rangle$$

As $delay(i)$ succeeds on the empty store, the agent B_1 has to suceeds. This derivation provides then a valid prefix for a successful derivation of the agent. This contradicts, by property $P3$, the fact that $delay(i) \ + \ (delay(0) \ ; \ ask(t))$ has only failling computations on the empty store. The absence of choice starting with a *tell* and $delay(0)$ primitive can be shown similarly.

Consequently, the agent $\mathcal{C}(delay(i))$ for $i > 0$ has then a normal form of the following type: $\mathcal{C}(delay(i)) = (delay(j_1) \ ; \ A_1) \ + \ \ldots \ + \ (delay(j_n) \ ; \ A_n)$, where $j_k > 0 \ (1 \leq j \leq n)$.

A second observation about $\mathcal{C}(delay(i))$ is that the j_k's $(1 \leq k \leq n)$ are greater than i. This property can be proved by induction on i. For $i = 1$, it results from the first observation. Now consider any $delay(i)$, $delay(i+1)$ and their coding

$$\begin{aligned} \mathcal{C}(delay(i)) &= (delay(j_1) \ ; \ A_1) \ + \ \ldots \ + \ (delay(j_n) \ ; \ A_n) \\ \mathcal{C}(delay(i+1)) &= (delay(k_1) \ ; \ B_1) \ + \ \ldots \ + \ (delay(k_m) \ ; \ B_m) \end{aligned}$$

We denote by k_K the smallest k_l's $(1 \leq l \leq m)$. If k_K is less than any j_l, the coding of the agent $delay(i+1) \ + \ (delay(i) \ ; \ ask(t))$ accepts the following derivation

$$\begin{aligned} &\langle \mathcal{C}(delay(i+1) \ + \ (delay(i) \ ; \ ask(t)) \mid \emptyset \rangle \\ &\qquad \rightsquigarrow^{k_K} \langle \ldots \ + \ (delay(0) \ ; \ B_K) \ + \ \ldots \mid \emptyset \rangle \rightarrow \langle B_K \mid \emptyset \rangle \end{aligned}$$

As $delay(i+1)$ succeeds on the empty store, this derivation provides a valid prefix for a successful derivation of the agent. This contradicts, by property $P3$, the fact that $delay(i+1) \ + \ (delay(i) \ ; \ ask(t))$ has only failing computations on the empty store. Any k_l must then be strictly greater than at least one of the j_l. By the induction principle, any j_l is greater than i, and therefore any k_l is greater than $i+1$.

Secondly, observe the coding of an agent $ask(t)$. By observing the derivations of $\mathcal{C}(ask(t) \ + \ delay(0))$, we conclude in a similar way to $delay(i)$ that the agent $\mathcal{C}(ask(t))$ has a normal form of the following type:

$$\mathcal{C}(ask(t)) = (delay(t_1) \ ; \ A_1) \ + \ \ldots \ + \ (delay(t_m) \ ; \ A_m)$$

We are now in a position to establish a contradiction. Assume we have such a coding of $ask(t)$ and denote by t_i the minimum of the $t_k (1 \leq k \leq m)$. Now, the coding of $delay(t_i)$ is

$$\mathcal{C}(delay(t_i)) = (delay(j_1) \ ; \ B_1) \ + \ \ldots \ + \ (delay(j_n) \ ; \ B_n)$$

where $j_k \geq t_i (1 \leq j \leq n)$. The agent $delay(t_i) \ + \ ask(t)$ accepts the following derivation

$$\langle \mathcal{C}(delay(t_i) \ + \ ask(t)) \mid \emptyset \rangle \rightsquigarrow^{t_i} \langle \ldots \ + \ (delay(0) \ ; \ A_i) \ + \ \ldots \mid \emptyset \rangle \\ \rightarrow \ \langle A_i \mid \emptyset \rangle$$

As $ask(t)$ fails on the empty store, this derivation provides a valid prefix for a failing derivation of the agent. This contradicts, by property $P3$, the fact that $delay(t_i) \ + \ ask(t)$ has only successful computations on the empty store. □

Proposition 3.7 *$\mathcal{D}(nask, tell, X) \not\leq \mathcal{D}(ask, tell)$, for any $X \subseteq \{ask, get\}$.*

Proof. The proof, similar to that of the previous proposition, is based on the examination of the normal form of the coding of the primitives *delay* and *nask*. □

In the presence of *tell* and *get* primitives, *ask* primitive is redundant.

Proposition 3.8

(*i*) $\mathcal{D}(get, tell) \equiv \mathcal{D}(ask, get, tell)$

(*ii*) $\mathcal{D}(nask, get, tell) \equiv \mathcal{D}(ask, nask, get, tell)$

Proof. *(i).* The inequality $\mathcal{D}(get, tell) \leq \mathcal{D}(ask, get, tell)$ is obvious. The converse inequality is obtained by coding each *get*, *tell* and *delay* primitive by itself and each $ask(t)$ primitive by $get(t) \ ; \ tell(t)$.

(ii). The inequality $\mathcal{D}(nask, get, tell) \leq \mathcal{D}(ask, nask, get, tell)$ is immediate. To establish the converse inequality, we first code any token t by a pair of tokens which we denote (t_1,t_2). Note that this can be done because *Stoken* is enumerable: for instance, it is sufficient to associate the token associated with the integer n to the tokens associated with the integers $2n$ and $2(n+1)$. Given such a coding of tokens, we define the coder $\mathcal{C}$ as follows.

$$\mathcal{C}(ask(t)) = get(t_2) \ ; \ tell(t_2)$$

$$\mathcal{C}(nask(t)) = nask(t_1) \qquad\qquad \mathcal{C}(tell(t)) = tell(t_1) \ ; \ tell(t_2)$$

$$\mathcal{C}(get(t)) = get(t_2) \ ; \ get(t_1) \qquad\qquad \mathcal{C}(delay(n)) = delay(n)$$

Moreover, the decoder $\mathcal{D}^e$ is defined as follows: $\mathcal{D}^e_{el}((\sigma, \delta)) = (\overline{\sigma}, \delta)$ where $\overline{\sigma}$ is composed of the tokens t for which t_1 and t_2 are in σ, the multiplicity of occurrences of t being that of pairs (t_1, t_2) in σ. To conclude, it remains to establish that $\mathcal{O}_d(A) = \mathcal{D}^e(\mathcal{O}_d(\mathcal{C}(A)))$, for any agent A of $\mathcal{D}(ask, nask, get, tell)$. The key point for this proof consists of first establishing that if, A' denotes $\mathcal{C}(A)$, for any agent A, and, if σ' denotes the store obtained by coding the tokens of σ, for any store σ, then $\langle A \mid \sigma \rangle \longrightarrow \langle B \mid \tau \rangle$ if and only if $\langle A' \mid \sigma' \rangle \longrightarrow^* \langle B' \mid \tau' \rangle$,

for any agents A, B and any stores σ, τ. This in turn is proved by inductively reasoning on the structure of the agent A and for parallelly composed agents by reasoning on their normal forms. □

The language $\mathcal{D}(get, tell)$ happens to be strictly more expressive than $\mathcal{D}(ask, tell)$.

Proposition 3.9 *$\mathcal{D}(ask, tell) \leq \mathcal{D}(get, tell)$ and $\mathcal{D}(get, tell) \not\leq \mathcal{D}(ask, tell)$*

To be complete, we show now that, if a language contains the *tell* primitive, *nask* and *get* are incomparable. The following lemma will help us in this task.

Lemma 3.10 *For any agent A in $\mathcal{D}(ask, nask, tell)$, if $\langle A \mid \sigma\rangle \mapsto^* \langle B \mid \sigma \cup \tau\rangle$ then $\langle A \,||\, A \mid \sigma\rangle \mapsto^* \langle B \,||\, B \mid \sigma \cup \tau \cup \tau\rangle$ where $\cup$ denotes union on multisets.*

Proof. The proof is conducted by induction on the number of steps of the computation $\langle A \mid \sigma\rangle \mapsto^* \langle B \mid \sigma \cup \tau\rangle$. □

Proposition 3.11 *For any $X \subseteq \{ask, get\}$ and $Y \subseteq \{ask, nask\}$, one has*

(i) $\mathcal{D}(nask, tell, X) \not\leq \mathcal{D}(get, tell)$

(ii) $\mathcal{D}(get, tell, Y) \not\leq \mathcal{D}(ask, nask, tell)$

Proof. For case (i), the proof is similar to that of proposition 3.7. For case (ii), let us proceed by contradiction and assume that $\mathcal{D}(get, tell) \leq \mathcal{D}(ask, nask, tell)$. In that case, as $\mathcal{O}_d(tell(t) \;;\; get(t)) = \{(\emptyset, \delta^+)\}$ any computation of $A = \mathcal{C}(tell(t)) \;;\; \mathcal{C}(get(t))$ starting in the empty store is successful by P_3. By lemma 3.10, there is a computation of $B = \mathcal{C}(tell(t)) \;;\; (\mathcal{C}(get(t)) \;||\; \mathcal{C}(get(t)))$ starting in the empty store that is successful, which contradicts, by P_2 and P_3, the fact that $\mathcal{O}_d(tell(t) \;;\; (get(t) \;||\; get(t))) = \{(\emptyset, \delta^-)\}$. □

3.3 The hierarchy of the languages with relative duration

As expected, the first result for the $\mathcal{R}$ family of languages is that any language embeds all its sublanguages.

Proposition 3.12 *For all subsets X and Y of $\{ask, nask, get, tell\}$ such that $X \subseteq Y$, one has $\mathcal{R}(X) \leq \mathcal{R}(Y)$*

The primitives *tell* and *ask* respectively introduce the possibility of modifying the store and the getting of failures. While $\mathcal{R}(tell)$ and $\mathcal{R}(ask)$ are both strictly more expressive than $\mathcal{R}(\emptyset)$, they are not comparable to one another.

Proposition 3.13 *For any* $X \subseteq \{nask, get, tell\}$*, one has* $\mathcal{R}(tell) \not\leq \mathcal{R}(\emptyset)$*,* $\mathcal{R}(ask) \not\leq \mathcal{R}(\emptyset)$*,* $\mathcal{R}(tell) \not\leq \mathcal{R}(ask)$*, and* $\mathcal{R}(ask, X) \not\leq \mathcal{R}(tell)$*.*

On the empty store, the *ask* and *get* primitives have the same behaviour.

Proposition 3.14 $\mathcal{R}(ask) \equiv \mathcal{R}(get) \equiv \mathcal{R}(ask, get)$*.*

In the $\mathcal{R}$ family, the *nask* primitive alone is strictly more expressive than the *ask* primitive alone.

Proposition 3.15 $\mathcal{R}(ask) \leq \mathcal{R}(nask)$ *and* $\mathcal{R}(nask) \not\leq \mathcal{R}(ask)$*.*

The *ask* and *get* primitives do not add any expressive power to $\mathcal{R}(nask)$.

Proposition 3.16 $\mathcal{R}(nask) \equiv \mathcal{R}(ask, nask) \equiv \mathcal{R}(nask, get) \equiv \mathcal{R}(ask, nask, get)$*.*

While $\mathcal{R}(nask)$ is strictly more expressive than $\mathcal{R}(ask)$, it is still incomparable to $\mathcal{R}(tell)$.

Proposition 3.17 *For any* $X \subseteq \{ask, get, tell\}$ *and* $Y \subseteq \{ask, nask, get\}$*, one has* $\mathcal{R}(nask, X) \not\leq \mathcal{R}(tell)$ *and* $\mathcal{R}(tell, Y) \not\leq \mathcal{R}(nask)$*.*

$\mathcal{R}(ask, tell)$ turns out to be strictly more expressive than $\mathcal{R}(nask)$.

Proposition 3.18 $\mathcal{R}(nask) \leq \mathcal{R}(ask, tell)$ *and* $\mathcal{R}(ask, tell) \not\leq \mathcal{R}(nask)$*.*

While $\mathcal{R}(ask, tell)$ and $\mathcal{R}(nask, tell)$ are both strictly more powerful than $\mathcal{R}(tell)$ and $\mathcal{R}(nask)$, they are incomparable.

Proposition 3.19 *For any* $X \subseteq \{nask, get\}$ *and* $Y \subseteq \{ask, get\}$*, one has* $\mathcal{R}(ask, tell, X) \not\leq \mathcal{R}(nask, tell)$ *and* $\mathcal{R}(nask, tell, Y) \not\leq \mathcal{R}(ask, tell)$*.*

The primitives $\{get, tell\}$ are strictly more expressive than the pair of primitives $\{ask, tell\}$. Moreover adding *ask* to $\mathcal{R}(get, tell)$ does not yield in an additional expressiveness.

Proposition 3.20

(*i*) $\mathcal{R}(ask, tell) \leq \mathcal{R}(get, tell)$

(*ii*) $\mathcal{R}(get, tell) \not\leq \mathcal{R}(ask, tell)$

(*iii*) $\mathcal{R}(get, tell) \equiv \mathcal{R}(ask, get, tell)$

Proof. *(i).* Because of the infinite enumerability of the tokens, we associate with each token t a pair of tokens that, for simplicity, we denote t_f and t_i. Intuitively, they correspond to a token t on the store with, a finite or infinite duration, respectively. As there is no *nask* primitives, decreasing the duration of finite tokens in the transitions will occur only in case of failing computation.

In this context, there will be temporal transitions until the current store σ satisfies $\sigma^- = \sigma$, i.e. until all tokens with finite duration disappear.

We can then define the coder $\mathcal{C}$ as follows, with $d_1, d_2 > 0$ and with d_1 finite.

$$\mathcal{C}(tell_0(t)) = tell_0(t) \qquad \mathcal{C}(tell_{d_1}(t)) = tell_\infty(t_f)$$
$$\mathcal{C}(tell_\infty(t)) = tell_\infty(t_i) \qquad \mathcal{C}(ask_0(t)) = ask_0(t)$$
$$\mathcal{C}(ask_{d_2}(t)) = (get_1(t_f)\ ;\ tell_\infty(t_f))\ +\ (get_1(t_i)\ ;\ tell_\infty(t_i))$$

The associated decoder $\mathcal{D}^e$ is defined by :

$$\mathcal{D}^e((\sigma,\delta)) = \begin{cases} (\sigma_\infty, \delta^-) & \text{if } \delta = \delta^- \\ (\sigma_f, \delta^-) & \text{if } \delta = \delta^+ \end{cases}$$

where $\sigma_\infty = \{t : t_i \in \sigma\}$ and $\sigma_f = \{t : t_i \in \sigma \vee t_f \in \sigma\}$.

(ii). Assume that $\mathcal{R}(get, tell) \leq \mathcal{R}(ask, tell)$ and consider $tell_1(a)\ ;\ get_1(a)$. Since $\mathcal{C}$ is compositional and since $\mathcal{O}_r(tell_1(a)\ ;\ get_1(a)) = \{(\emptyset, \delta^+)\}$, the termination mark of any element of $\mathcal{O}_r(\mathcal{C}(tell_1(a))\ ;\ \mathcal{C}(get_1(a)))$ is successful. As $\mathcal{C}(get_1(a))$ is composed of *ask* and *tell* primitives only and since *ask*, *tell* primitives do not destroy elements, it follows that any element of $\mathcal{O}_r(\mathcal{C}(tell_1(a))\ ;\ \mathcal{C}(get_1(a))\ ;\ \mathcal{C}(get_1(a)))$ has a successful termination mark. However, $\mathcal{O}_r(tell_1(a)\ ;\ get_1(a)\ ;\ get_1(a)) = \{(\emptyset, \delta^-)\}$ which contradicts property P_3.

(iii). The inequality $\mathcal{R}(get, tell) \leq \mathcal{R}(ask, get, tell)$ follows directly from the inclusion of languages. To prove the converse inequality, we consider the coder of point (i), extended by $\mathcal{C}(get_0(t)) = get_0(t)$ and $\mathcal{C}(get_{d_2}(t)) = get_1(t_f)\ +\ get_1(t_i)$ □

Moreover the languages $\mathcal{R}(ask, nask, tell)$ and $\mathcal{R}(get, tell)$ are incomparable. To establish this property, we introduce an auxiliary lemma.

Lemma 3.21 *For any agent A in $\mathcal{R}(ask, nask, tell)$, if $\langle A \mid \sigma\rangle \mapsto^* \langle B \mid \tau\rangle$ then for some $\tau' \subseteq \tau$: $\langle A \mid\mid A \mid \sigma\rangle \mapsto^* \langle B \mid\mid B \mid \tau \cup \tau'\rangle$ where $\cup$ denotes union on multisets.*

Proposition 3.22 *For any $X \subseteq \{ask, get\}$ and $Y \subseteq \{ask, nask\}$,*

(i) $\mathcal{R}(get, tell, Y) \not\leq \mathcal{R}(ask, nask, tell)$

(ii) $\mathcal{R}(nask, tell, X) \not\leq \mathcal{R}(get, tell)$

Proof. *(i).* Let us proceed by contradiction and assume that $\mathcal{R}(get, tell) \leq \mathcal{R}(ask, nask, tell)$. In that case, as $\mathcal{O}_r(tell_1(t) \ ; \ get_1(t)) = \{(\emptyset, \delta^+)\}$, any computation of $A = \mathcal{C}(tell_1(t)) \ ; \ \mathcal{C}(get_1(t))$ starting in the empty store is successful by P_3. By lemma 3.21, there is a computation of $B = \mathcal{C}(tell_1(t)) \ ; \ (\mathcal{C}(get_1(t)) \ || \ \mathcal{C}(get_1(t)))$ starting in the empty store that is successful, which contradicts, by P_2 and P_3, the fact that $\mathcal{O}_r(tell_1(t) \ ; \ (get_1(t) \ || \ get_1(t))) = \{(\emptyset, \delta^-)\}$.

(ii). The proof similar to that of proposition 3.19 (ii). □

4 Inter-family comparisons

4.1 Comparing the $\mathcal{L}$ and $\mathcal{D}$ families

The comparison between the $\mathcal{L}$ and $\mathcal{D}$ families is substantiated by the results presented in this section together with those established in [19] and in section 3.2.

The main observation here is that – except in the case of *nask* primitive alone – the $\mathcal{D}(X)$ language is strictly more expressive than the corresponding $\mathcal{L}(X)$ and no $\mathcal{L}$ is more expressive than a $\mathcal{D}$ language. In other words, the *delay* primitive can not be expressed by (any combination of) the other primitives.

The first result is that, as intuitively expected, for the same set of primitives X, the language $\mathcal{D}(X)$ is more powerful than $\mathcal{L}(X)$.

Proposition 4.1 *For any $X \subseteq \{ask, nask, get, tell\}$, $\mathcal{L}(X) \leq \mathcal{D}(X)$.*

The only member of the $\mathcal{D}$ family equivalent to a $\mathcal{L}(X)$ is $\mathcal{D}(nask)$.

Proposition 4.2 *One has $\mathcal{L}(nask) \equiv \mathcal{D}(nask)$.*

If the language contains at least one primitive other than *nask*, the *delay* primitive cannot be expressed in any $\mathcal{L}(X)$.

Proposition 4.3 *For any $X, Y \subseteq \{ask, nask, get, tell\}$, if X contains at least one primitive other than nask, then $\mathcal{D}(X) \not\leq \mathcal{L}(Y)$.*

Proof. There are three cases to consider, where the primitive in X other than *nask* is *ask*, *get* or *tell*.

Case 1: $ask \in X$. Consider the agents $A = delay(0)$ and $B = delay(1) \ ; \ ask(t)$. The agent $A \ + \ B$ is in $\mathcal{D}(X)$ and $\mathcal{O}_d(A \ + \ B) = \{(\emptyset, \delta^+)\}$.

We proceed by contradiction. Assume that $\mathcal{D}(X) \leq \mathcal{L}(Y)$ and that there is a coder $\mathcal{C}$ from agents of $\mathcal{D}(X)$ to agents of $\mathcal{L}(Y)$. We shall establish that

$\mathcal{O}(\mathcal{C}(A + B))$ contains a failing computation which is imposssible in view of property $P3$.

By property $P2$, one has $\mathcal{C}(delay(1) ; ask(t)) = \mathcal{C}(delay(1)) ; \mathcal{C}(ask(t))$. As $\mathcal{O}_d(delay(1)) = \{(\emptyset, \delta^+)\}$ and $\mathcal{O}_d(delay(1) ; ask(t)) = \{(\emptyset, \delta^-)\}$, one should have $\langle \mathcal{C}(B) \mid \emptyset \rangle \longrightarrow \langle T \mid \tau \rangle$ for some agent $T \in \mathcal{L}(Y)$ and some store τ, with $\langle T \mid \tau \rangle$ leading to a failing computation. By property $P2$, one has $\mathcal{C}(A + B) = \mathcal{C}(A) + \mathcal{C}(B)$. The thesis then results from the fact that $\langle \mathcal{C}(A) + \mathcal{C}(B) \mid \emptyset \rangle \longrightarrow \langle T \mid \tau \rangle$ is a valid computation prefix of $\mathcal{C}(A + B)$ which leads to a failing computation.

Case 2: $get \in X$. This case is treated as the first one by considering the agents $A = delay(0)$ and $B = delay(1) ; get(t)$.

Case 3: $tell \in X$. Consider the agents $A = tell(a)$ and $B = delay(1) ; tell(b)$. The agent $A + B$ is in $\mathcal{D}(X)$ and $\mathcal{O}_d(A + B) = \{(\{a\}, \delta^+)\}$.

We proceed by contradiction. Assume that $\mathcal{D}(X) \leq \mathcal{L}(Y)$ and that there is a coder $\mathcal{C}$ from agents of $\mathcal{D}(X)$ to agents of $\mathcal{L}(Y)$ and a decoder $\mathcal{D}^e$ which satisfies the constraints $P1$ to $P3$.

The definition of coder and decoder gives:

$$\mathcal{D}^e(\mathcal{O}(\mathcal{C}(A))) = \mathcal{O}_d(A) = \{(\{a\}, \delta^+)\}$$
$$\mathcal{D}^e(\mathcal{O}(\mathcal{C}(B))) = \mathcal{O}_d(B) = \{(\{b\}, \delta^+)\}$$

As all the computations of $\mathcal{C}(A)$ and $\mathcal{C}(B)$ are succesfull, $\mathcal{O}(\mathcal{C}(A) + \mathcal{C}(B)) = \mathcal{O}(\mathcal{C}(A)) \cup \mathcal{O}(\mathcal{C}(B))$. Properties $P1$ and $P3$ then give

$$\begin{aligned}\mathcal{D}^e(\mathcal{O}(\mathcal{C}(A + B))) &= \mathcal{D}^e(\mathcal{O}(\mathcal{C}(A) + \mathcal{C}(B))) \\ &= \mathcal{D}^e(\mathcal{O}(\mathcal{C}(A)) \cup \mathcal{O}(\mathcal{C}(B))) \\ &= \mathcal{D}^e(\mathcal{O}(\mathcal{C}(A))) \cup \mathcal{D}^e(\mathcal{O}(\mathcal{C}(B))) \\ &= \{(\{a\}, \delta^+), (\{b\}, \delta^+)\}\end{aligned}$$

□

In addition to the $\mathcal{L}(X) \leq \mathcal{D}(X)$ inclusions of property 4.1, one has the quite unexpected following inequality.

Proposition 4.4 $\mathcal{L}(ask, nask) \leq \mathcal{D}(nask, tell)$.

In the rest of the section we show that there is no other inclusions between those two hierarchies. This corresponds to the fact that, except for the empty store, the *delay* primitive is not able to express any other primitive.

Proposition 4.5 *For any $X \subseteq \{ask, nask, get, tell\}$, if $X \not\subseteq \{nask\}$ then $\mathcal{L}(X) \not\leq \mathcal{D}(nask)$*

Proposition 4.6 *$\mathcal{L}(ask, X) \not\leq \mathcal{D}(tell)$, $\mathcal{L}(get, Y) \not\leq \mathcal{D}(tell)$, and $\mathcal{L}(nask, tell) \not\leq \mathcal{D}(tell)$, for any $X \subseteq \{nask, get, tell\}$, $Y \subseteq \{ask, nask, tell\}$.*

Proposition 4.7 *$\mathcal{L}(tell, X) \not\leq \mathcal{D}(ask)$, for any $X \subseteq \{ask, nask, get\}$.*

Proposition 4.8 *$\mathcal{L}(ask, tell, X) \not\leq \mathcal{D}(nask, tell)$ and $\mathcal{L}(get, tell, Y) \not\leq \mathcal{D}(nask, tell)$, for any $X \subseteq \{nask, get\}$, $Y \subseteq \{ask, nask\}$.*

Proposition 4.9 *$\mathcal{L}(nask, tell, X) \not\leq \mathcal{D}(ask, tell)$ and $\mathcal{L}(get, tell, Y) \not\leq \mathcal{D}(ask, tell)$, for any $X \subseteq \{ask, get\}$, $Y \subseteq \{ask, nask\}$.*

Proposition 4.10 *For any $X \subseteq \{ask, nask\}$, $Y \subseteq \{ask, get\}$,*

(i) $\mathcal{L}(get, tell, X) \not\leq \mathcal{D}(ask, nask, tell)$

(ii) $\mathcal{L}(tell, nask, Y) \not\leq \mathcal{D}(get, tell)$

Proof. *(i).* We proceed by contradiction and assume that $\mathcal{L}(get, tell) \leq \mathcal{D}(ask, nask, tell)$. In that case, as $\mathcal{O}(tell(t)\ ;\ get(t)) = \{(\emptyset, \delta^+)\}$, any computation of $A = \mathcal{C}(tell(t))\ ;\ \mathcal{C}(get(t))$ starting with the empty store is successful by P_3. Lemma 3.10 gives that, as a consequence, there is a computation of $B = \mathcal{C}(tell(t))\ ;\ (\mathcal{C}(get(t))\ ||\ \mathcal{C}(get(t)))$ starting with the empty store that is successful, which contradicts, by P_2 and P_3, the fact that $\mathcal{O}(tell(t)\ ;\ (get(t)\ ||\ get(t))) = \{(\emptyset, \delta^-)\}$.

(ii). Again we proceed by contradiction. Otherwise, $\mathcal{C}(tell(a))\ ;\ \mathcal{C}(nask(a))$ has only successful computations, which, by P_3, contradicts the fact that $\mathcal{O}(tell(a)\ ;\ nask(a)) = \{(\{a\}, \delta^-)\}$. Indeed, since $\mathcal{O}(tell(a)) = \{(\{a\}, \delta^+)\}$, by P_3 any computation of $\mathcal{C}(tell(a))$ (starting with the empty store) is successful. Similarly, it follows from $\mathcal{O}(nask(a)) = \{(\emptyset, \delta^+)\}$ that any computation starting with the empty store is successful, and consequently, so is any computation starting from any store, since $\mathcal{C}(nask(a))$ is composed of get, tell and delay primitives. Summing up, any (successful) computation of $\mathcal{C}(tell(a))$ starting with the empty store can be continued by a (successful) computation of $\mathcal{C}(nask(a))$. □

4.2 Comparing the $\mathcal{L}$ and $\mathcal{R}$ families

The first result in the comparison between the $\mathcal{L}(X)$ and $\mathcal{R}(X)$ families is that, as intuitively expected, for the same set of primitives X, the language $\mathcal{R}(X)$ is more expressive than the language $\mathcal{L}(X)$.

Proposition 4.11 *$\mathcal{L}(X) \leq \mathcal{R}(X)$, for any $X \subseteq \{ask, nask, get, tell\}$.*

The two empty languages are equivalent.

Proposition 4.12 *$\mathcal{L}(\emptyset) \equiv \mathcal{R}(\emptyset)$.*

The languages $\mathcal{L}(X)$ and $\mathcal{R}(X)$ do not differ if X contains only one of the primitives *ask*, *get* or *tell*.

Proposition 4.13

$$\mathcal{L}(ask) \equiv \mathcal{L}(get) \equiv \mathcal{L}(ask, get) \equiv \mathcal{R}(ask) \equiv \mathcal{R}(get) \equiv \mathcal{R}(ask, get)$$
$$\mathcal{L}(tell) \equiv \mathcal{R}(tell)$$

Unlike the other primitives *nask* is sufficient to distinguish $\mathcal{L}(nask)$ and $\mathcal{R}(nask)$.

Proposition 4.14 $\mathcal{R}(nask) \not\leq \mathcal{L}(nask)$.

The pairs of primitives $(ask, nask)$, $(nask, get)$, $(ask, tell)$ and $(get, tell)$ do not dinstinguish the languages $\mathcal{L}$ and $\mathcal{R}$.

Proposition 4.15

$$\mathcal{L}(ask, nask) \equiv \mathcal{R}(ask, nask)$$
$$\mathcal{L}(get, nask) \equiv \mathcal{R}(get, nask)$$
$$\mathcal{L}(ask, tell) \equiv \mathcal{R}(ask, tell)$$
$$\mathcal{L}(get, tell) \equiv \mathcal{R}(get, tell)$$

The pair of primitives *nask*, *tell* distinguish the two families $\mathcal{L}$ and $\mathcal{R}$.

Proposition 4.16 $\mathcal{R}(nask, tell, X) \not\leq \mathcal{L}(Y)$, *for any* $X \subseteq \{ask, get\}$ *and* $Y \subseteq \{ask, nask, get, tell\}$,

4.3 Comparing the $\mathcal{D}$ and $\mathcal{R}$ families

We finally compare the $\mathcal{D}$ and $\mathcal{R}$ families. The first main observation is that the *delay* primitive cannot be expressed in any $\mathcal{R}(X)$ language. The second one is that, when $\mathcal{R}(X)$ is more expressive than $\mathcal{L}(X)$ – i.e. if $\{nask, tell\} \subseteq X$ – the corresponding $\mathcal{D}(X)$ and $\mathcal{R}(X)$ languages are not comparable. The only member of the $\mathcal{D}(X)$ languages that is less powerful than some $\mathcal{R}(Y)$ is $\mathcal{D}(nask)$.

Proposition 4.17 $\mathcal{D}(nask) \leq \mathcal{R}(nask)$, $\mathcal{R}(nask) \not\leq \mathcal{D}(nask)$, $\mathcal{D}(nask) \leq \mathcal{R}(tell)$, *and* $\mathcal{R}(tell) \not\leq \mathcal{D}(nask)$

If a language contains at least one of the primitives *ask*, *get* and *tell*, the *delay* primitive cannot be expressed in any $\mathcal{R}(X)$ language.

Proposition 4.18 *For any* $X, Y \subseteq \{ask, nask, get, tell\}$ *such that* $X \cap \{ask, get, tell\} \neq \emptyset$, *one has* $\mathcal{D}(X) \not\leq \mathcal{R}(Y)$

Proof. The proof is conducted according to the inequality $X \cap \{ask, get, tell\} \neq \emptyset$, which naturally leads to three cases: $ask \in X$, $get \in X$, $tell \in X$.

Case 1. $ask \in X$. For contradiction, suppose that $\mathcal{D}(X) \leq \mathcal{R}(Y)$ and consider the coder $\mathcal{C}$ and the decoder $\mathcal{D}^e$ which satisfy the properties $P1$ to $P3$.

By property $P3$, the coding of $delay(1)$ has only successful computations on the empty set. The first step of any such computation corresponds to the execution of a $tell_d(t)$ or $nask_d(t)$ primitive on the empty set and thus is not a temporal step. Any computation can be represented as follow.

$$\langle \mathcal{C}(delay(1)) \mid \emptyset \rangle \rightarrow \langle C' \mid \sigma \rangle \mapsto^* \langle E \mid \tau \rangle$$

where $\mathcal{D}^e((\tau, \delta^+)) = (\emptyset, \delta^+)$.

We now consider the agent $delay(1)$; $ask(t)$. By property $P3$, its coding has only failing computations. Property $P2$ then gives that the following computation is a valid prefix for a failing computation.

$$\langle \mathcal{C}(delay(1)) \,;\, \mathcal{C}(ask(t)) \mid \emptyset \rangle \rightarrow \langle C' \,;\, \mathcal{C}(ask(t)) \mid \sigma \rangle \mapsto^* \langle \mathcal{C}(ask(t)) \mid \tau \rangle$$

As the first step is not a temporal transition, this gives, by definition of $+$, a valid prefix for a failing computation of the coding of the agent $delay(0) + (delay(1) \,;\, ask(t))$. That contradicts the fact that, by $P3$, this agent has only successful computations.

Case 2. $get \in X$. It is sufficient to replace ask by get in the previous proof.

Case 3. $tell \in X$. We proceed by contradiction as for the first case and consider the agent $delay(1)$; $tell(t)$. By property $P3$, its coding has only successful computations. By property $P2$, such a computation begin with a successful computation of $\mathcal{C}(delay(1))$ and goes on with a successful computation of $\mathcal{C}(tell(t))$. One thus has

$$\langle \mathcal{C}(delay(1)) \,;\, \mathcal{C}(tell(t)) \mid \emptyset \rangle \rightarrow \langle C' \,;\, \mathcal{C}(tell(t)) \mid \sigma \rangle \mapsto^* \langle \mathcal{C}(tell(t)) \mid \tau \rangle \mapsto^* \langle E \mid \mu \rangle$$

where $\mathcal{D}^e((\mu, \delta^+)) = (\{t\}, \delta^+)$.

As the first step is not a temporal transition, this gives by the definition of $+$ a valid successful computation of the coding of the agent $delay(0) + (delay(1) \,;\, tell(t))$. This contradicts the fact that any computation of this agent finishes on the empty set. □

We have argued in section 4.2 that if X is a set of primitives that is not the $\{nask\}$ singleton and does not contain the pair $\{nask, tell\}$, the languages $\mathcal{L}(X)$ and $\mathcal{R}(X)$ are equivalent. The following results are the transcription of the comparison of $\mathcal{D}(X)$ and $\mathcal{L}(X)$ languages of section 4.1 in the comparison between the $\mathcal{D}(X)$ and $\mathcal{R}(X)$ languages due to these equivalences.

Proposition 4.19

$$\mathcal{R}(\emptyset) \leq \mathcal{D}(\emptyset) \qquad \mathcal{R}(ask, tell) \leq \mathcal{D}(ask, tell)$$
$$\mathcal{D}(\emptyset) \not\leq \mathcal{R}(\emptyset) \qquad \mathcal{R}(ask, tell) \not\leq \mathcal{D}(ask)$$
$$\mathcal{R}(tell) \leq \mathcal{D}(tell) \qquad \mathcal{R}(ask, tell) \not\leq \mathcal{D}(nask, tell)$$
$$\mathcal{R}(nask) \leq \mathcal{D}(ask) \qquad \mathcal{R}(get, tell) \leq \mathcal{D}(get, tell)$$
$$\mathcal{R}(nask) \leq \mathcal{D}(nask, tell) \qquad \mathcal{R}(get, tell) \not\leq \mathcal{D}(ask, nask, tell)$$

The *nask* and *tell* primitives taken together cannot be expressed in any language in the $\mathcal{D}$ family.

Proposition 4.20 *$\mathcal{R}(nask, tell, X) \not\leq \mathcal{D}(Y)$, for any $X \subseteq \{ask, get\}$ and $Y \subseteq \{ask, nask, get, tell\}$.*

Proof. Since $\mathcal{D}(ask, nask, get, tell) \equiv \mathcal{D}(nask, get, tell)$, it is sufficient to prove that $\mathcal{R}(nask, tell, X) \not\leq \mathcal{D}(nask, get, tell)$.

For contradiction, suppose that $\mathcal{R}(nask, tell, X) \leq \mathcal{D}(nask, get, tell)$ and consider the coder $\mathcal{C}$ and decoder $\mathcal{D}^e$ satisfying properties $P1$ to $P3$. The proof is based on the examination of the normal form of the coding of the primitives *nask*.

For any $i \in Stime$, the agent $\mathcal{C}(nask_i(t))$ is in $\mathcal{D}(ask, nask, get, tell)$, and its normal form can then be written as

$$\begin{aligned}\mathcal{C}(nask_i(t)) = {} & (delay(j_1)\ ;\ A_1)\ +\ \ldots\ +\ (delay(j_n)\ ;\ A_n) \\ & +\ (nask(t_1)\ ;\ B_1)\ +\ \ldots\ +\ (nask(t_m)\ ;\ B_m) \\ & +\ (get(u_1)\ ;\ C_1)\ +\ \ldots\ +\ (get(u_l)\ ;\ C_l) \\ & +\ (tell(v_1)\ ;\ D_1)\ +\ \ldots\ +\ (tell(v_k)\ ;\ D_k)\end{aligned}$$

where $n, m, l, k \geq 0$.

Our first observation is that the coding can not contain any choice starting with a *tell* or a $delay(0)$ primitive, i.e. $k = 0$ and $j_x > 0$ $(1 \leq x \leq n)$. Indeed, if there is one choice starting with a *tell* primitive, then the coding of the agent $tell_{i+1}(t)\ ;\ (nask_i(t)\ +\ nask_1(s))$ accepts the following derivation

$$\begin{aligned}&\langle \mathcal{C}(tell_{i+1}(t)\ ;\ (nask_i(t)\ +\ nask_1(s)) \mid \emptyset\rangle \\ &\quad \mapsto^* \langle \mathcal{C}(nask_i(t)\ +\ nask_1(s)) \mid \sigma t_{i+1}\rangle \rightarrow \langle D_1 \mid \sigma t_{i+1} \cup \{v_1\}\rangle\end{aligned}$$

As the computation of $tell_{i+1}(t)\ ;\ nask_i$ fails, this derivation provides a valid prefix for a failing derivation of the agent. That contradicts, by property $P3$, the fact that $tell_{i+1}(t)\ ;\ (nask_i(t)\ +\ nask_1(s))$ has only successful computa-

tions on the empty store. The absence of an alternative in the choice starting with a $delay(0)$ primitive can be shown similarly.

Now, denote by σt_j ($j \in Stime$) any store such that

$$\langle \mathcal{C}(tell_j(t)) \mid \emptyset \rangle \mapsto^* \langle E \mid \sigma t_j \rangle.$$

The second observation is that any of the $nask(t_k)$ $(k = 1, \ldots, m)$ and $get(u_k)$ $(k = 1, \ldots, l)$ primitives appearing in the coding of $nask(i)$ fails on any $\sigma t_j (j \in Stime)$. Indeed if there is a $nask(t_K)$ that succeeds with σt_J, the coding of the agent $tell_J(t)\ ;\ (nask_i(t)\ +\ nask_1(s))$ has the following derivation

$$\begin{aligned}\langle \mathcal{C}(tell_J(t)\ ;\ (nask_i(t)\ +\ nask_1(s)) \mid \emptyset \rangle &\mapsto^* \langle \mathcal{C}(nask_i(t)\ +\ nask_1(s)) \mid \sigma t_J \rangle \\ &\rightarrow \langle B_K \mid \sigma t_J \rangle\end{aligned}$$

On the one hand, if $J \leq i$, $tell_J(t)\ ;\ nask_i(t)$ fails and this provides a valid prefix for a failing derivation of the agent. On the other hand, if $J > i$, this derivation provides a successful derivation with a final configuration decoded as $(\delta^+, \emptyset)$. Both cases contradict, by property $P3$, the fact that the semantics of $tell_{i+1}(t)\ ;\ (nask_i(t)\ +\ nask_1(s))$ is $\{(\delta^+, \{v\})\}$. The absence of successful $get(u_k)$ on σt_j can be shown similarly.

The third observation is about the *delay* primitives appearing in the coding. None of the $j_1, \ldots, j_l > 0$ can have 1 as value. Indeed, if $j_J = 1$, in view of our second observation, the coding of the agent $tell_i(t)\ ;\ (nask_i(t)\ +\ nask_{i+1}(t))$ accepts the following derivation

$$\begin{aligned}&\langle \mathcal{C}(tell_i(t)\ ;\ (nask_i(t)\ +\ nask_{i+1}(t)) \mid \emptyset \rangle \\ &\mapsto^* \langle \mathcal{C}(nask_i(t)\ +\ nask_{i+1}(v)) \mid \sigma t_i \rangle \\ &\rightsquigarrow \langle \ldots\ +\ (delay(0)\ ;\ A_J)\ +\ \ldots \mid \sigma t_i{}^- \rangle \\ &\rightarrow \langle A_J \mid \sigma t_i{}^- \rangle\end{aligned}$$

As $tell_i(t)\ ;\ nask_i(t)$ fails, this derivation provides a valid prefix for a failing derivation of the agent. That contradicts, by property $P3$, the fact that $tell_{i+1}(t)\ ;\ (nask_i(t)\ +\ nask_{i+1}(t))$ has only successful computations on the empty store.

An inductive reasoning leads similarly to the property that no value of *Stime* is possible for t_j.

All these observations together lead to the fact that the coding of a $nask_i(t)$ primitive has a normal form of the following type:

$$\begin{aligned}\mathcal{C}(nask_i(t)) = &(nask(t_1)\ ;\ B_1)\ +\ \ldots\ +\ (nask(t_m)\ ;\ B_m) \\ &+ (get(u_1)\ ;\ C_1)\ +\ \ldots\ +\ (get(u_l)\ ;\ C_l)\end{aligned}$$

where every $nask(t_k)(k = 1, \ldots, m)$ and $get(u_k)(k = 1, \ldots, l)$ primitive appearing in the coding of $nask(i)$ fails on any $\sigma t_j (j \in Stime)$. Consequently, $\langle \mathcal{C}(tell_1(t) \ ; \ nask_2(t)) \mid \emptyset \rangle \mapsto^* \langle \mathcal{C}(nask_2(t)) \mid \sigma t_1 \rangle$ is a valid prefix for a failling computation of $\mathcal{C}(tell_1(t) \ ; \ nask_2(t))$. However, this contraditcs, by property $P3$, the fact that $tell_1(t) \ ; \ nask_2(t)$ has only successful computations. □

5 Conclusion

In this paper we studied two extensions of Linda in order to introduce relative time in coordination languages. Both are based on the two-phase functioning approach to real-time systems already employed by languages such as Lustre ([10]) and Esterel ([3]).

The resulting families of languages have been described by means of transition systems written in Plotkin's style. Their expressiveness has been studied by means of the concept of modular embedding introduced in [13]. The complete expressiveness hierarchy of each family has been examined. We have also compared the expressiveness of languages of different families. All these results are summed up in figure 3. On the point of notations, an arrow from a language $\mathcal{L}_1$ to a language $\mathcal{L}_2$ means that $\mathcal{L}_2$ strictly embeds $\mathcal{L}_1$, that is $\mathcal{L}_1 < \mathcal{L}_2$.

This paper is a continuation of our previous work [19]. There the families $\mathcal{D}$ and $\mathcal{R}$ were introduced and some expressiveness results were presented. However, this paper presents a much deeper study of the expressiveness of $\mathcal{D}$ and $\mathcal{R}$. In particular, none of the results of sections 3 and 4.2 have appeared in [19]. Moreover, the comparisons of sections 4.1 and 4.3 have lead here to 13 propositions whereas only 5 were presented in [19].

Other related proposals for the introduction of time in coordination-like languages are [27] and [28]. Both pieces of work concern concurrent constraint languages ([29]), which may be viewed as a variant of Linda restricted to two communication primitives putting information in a tuple space and checking the presence of information in it. Technically, concurrent constraint languages can thus be viewed as the language $\mathcal{L}(\{ask, tell\})$. The paper [27] introduces time in this context by identifying quiescent points in the computation where no new information is introduced and by providing an operator for delaying computations by one unit. At each quiescent point in time, the tuple space is reinitialized to an empty content. The paper [28] extends this framework, on the one hand, by introducing a primitive for checking the absence of information and reacting on this absence during the same unit of time and, on the other hand, by generalizing the $delay(1)$ mechanism in a *hence* A construct which states that A holds at every instant after the considered time.

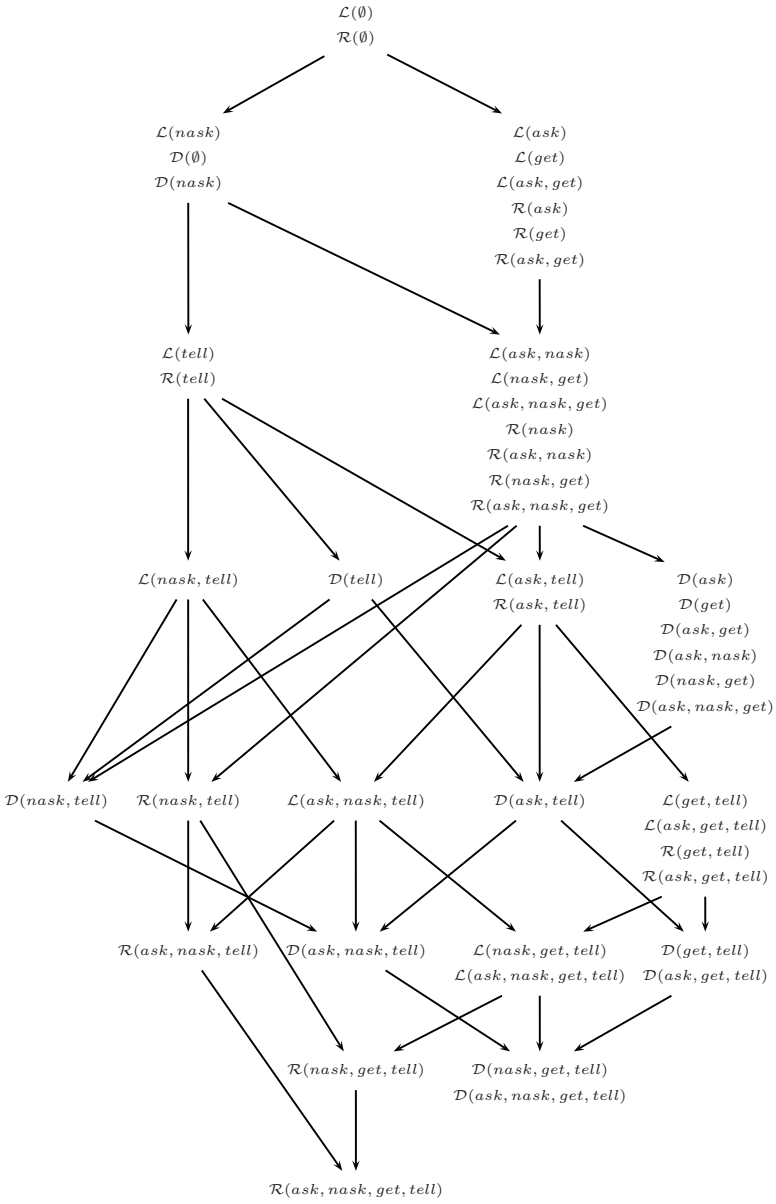

Fig. 3. Comparison of the $\mathcal{L}$, $\mathcal{D}$ and $\mathcal{R}$ families of languages

The resulting languages are called *tcc* and *tdcc*. In fact, rephrased in our framework, these languages correspond respectively to restricted variants of our $\mathcal{D}(\{ask, tell, nask\})$.

Although weaker than, for instance, the whole $\mathcal{R}$ language, the paper [32]

has shown that the language *tcc* can embed one classical representative of the state oriented synchronous languages, namely Argos ([21]), and one representative of the declarative class of dataflow synchronous languages, namely Lustre ([10]). It follows from section 4 that the same result holds for most of the languages we have proposed.

De Boer, Gabbrielli, and Meo have presented in [4] a timed interpretation of concurrent languages by fixing the time needed for the execution of parallel tell and ask operations as one unit and by interpreting action prefixing as the next operator. A delay mechanism is presented in Oz ([31]), a language which combines object oriented features with symbolic computation and constraints, and, (relative) time-outs have been introduced in TSpaces ([34]) and JavaSpaces ([14]). A formal semantics of these time-outs and other mechanisms, different from our expressiveness study, is presented in [7].

Another piece of work on the expressiveness of timed constraint systems is [24]. There, various extensions of the *tcc* languages have been studied: extension with replication and recursive procedures with static scoping. Decidability results are proved as well as several encodings, which are however not of the form of modular embeddings studied in this paper.

Finally, [8] investigates the impact of various mechanisms for expired data collection on the expressiveness of coordination systems. However, the study is based on Random Access Machines, on ordered and unordered tells of timed data and on decidability results. In contrast, we study a richer class of mechanisms and foccus on modular embeddings.

References

[1] J.-M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[2] J. Banatre and D. LeMetayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1991.

[3] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19, 1992.

[4] F.S. De Boer, M. Gabbrielli, and M.C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1):45–83, 2000.

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

[6] A. Brogi and J.-M. Jacquet. On the Expressiveness of Linda-like Concurrent Languages. *Electronic Notes in Theoretical Computer Science*, 16(2):61–82, 1998.

[7] N. Busi, R. Gorrieri, and G. Zavattaro. Process Calculi for Coordination: from Linda to JavaSpaces. In *Proc. AMAST*, Lecture Notes in Computer Science. Springer Verlag, 2000.

[8] N. Busi and G. Zavattaro. Expired Data Collection in Shared Dataspaces. *Theoretical Computer Science*, 298:529–556, 2003.

[9] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[10] P. Caspi, N. Halbwachs, P. Pilaud, and J. Plaice. Lustre: a Declarative Language for Programming Synchronous Systems. In *Proc. POPL'87*. ACM Press, 1987.

[11] P. Ciancarini. Distributed Programming with Logic Tuple Spaces. *New Generation Computing*, 12(3):251–284, 1994.

[12] P. Ciancarini and D. Rossi. Jada: Coordination and Communication for Java Agents. In *Proc.* 2^{nd} *International Workshop on Mobile Object Systems*, volume 1222 of *Lecture Notes in Computer Science*, pages 213–228. Springer-Verlag, 1996.

[13] F.S. de Boer and C. Palamidessi. Embedding as a Tool for Language Comparison. *Information and Computation*, 108(1):128–157, 1994.

[14] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns, and Practice*. Addison-Wesley, 1999.

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

[16] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987.

[17] E. Horita, J.W. de Bakker, and J.J.M.M. Rutten. Fully abstract denotational models for nonuiform concurrent languages. *Information and computation*, 115(1):125–178, 1994.

[18] J.-M. Jacquet and K. De Bosschere. On the Semantics of μLog. *Future Generation Computer Systems*, 10:93–135, 1994.

[19] J.-M. Jacquet, K. De Bosschere, and A. Brogi. On Timed Coordination Languages. In A. Porto and G.-C. Roman, editors, *Proc. 4th International Conference on Coordination Languages and Models*, volume 1906 of *Lecture Notes in Computer Science*. Springer, 2000.

[20] I. Linden, J.-M. Jacquet, K. de Bosschere, and A. Brogi. On the expressiveness of relative-timed coordination models. Technical report, Institute of Informatics, University of Namur, Belgium, 2003.

[21] F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Composition. In *Proc. Concurr'92*, volume 630 of *Lecture Notes in Computer Science*. Springer, 1992.

[22] D. Gelernter N. Carriero and L. Zuck. Bauhaus Linda. In In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object based models and languages for concurrent systems*, volume 924 of *Lecture Notes in Computer Science*, pages 66–76. Springer-Verlag, 1994.

[23] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 1998.

[24] M. Nielsen, C. Palamidessi, and F.D. Valencia. On the Expressive Power of Temporal Concurrent Constraint Programming Languages. In *Proceedings of the 4th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 156–167. ACM, 2002.

[25] G.A. Papadopolous and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 48, 1998.

[26] A. Rowstron and A. Wood. A Set of Tuple Space Primitives for Distributed Coordination. In *Proc.* 30^{th} *Hawaii International Conference on System Sciences*, volume 1, pages 379–388. IEEE Press, 1997.

[27] V. Saraswat, R. Jagadeesan, and V. Gupta. Programming in Timed Concurrent Constraint Languages. In B. Mayoh, E. Tougu, and J. Penjam, editors, *Computer and System Sciences*, volume ASI-131 of *NATO*. Springer Verlag, 1994.

[28] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 11, 1996.

[29] V.A. Saraswat. *Concurrent Constraint Programming Languages*. The MIT Press, 1993.

[30] E.Y. Shapiro. Embeddings among Concurrent Programming Languages. In W.R. Cleaveland, editor, *Proceedings of CONCUR'92*, pages 486–503. Springer-Verlag, 1992.

[31] G. Smolka. The Oz Programming Model. In J. Van Leuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer Verlag, 1995.

[32] S. Tini. On the Expressiveness of Timed Concurrent Constraint Programming. *Electronics Notes in Theoretical Computer Science*, 1999.

[33] R. Tolksdorf. Coordinating Services in Open Distributed Systems with LAURA. In P. Ciancarini and C. Hankin, editors, *Coordination'96: First International Conference on Coordination Models and Languages*, volume 1061 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[34] P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. TSpaces. *IBM Systems Journal*, 37(3), 1998.