Quantitative Information in the Tuple Space Coordination Model

Mario Bravetti, Roberto Gorrieri, Roberto Lucchi [1] and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università degli Studi di Bologna, Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
e-mail:{bravetti,gorrieri,lucchi,zavattar}@cs.unibo.it

Abstract

Tuple Spaces is a well known coordination model at the basis of coordination languages such as Linda, JavaSpaces and TSpaces. Tuple spaces are flat and unstructured multisets of tuples that can be accessed via output, read, and input operations. We investigate, from an expressiveness viewpoint, the impact of the introduction of quantitative information in the tuple space coordination model in order to quantify the relevance or importance of each tuple. We consider two possible interpretations for this information: in the first case it quantifies a weight indicating how much frequently each tuple should be retrieved, in the second case it expresses a priority level for the tuples.

1 Introduction

The notion of *coordination model* has been introduced by Carriero and Gelernter in [8], where a programming-specific meaning of the term *coordination* is reported presenting the following equation [8]:

$$\text{Programming} = \text{Computation} + \text{Coordination}$$

They formulated this equation arguing that in the design of a concurrent system there should be a clear separation between the specification of the components of the system and the specification of their interactions or dependencies. On the one hand, this separation facilitates the reuse of components; on the other hand, the same patterns of interaction usually occur in many different

[1] Corresponding author. E-mail: lucchi@cs.unibo.it.

problems – so it might be possible to reuse the coordination specification as well.

A number of interesting models have been proposed. Examples include "tuple spaces" as in Linda [11], various forms of "multiset rewriting" or "chemical reactions" as in Gamma [2]. The common feature of these models is to support the coordination among the collaborating entities through a common repository of data; for this reason, those models are called data-driven coordination models [18].

Coordination languages are the linguistic embodiement of coordination models; they are a class of programming notations which offer a solution to the problem of specifying and managing the interactions among computing entities. In fact, they generally offer language mechanisms for composing, configuring, and controlling systems made of independent, even distributed, active components.

The tuple space approach is the most prominent among the data-driven coordination models: processes communicate with the other entities in the environment by introducing (executing the *out* primitive) tuples, i.e. ordered sequences of data, in the tuple space; processes interested in retrieving information useful for coordinating their activities perform *rd* (nonconsuming read) or *in* (consuming input) operations specifying via a template the kind of tuples they are interested in. Following this approach, the tuple space is an unstructured and flat bag of tuples. All tuples have the same importance and relevance inside the shared repository, in the sense that when several tuples match the template of a data-retrieval operation, one of them is selected *non deterministically*.

In this paper we investigate the impact, from an expressiveness viewpoint, of the introduction of quantitative information in the tuple space coordination model. This information could be used to quantify the relevance or importance of each tuple in the tuple space. For instance, a tuple could be more important because it should be retrieved more frequently or with a higher precedence. The introduction of this information is justified by the existence of an interesting class of applications whose design could benefit from this enhanced tuple space coordination model. We briefly discuss two prototypical examples of this class of applications.

As a first example, consider the problem of coordinating clients and services in a dynamic environment where new services enter and old services exit. In this scenario, a typical approach is to exploit a service registry: new services advertise their availability registering at the service registry, while clients discover new services sending queries to the registry. The tuple space coordination model is particularly suited for supporting this kind of interaction between clients and services. The services register producing a tuple that describes the kind of service they intend to offer, while the clients access the tuple space in order to discover the actual service availability. In the case more than one service is willing to offer the required service, according to the standard tuple retrieval mechanism, one is chosen non deterministically. This is not satisfactory if we intend to distribute in a balanced way the workload, thus to avoid the overwhelming of requests towards one service while leaving other services under-utilized. This problem can be naturally solved if the tuple space coordination model is extended with the possibility to associate to the tuples a weight, and assuming that tuple retrieval occurs probabilisitically depending on those weights. The higher is the weight of a tuple, the greater is the probability for that tuple to be retrieved. In this enhanced coordination model, it is sufficient for a service to calculate a weight which quantifies (the inverse of) its current workload and associate it to the tuple used for the advertisement.

As a second example, consider a master-worker application where the masters produce jobs that are executed from the workers. Tuple spaces are particularly suited for this kind of applications: masters produce job requests storing tuples in the tuple space, while workers access the tuple space to retrieve the description of the jobs to execute. However, due to non determinism in tuple retrieval, it could be the case that a job is never executed because the masters indefinitely select more recent requests. Besides this problem of fairness, there are cases in which some job is to be executed more urgently with respect to the other ones. For instance, a job requested within a fire or an earthquake alarm should be executed by workers as soon as possible independently of the other currently available jobs. This scenario (as well as the problem of fairness) could be easily tackled assuming a tuple space coordination model extended with the possibility to decorate tuples with a urgency level.

1.1 Paper contribution and structure

In order to perform a formal investigation of the impact of the introduction of quantitative information in the tuple space coordination model, we need to introduce some adequate formal model. To be as general as possible, we intend to exploit a model which is independent of the different tuple space coordination languages available in the literature. For instance we model tuples as sequences of names all belonging to a unique sort. In other words, we do not consider typed fields as it happens e.g. in languages such as C-Linda [9] or JavaSpaces [10], where the fields are typed with C types or Java classes, respectively. We consider three coordination primitives: *out* to produce a new tuple, *in* and *rd* to consume and read tuples, respectively. The *in* and *rd* primitives use templates to indicate the kind of tuple to be retrieved: a template is an ordered sequence of names and wildcards. A tuple matches a template if it has the same length and it has the same names in the exact position where they appear in the template.

We embed the coordination primitives in a minimal language that comprises only three constructs: a prefix operator that associates to a primitive its continuation, a parallel operator to glue together the processes that coordinate through the tuple space, and a replication operator used to support a limited form of recursive behaviour. The obtained formal modeling of the tuple space coordination model is called `LinCa`.

After, we extend the syntax of `LinCa` adding the possibility to associate quantitative labels to tuples; we equip the new syntax with two different semantics thus obtaining two calculi, `PrioLinCa` in which the quantitative labels denote priorities and `ProbLinCa` in which they represent weights. These calculi permit to formally prove gap of expressiveness between the considered extensions and the native tuple space model.

The gap of expressiveness between `LinCa` and `PrioLinCa` is rather surprising; the addition of a minimum structure of priority, comprising only two priority levels *high* and *low*, is sufficient to increase the expressive power of the calculus from Turing incompleteness to Turing completeness. Indeed, we show how to model Random Access Machines (a well-known register based Turing powerful formalism) using two levels of priority, and we prove that such an encoding cannot be provided in `LinCa` because termination is decidable for such a calculus.

On the other hand, the gap of expressiveness we prove between `LinCa` and `ProbLinCa` follows a more traditional discriminating technique between nondetermistic and probabilistic behaviour in distributed systems with asynchronous communication. Namely, we consider the *leader election problem* (see, e.g., [1]), and we show that it can be solved for symmetric systems only in `ProbLinCa` while this is not the case for `LinCa`.

The paper is structured as follows. In Section 2 we present the `LinCa` calculus. In Section 3 we extend the syntax of `LinCa` introducing the quantitative labels, we provide the prioritized semantics (thus obtaining the `PrioLinCa` calculus) and we formally prove our first gap of expressiveness modeling Random Access Machines in `PrioLinCa` and proving that termination is decidable in `LinCa`. In Section 4 we present the probabilistic semantics (thus obtaining the `ProbLinCa` calculus) and we discuss how weights on tuples permit to solve the leader election problem (the proof of the impossibility to solve leader election in `LinCa` is reported in Appendix A). Finally, in Section 5 we draw some conclusive remark discussing how our results, proved on abstract formal models, adapt to concrete tuple space coordination languages.

2 `LinCa`: the Linda Calculus

The coordination primitives that we basically have in Linda-like coordination languages are: $out(e)$, $in(t)$ and $rd(t)$. The output operation $out(e)$ inserts a tuple e in the tuple space (TS for short). Primitive $in(t)$ is the blocking input operation: when an occurrence of a tuple e matching with t (denoting a template) is found in the TS, it is removed from the TS and the primitive returns the tuple. The read primitive $rd(t)$ is the blocking read operation that, differently from $in(t)$, returns the matching tuple e without removing it from the TS.

The tuples are ordered and finite sequences of typed fields, while template are ordered and finite sequences of fields that can be either *actual* or *formal* (see [9]): a field is actual if it specifies a type and a value, whilst it is formal if the type only is given. For the sake of simplicity, in the formalization of Linda we are going to present, fields are not typed.

Formally, let $Mess$, ranged over by m, m', $\ldots$, be a denumerable set of messages and Var, ranged over by x, y, $\ldots$, be the set of data variables. In the following, we use $\vec{x}$, $\vec{y}$, $\ldots$, to denote finite sequences $x_1; x_2; \ldots; x_n$ of variables.

Tuples, denoted by e, e', $\ldots$, are finite and ordered sequences of data fields (we use $arity(e)$ to denote the number of fields of e), whilst templates, denoted by t, t', $\ldots$, are finite and ordered sequences of fields that can be either data or wildcards (used to match with any message).

Formally, tuples are defined as follows:

$$e = \langle \vec{d} \rangle$$

where $\vec{d}$ is a term of the following grammar:

$\vec{d} ::= d \mid d; \vec{d}$
$d ::= m \mid x.$

The definition of template follows:

$$t = \langle \vec{dt} \rangle$$

where $\vec{dt}$ is a term of the following grammar:

$\vec{dt} ::= dt \mid dt; \vec{dt}$
$dt ::= d \mid null.$

A *data field* d can be a message or a variable. The additional value *null* denotes the wildcard, whose meaning is the same of formal fields of Linda, i.e. it matches with any field value. In the following, the set $Tuple$ (resp. $Template$) denotes the set of tuples (resp. templates) containing no variable.

The matching rule between tuples and templates we consider is the classical one of Linda, whose definition is as follows.

Definition 2.1 ***Matching rule -*** *Let $e = \langle d_1; d_2; \ldots ; d_n\rangle \in Tuple$ be a tuple, $t = \langle dt_1; dt_2; \ldots ; dt_m\rangle \in Template$ be a template; we say that e matches t (denoted by $e \triangleright t$) if the following conditions hold:*

(1) $m = n$.
(2) $dt_i = d_i$ or $dt_i = null$, $1 \leq i \leq n$.

Condition 1. checks if e and t have the same arity, whilst 2. tests if each non-wildcard field of t is equal to the corresponding field of e.

Processes, denoted by P, Q, ..., are defined as follows:

$P, Q, \ldots ::=$		processes
	$\mathbf{0}$	null process
$\mid$	$out\ (e).P$	output
$\mid$	$rd\ t(\vec{x}).P$	read
$\mid$	$in\ t(\vec{x}).P$	input
$\mid$	$P \mid P$	parallel composition
$\mid$	$!in\ t(\vec{x}).P$	replication

A process can be a terminated program $\mathbf{0}$ (that we usually omit), a prefix form $\mu.P$, the parallel composition of two programs, or the replication of a program. The prefix μ can be one of the following coordination primitives: i) $out\ (e)$, that writes the tuple e in the TS; ii) $rd\ t(\vec{x})$, that given a template t reads a matching tuple e in the TS and stores the return value in $\vec{x}$; iii) $in\ t(\vec{x})$, that given a template t consumes a matching tuple e in the TS and stores the return value in $\vec{x}$. In both the $rd\ t(\vec{x})$ and $in\ t(\vec{x})$ operations $(\vec{x})$ is a binder for the variables in $\vec{x}$. The parallel composition $P \mid Q$ of two processes P and Q behaves as two processes running in parallel. Infinite behaviours can be expressed using the replication operator $!in\ t(\vec{x}).P$. Replication is a typical operator used in process calculi to denote the parallel composition of an unbounded amount of instances of the same process. In our calculus we restrict the application of replication to input guarded processes. This is justified by the fact that replicated output operation (resp. read operations), may give rise to an undesired behaviour by producing (reading) a tuple an unbounded amount of time.

In the following, $P[d/x]$ denotes the process that behaves as P in which all occurrences of x are replaced with d. We also use $P[\vec{d}/\vec{x}]$ to denote the process obtained by replacing in P all occurrences of variables in $\vec{x}$ with the corresponding value in $\vec{d}$, i.e. $P[d_1; d_2; \ldots ; d_n/x_1; x_2; \ldots ; x_n] = P[d_1/x_1][d_2/x_2] \ldots [d_n/x_n]$.

We say that a process is *well formed* if each prefix of kind $rd/in\ \langle\vec{dt}\rangle(\vec{x})$ is such that the variables $\vec{x}$ and the data $\vec{dt}$ have the same arity. Notice that in the $rd\ t(\vec{x})$ and $in\ t(\vec{x})$ operations we use a notation which is different from the standard Linda notation: we explicitly indicate in $(\vec{x})$ the variables that will be bound to the actual fields of the matching tuple, while in the standard Linda notation these variables are part of the template. Observe that the two notations are equivalent up to the fact that our notation introduces variables also in association to the formal fields of the template. In the following, we consider only processes that are well formed; $Process$ denotes the set of such processes.

Let $DSpace$, ranged over by DS, DS', $\ldots$, be the set of possible configurations of the TS, that is $DSpace = \mathcal{M}_{fin}(Tuple)$, where $\mathcal{M}_{fin}(S)$ denotes the set of all the possible finite multisets on S. In the following, we use $DS(e)$ to denote the number of occurrences of e within $DS \in DSpace$.The set $System = \{[P, DS] \mid P \in Process, DS \in DSpace\}$, ranged over by s, s', $\ldots$, denotes the possible configurations of systems.

The semantics we use to describe processes interacting via coordination primitives is defined in terms of a transition system $(System, \longrightarrow)$, where $\rightarrow \subseteq System \times System$. More precisely, $\longrightarrow$ is the minimal relation satisfying the axioms and rules of Table 1 (symmetric rule of (4) is omitted). $(s, s') \in \longrightarrow$ (also denoted by $s \longrightarrow s'$) means that a system s can evolve (performing a single action) in the system s'. Finally, we use $s \longrightarrow^+ s'$ (resp. $s \longrightarrow^* s'$) for the transitive (resp. reflexive and transitive) closure of $\longrightarrow$. A computation $s_1 \longrightarrow s_2 \longrightarrow \ldots \longrightarrow s_n$ is maximal if there exists no s' such that $s_n \longrightarrow s'$. We say that a process *terminates* if it has a finite maximal computation.

Axiom (1) describes the output operation that produces a new occurrence of the tuple e in the shared space DS ($DS \oplus e$ denotes the multiset obtained by DS increasing by 1 the number of occurrences of e). Rules (2) and (3) describe the in and the rd operations, respectively: if a matching e tuple is currently available in the space, it is returned at the process invoking the operation and, in the case of in, it is also removed from the space ($DS - e$ denotes the removal of an occurrence of e from the multiset DS). Rule (4) represents a local computation of processes, whilst (5) the replication operator that produces a new instance of the process and copies itself.

2.1 Examples

Here, we report some simple examples with the aim of introducing the reader to the notation used in `LinCa`. These examples are inspired to the master-worker applications and the service registry discussed in the Introduction.

$$(1) \quad [out\ (e).P, DS] \longrightarrow [P, DS \oplus e]$$

$$(2) \quad \frac{\exists e \in DS : e \triangleright t}{[in\ t(\vec{x}).P, DS] \longrightarrow [P[e/\vec{x}], DS - e]}$$

$$(3) \quad \frac{\exists e \in DS : e \triangleright t}{[rd\ t(\vec{x}).P, DS] \longrightarrow [P[e/\vec{x}], DS]}$$

$$(4) \quad \frac{[P, DS] \longrightarrow [P', DS']}{[P \mid Q, DS] \longrightarrow [P' \mid Q, DS']}$$

$$(5) \quad \frac{[in\ t(\vec{x}).P, DS] \longrightarrow [P', DS']}{[!in\ t(\vec{x}).P, DS] \longrightarrow [P' \mid !in\ t(\vec{x}).P, DS']}$$

Table 1
Semantics of `LinCa`. Symmetric rule of (4) omitted.

Example 2.2 Master-worker - *The idea is that masters request a job supplied by workers by inserting a tuple containing the job and workers select jobs by retrieving such tuples from the space. Let $job \in Mess$ be a job; the procedures $submit(job)$ and $supply\_job$, which submits and supplies a job, respectively, are defined as follows:*

$submit(job) \stackrel{def}{=} out(\langle job \rangle)$

$supply\_job \stackrel{def}{=} in\ \langle null \rangle(x).Supply(x)$

where $Supply(x)$ is the process performing the job x. It is clear that the workers select submitted jobs in a non-deterministic way, thus preventing to manage different urgency levels of jobs.

Example 2.3 Service registry - *In this case the tuple space is used as a services registry. The registration of a new service consists of inserting a new tuple containing the service information. To discover services, processes can perform read operations. Let $s \in Mess$ and $pl \in Mess$ be a task and a link to a service, respectively. The procedure $register(s, pl)$, which registers a service supplying task s that is available at link pl, is defined as follows:*

$register(s, pl) \stackrel{def}{=} out(\langle s; pl \rangle)$

while the procedure $discover(s)$, which discovers a service supplying task s, is defined as follows:

$discover(s) \stackrel{def}{=} rd\ \langle s; null \rangle (x_1; x_2)$
where at the end of the computation x_2 will contain the link to a service supplying task s. When more than one service is available, the retrieved link is non-deterministically chosen.

3 `PrioLinCa`: the Calculus with Prioritized Data Retrieval

We extend the syntax of `LinCa` permitting to decorate each tuple with quantitative information in terms of a *quantitative label.* This is a positive (non-zero) natural number that can be associated to the tuples in order to quantify the relevance of the tuple inside the tuple space.

In this section we analyze the expressiveness of the quantitative labels when they are interpreted as priorities. Priorities on tuples represent an absolute preference of the currently available tuples in the shared space. More precisely, if a process performing a rd/in operation receives as the return value a tuple e, there is no currently available matching tuple e' in TS such that its priority level is greater than the one of e.

We present a model that formalizes the following intuition: priorities on tuples are set by the output operations and the data-retrieval operations return a matching tuple with the highest priority among the available ones. The obtained calculus is named `PrioLinCa`. We prove that `PrioLinCa` is expressive enough for modeling Random Access Machines (RAMs) [23], a well known Turing powerful formalism. On the contrary, we show that such an encoding cannot be provided in `LinCa` as termination (the existence of a terminating computation) is decidable in the calculus without quantitative labels.

3.1 The Syntax with Quantitative Labels

In order to add the quantitative labels, the syntax of `LinCa` is slightly modified as follows.

Let $QLab$, ranged over by $l, l', \ldots$, be the set of the possible quantitative labels. Even if not necessary, for the sake of simplicity, we assume to use positive (non-zero) natural numbers as quantitative labels, thus $QLab$ coincides with $\mathbb{N} \setminus \{0\}$. Tuples are now defined as follows:

$$e = \langle \vec{d} \rangle\ [l]$$

where $l \in QLab$ and $\vec{d}$ is a sequence of data fields d that are defined by the following grammar:

$d ::= m \mid l \mid x.$

A data field d now can be a message, a quantitative label, or a variable. We also define $\tilde{\cdot}$ as the function that, given a tuple e, returns its sequence of data fields (e.g. if $e = \langle \vec{d} \rangle\,[l]$ then $\tilde{e} = \vec{d}$). In the following, we denote with QL the function that, given a tuple, returns its quantitative label (e.g., if $e = \langle \vec{d} \rangle\,[l]$ then $QL(e) = l$). Quantitative labels are not considered in the matching rule whose definition is unchanged.

Example 3.1 Master-worker with job priorities - *We extend the Example 2.2 by allowing a classification between critical and standard jobs. Critical jobs must be executed as soon as a free worker is available. We can program such a system using 'critical' and 'standard' as symbolic names representing quantitative labels, and interpreting critical as a priority higher than standard. Thus, the two procedures for job submissions become:*

$submitCritical(job) \stackrel{def}{=} out(\langle job \rangle[critical])$

$submitStandard(job) \stackrel{def}{=} out(\langle job \rangle[standard]).$

The workers can supply a job by using the following procedure:

$supply\_job \stackrel{def}{=} in\ \langle null \rangle(x).Supply(x)$

that is exactly the one defined in Example 2.2. In this way, when a worker performs the supply_job procedure it is ensured that no standard jobs are served in case at least one critical job has been submitted.

3.2 The Prioritized Semantics

In order to formalize the prioritized semantics we follow an approach similar to the one adopted in [4,3] where priorities are represented in terms of attributes.

The new semantics extends that of `LinCa` by verifying that when the rd/in primitives are performed all the available matching tuples have a lower priority with respect to the priority of the returned one. The semantics is thus obtained by taking the rules in Table 1 and replacing rules (2) and (3) with those reported in Table 2.

Informally, priority levels partition the space: the idea is that the search space is restricted to the partition of DS which has the greatest priority level and contains a matching tuple.

3.3 Modelling RAMs in `PrioLinCa`

We show that `PrioLinCa` is expressive enough for encoding any Random Access Machine (RAM) [23], that is a Turing complete formalism. In Section 3.4

$$(2') \quad \frac{\exists e \in DS : e \triangleright t \quad \forall e' \in DS : e' \triangleright t \quad QL(e) \geq QL(e')}{[in\ t(\vec{x}).P, DS] \longrightarrow [P[\tilde{e}/\vec{x}], DS - e]}$$

$$(3') \quad \frac{\exists e \in DS : e \triangleright t \quad \forall e' \in DS : e' \triangleright t \quad QL(e) \geq QL(e')}{[rd\ t(\vec{x}).P, DS] \longrightarrow [P[\tilde{e}/\vec{x}], DS]}$$

Table 2
Semantics of `PrioLinCa`. Rules (1), (4) and (5) of Table 1 omitted.

we prove that such an encoding cannot be provided in `LinCa` by showing that process termination is decidable in that calculus. This is proved by giving the semantics of `LinCa` in terms of Petri nets where deadlock is decidable.

Definition 3.2 *A RAM is a computational model composed of a finite set of registers, that can hold arbitrary natural numbers, and by a program, that is a sequence of simple numbered instructions, like arithmetical operations (on the contents of registers) or conditional jumps. Programs receive the input parameters in registers $r_1, \ldots, r_m$, that can hold arbitrary large numbers; any other register used by the program during the computation $r_{m+1}, \ldots, r_n$ are supposed to contain the value 0 when it starts. The program execution starts with the first instruction and continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached; this happens if the program was executing the last instruction of the program and this instruction does not require a jump, or if the current instruction requires a jump to an instruction number not appearing in the program. The output of the program, if it terminates, is represented by the value of certain registers.*

In [14] it is shown that recursive functions can be modeled by RAMs programmed with the following instructions only:

- *$Succ(r_j)$: adds 1 to the contents of register r_j;*
- *$DecJump(r_j, s)$: if the contents of register r_j is not zero, then decreases it by 1 and go to the next instruction, otherwise jumps to instruction s.*

The state of the computation is represented by means of configurations. A configuration is denoted by $(i, c_1, c_2, \ldots, c_n)$ where i indicates that the next instruction to execute is the i^{th} and c_l is the contents of the register r_l for $l = 1, \ldots, n$. Given a program R and a configuration $(i, c_1, c_2, \ldots, c_n)$ we use the notation:

$(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (j, c'_1, c'_2, \ldots, c'_n)$

to state that after the execution of the i^{th} instruction of the program R with *contents of the registers $c_1, \ldots, c_n$, the program counter points to the j^{th} instruction, and the registers will contain $c'_1, \ldots, c'_n$.*

The technique we use to encode RAM programs is inspired to the one used in [5]. In that paper RAMs are modeled as follows. Each unit inside a register r_j is modeled with an instance of the tuple $\langle r_j \rangle$. In this way, an increment instruction on r_j simply produces a new instance of $\langle r_j \rangle$, while a decrement instruction removes, if available, one instance of $\langle r_j \rangle$. In this last case, the consumption operation is performed using an *inp* operation; *inp* is a nonblocking version of *in* which terminates communicating failure in case no tuple is available for consumption.

In `PrioLinCa`, nonblocking primitives similar to *inp* are not available. Thus, we have to modify the representation of registers in order to avoid blocking in case a decrement operation is performed on an empty register. The solution we adopt exploits two levels of priority that we call *high* and *low* (where the former is greater than the latter). An empty register r_j is modeled by a tuple $\langle r_j, \mathit{empty} \rangle[\mathit{low}]$. Each unit inside r_j is represented by a tuple $\langle r_j, \mathit{nonempty} \rangle[\mathit{high}]$. When a decrement operation is performed, the instruction $\mathit{in}\langle r_j; \mathit{null} \rangle(x_1; x_2)$ is executed; this operation cannot block because at least one tuple among $\langle r_j, \mathit{nonempty} \rangle[\mathit{high}]$ and $\langle r_j, \mathit{empty} \rangle[\mathit{low}]$ is available. As the units $\langle r_j, \mathit{nonempty} \rangle[\mathit{high}]$ have a higher priority w.r.t. $\langle r_j, \mathit{empty} \rangle[\mathit{low}]$, it is ensured that the variable x_2 is assigned the value *empty* if and only if the register is actually empty. In this way it is enough to test the value of x_2 in order to chose the correct continuation, either incrementing the program counter by 1 or jumping.

Formally, we encode RAMs by defining systems $[P, DS]$ in which i) program states define the tuple space DS, and ii) program instructions define the process P.

Program states are encoded by means of tuples. Given the configuration $(i, c_1, c_2, \ldots, c_n)$, the next instruction is indicated by the tuple $\langle p_i \rangle[l]$ (l arbitrary), while the value of register r_j is encoded by c_j occurrences of the tuple $\langle r_j; \mathit{nonempty} \rangle[\mathit{high}]$ and by one tuple $\langle r_j; \mathit{empty} \rangle[\mathit{low}]$ (where $\mathit{low} < \mathit{high}$), for $1 \leq j \leq n$. It is worth noting that $\mathit{in}\ \langle r_j, \mathit{null} \rangle(\vec{x}).P$ (the same holds for rd) retrieves the tuple $\langle r_j; \mathit{empty} \rangle[\mathit{low}]$, that we call bottom of r_j, only in the case no occurrences of $\langle r_j; \mathit{nonempty} \rangle[\mathit{high}]$ are available and this happens when the value of the register r_j is 0.

Formally, the encoding of program states that determines the configuration of the tuple space is defined as follows.

$$
\begin{aligned}
[\![(i, c_1, c_2, \ldots, c_n)]\!] \stackrel{def}{=}\ & \langle p_i \rangle[l] \oplus \langle r_1; \mathit{empty}\rangle[\mathit{low}] \oplus \ldots \oplus \langle r_n; \mathit{empty}\rangle[\mathit{low}] \oplus \\
& \underbrace{\langle r_1; \mathit{nonempty}\rangle[\mathit{high}] \oplus \ldots \oplus \langle r_1; \mathit{nonempty}\rangle[\mathit{high}]}_{c_1 \text{ times}} \oplus \\
& \ldots \\
& \oplus \underbrace{\langle r_n; \mathit{nonempty}\rangle[\mathit{high}] \oplus \ldots \oplus \langle r_n; \mathit{nonempty}\rangle[\mathit{high}]}_{c_n \text{ times}}
\end{aligned}
$$

Let R be a program composed of the sequence of instructions $I_1 \ldots I_k$. Each instruction is modeled by a replicated process guarded by an input operation on the corresponding program counter tuple. Instructions are replicated because they can be executed an unbounded amount of time during the computation. Formally we have define the encoding for instructions as follows:

$$
\begin{aligned}
[\![R]\!] \quad & \stackrel{def}{=} [\![I_1]\!] \mid \ldots \mid [\![I_k]\!] \\
[\![i : \mathit{Succ}(r_j)]\!] \quad & \stackrel{def}{=} !\mathit{in}\langle p_i\rangle(x).\mathit{out}(\langle r_j; \mathit{nonempty}\rangle[\mathit{high}]).\mathit{out}(\langle p_{i+1}\rangle[l]) \\
[\![i : \mathit{DecJump}(r_j, s)]\!] \quad & \stackrel{def}{=} !\mathit{in}\langle p_i\rangle(x).\mathit{in}\langle r_j; \mathit{null}\rangle(x_1; x_2).\mathit{out}(\langle \mathit{cnt}_i; x_2\rangle[l]) \mid \\
& \quad !\mathit{in}\langle \mathit{cnt}_i; \mathit{nonempty}\rangle(y_1; y_2).\mathit{out}(\langle p_{i+1}\rangle[l]) \mid \\
& \quad !\mathit{in}\langle \mathit{cnt}_i; \mathit{empty}\rangle(y_1; y_2).\mathit{out}(\langle r_j; \mathit{empty}\rangle[\mathit{low}]). \\
& \quad \mathit{out}(\langle p_s\rangle[l])
\end{aligned}
$$

The increment instruction simply produces a new instance of the register tuple and then increments the program counter. The decrement instruction requires a more complex encoding. It is composed of three replicated processes. The first process is responsible for starting the instruction by trying to decrement the register; the other two processes manage the two possible continuations. The first of these two possible continuations is activated if the decrement succeeded and simply increments the program counter; the second one manages the case in which the register is empty by performing the corresponding jump.

After having consumed the corresponding program counter, the first process performs the input operation $\mathit{in}\langle r_j; \mathit{null}\rangle(x_1; x_2)$; as discussed above the variable x_2 is assigned with either *nonempty* (if the decrement has succeeded) or *empty* (if the register is actually empty). After, the output operation $\mathit{out}(\langle \mathit{cnt}_i; x_2\rangle[l])$ is executed, where l is an arbitrary priority and cnt_i is a datum indicating that one of the two continuations should start. The correct continuation is selected by the second field x_2. The continuations are simple; the unique relevant note is that in the case the register is empty the tuple $\mathit{out}(\langle r_j; \mathit{empty}\rangle[\mathit{low}])$ should be reproduced as it could be read in subsequent decrement instructions.

In order to formalize the correctness of the encoding, we abstract away from the empty processes that are produced during the computation; namely, we do not distinguish the process P from the processes $\mathbf{0}|P$ and $P|\mathbf{0}$.

Theorem 3.3 *Let R be a RAM program, then:*

- Soundness: *for any maximal computation*
  $[[\![R]\!], [\![(i, c_1, c_2, \ldots, c_n)]\!]] \longrightarrow [P_1, DS_1] \longrightarrow [P_2, DS_2] \longrightarrow \ldots$
  there exist i and a configuration $(j, c'_1, c'_2, \ldots, c'_n)$ *such that:*
  $[P_i, DS_i] = [[\![R]\!], [\![(j, c'_1, c'_2, \ldots, c'_n)]\!]]$ *and*
  $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (j, c'_1, c'_2, \ldots, c'_n)$
- Completeness: *if* $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (j, c'_1, c'_2, \ldots, c'_n)$ *then also:*
  $[[\![R]\!], [\![(i, c_1, c_2, \ldots, c_n)]\!]] \longrightarrow^+ [[\![R]\!], [\![(j, c'_1, c'_2, \ldots, c'_n)]\!]]$

Proof *By cases on the possible instruction (either increment, decrement or jump) that can be activated. In the proof of soundness we use the fact that the program counter tuple* $\langle p_i \rangle[l]$ *in the system* $[[\![R]\!], [\![(i, c_1, c_2, \ldots, c_n)]\!]]$ *ensures that only the agent corresponding to the* i^{th} *instruction can start the maximal computation; moreover the computation evolves deterministically.* ■

Corollary 3.4 *Let R be a RAM program and* $(i, c_1, c_2, \ldots, c_n)$ *its initial configuration. The RAM R terminates if and only if* $[[\![R]\!], [\![(i, c_1, c_2, \ldots, c_n)]\!]]$ *terminates.*

Proof *We first observe that given a configuration* $(i', c'_1, c'_2, \ldots, c'_n)$ *the system* $[[\![R]\!], [\![(i, c'_1, c'_2, \ldots, c'_n)]\!]]$ *has no outgoing transition if and only if the configuration* $(i', c'_1, c'_2, \ldots, c'_n)$ *is a final configuration for the RAM R.*
After we observe that given the RAM computation that terminates, by completeness we can produce a finite computation that is maximal for the above observation.
On the other hand, given a finite maximal computation of the encoding, by soundness we can produce a RAM computation by partitioning the maximal computation in such a way that each fragment of the computation corresponds to a RAM step; in light of the above observation this RAM computation reaches a terminating configuration. ■

From this Corollary we can conclude that termination of `PrioLinCa` systems is undecidable.

3.4 Deciding Termination in `LinCa`

We show that the calculus `LinCa` is less expressive than the prioritized extension `PrioLinCa` by proving that it is not expressive enough for modeling faithfully RAMs. This is a consequence of the fact that termination turns to be decidable in `LinCa`.

To prove this decidability result we present an encoding of `LinCa` systems into finite Place/Transition nets that preserves the existence of a finite computation; the decidability of termination follows from the fact that the deadlock problem is decidable [20] for finite Place/Transition nets.

This proof technique has been already used in [5] to prove that the the deadlock problem is decidable in a Linda language equipped with a different semantics for the output operation, called *unordered*. An output is unordered if there is an unpredictable delay between the execution of the operation and the actual availability of the emitted tuple in the space. Besides the different semantics of the output operation, there are two other main differences between the calculus in [5] and `LinCa`. In `LinCa` the system state is represented by a pair composed of the processes and the tuple space, while in [5] the system glues together processes and tuples that are terms of the same algebra. In `LinCa` the in and rd operations permit to acquire names using wildcards while in [5] tuples as well as templates are composed of only one name, and no wildcards are permitted. This difference has the following consequence: in [5] the execution of an in or rd operation has only one possible continutation, while in `LinCa` there are different continuations depending on the actually acquired names. Thus, it is necessary to prove that the Place/Transition nets obtained by encoding `LinCa` systems are finite also in this more general case. Informally, such a condition holds because the calculus does not allow new names definition, and the number of possible continuations is always less than the number of tuples that can be produced with the (finite) number of initially available names.

Definition 3.5 *A P/T system is a triple $N = (S, T, m_0)$ where S is the set of* places*, $T \subseteq \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S)$ is the set of* transitions*, and m_0 is a finite multiset over the set S of places. Finite multisets over the set S of places are called* markings*; m_0 is called* initial marking*. Given a marking m and a place s, we say that the place s contains $m(s)$* tokens*. We also define $dom(m)$ as the domain of m which is defined as the set of places s with $m(s) > 0$. A P/T system is finite if both S and T are finite.*

A transition $\tau = (c, p)$ (also denoted by $c \longrightarrow p$) is composed of the marking c (usually called preset *and denoted by ${}^\bullet\tau$) which represents the tokens to be* consumed*, and the marking p (usually called* postset *and denoted by $\tau^\bullet$) which represents the tokens to be* produced*. A transition τ can be performed at a marking m if ${}^\bullet\tau \subseteq m$ and the reached marking is $m' = (m \setminus {}^\bullet\tau) \oplus \tau^\bullet$. This is written as $m \xrightarrow{\tau} m'$ or simply $m \longrightarrow m'$ when the transition τ is not relevant. We use σ to range over sequences of transitions; the empty sequence is denoted by ε; let $\sigma = \tau_1, \ldots, \tau_n$, we write $m \xrightarrow{\sigma} m'$ to mean the* firing sequence *$m \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_n} m'$. We say that m' is* reachable from *m if there exists σ such that $m \xrightarrow{\sigma} m'$.*

A marking m is dead *if $\not\exists m'$ s.t. $m \longrightarrow m'$. The net $N = (S, T, m_0)$ has a* deadlock *if there exists a dead marking reachable from m_0. The deadlock problem for a net consists of deciding if it has a deadlock.*

Now we show that any `LinCa` system can be represented by a finite P/T net: places are used to represent tuples and sequential processes. By sequential process we mean a process that is guarded by a coordination primitive or replicated (remember that replication can be applied only to processes guarded by input operations).

In this respect, parallel processes can be encoded by defining a place for every sequential process it contains or that can be reached during the computation; the transitions are used to describe the system behavior.

We use $\mathcal{S}$ to denote the possible places for the P/T nets we consider:

$$\mathcal{S} = \{P \mid P \text{ is a sequential process}\} \cup \{e \mid e \in \mathit{Tuple}\}$$

We define the decomposition function $dec : \mathit{System} \rightarrow \mathcal{M}_{fin}(\mathcal{S})$ that maps `LinCa` systems into markings as follows:

$$dec([P, DS]) = dec_p(P), DS$$

$$\begin{array}{ll} dec_p(\mathbf{0}) = \phi & dec_p(\mu.P) = \{\mu.P\} \\ dec_p(P|Q) = dec_p(P), dec_p(Q) & dec_p(!in\ t(\vec{x}).P) = \{!in\ t(x).P\} \end{array}$$

We also define $\mathcal{T}$ as the transition set obtained as instances of the following axioms schemata (we parameterize the axioms in order to identify specific instances):

$$\begin{array}{lll} IN(t, e, Q) & in\ t(\vec{x}).Q, e \longrightarrow dec(Q\,[e/\vec{x}]) & \text{if } e \rhd t \\ RD(t, e, Q) & rd\ t(\vec{x}).Q, e \longrightarrow dec(Q\,[e/\vec{x}]), e & \text{if } e \rhd t \\ OUT(e, Q) & out(e).Q \longrightarrow e, dec(Q) & \\ REP(t, e, Q) & !in\ t(\vec{x}).Q, e \longrightarrow\ !in\ t(\vec{x}).Q, dec(Q\,[e/\vec{x}]) & \text{if } e \rhd t \end{array}$$

Axioms IN, RD and OUT describe the three prefixes, while REP are the transitions which describe the behavior of the replication operator.

Let $ma([P, DS])$ be a function which returns the maximum among the arity of the longest tuple contained in DS, or the arity of the longest template or tuple in P. Obviously, $ma([P, DS])$ is finite for any `LinCa` system $[P, DS]$. We also define $df(P)$ and $df(DS)$ as the functions that return the set of data fields, except the variables, contained in P and DS, respectively. With $Tup_{[P,DS]}$ we denote the set of tuples that can be obtained combining the possible data fields appearing in the system $[P, DS]$ for an arity less than the maximal arity appearing in the same system:

$$
\begin{aligned}
Tup_{[P,DS]} \;=\; & \{e \in Tuple \mid df(e) \subseteq df(P) \cup df(DS) \text{ and} \\
& \qquad\qquad arity(e) \leq ma(P, DS)\}
\end{aligned}
$$

This set is clearly finite for any system $[P, DS]$.

Since the calculus uses value passing, the set of places should be defined taking into account all the possible substitutions of formal variables to actual received messages.

$$
\begin{aligned}
Prc_{[P,DS]} \;=\; & \{Q \mid Q \text{ is obtained by a sequential subprocess of } P \\
& \qquad \text{changing its variables with values of } df(P) \cup df(DS)\}
\end{aligned}
$$

Observe that also this set is finite for any system $[P, DS]$.

We are now in place for formally defining the P/T net associated with a `LinCa` system.

Definition 3.6 *Let $[P, DS]$ be a `LinCa` system, the corresponding P/T net, denoted by $Net([P, DS]) = (S, T, m_0)$, is defined as follows:*

$$
\begin{aligned}
S &= Tup_{[P,DS]} \cup Prc_{[P,DS]} \\
T &= \{c \to p \mid (c, p) \in \mathcal{T} \text{ s.t. } dom(c) \subseteq S\} \\
m_0 &= dec([P, DS]).
\end{aligned}
$$

As $Tup_{[P,DS]}$ and $Prc_{[P,DS]}$ are finite for any `LinCa` system $[P, DS]$, we can conclude that $Net([P, DS])$ is finite.

In the remaining part of this section we will prove that the encoding respects the transition system semantics of `LinCa` given in Section 2, thus the deadlock problem is decidable in `LinCa`. In order to prove this result, formalized in Theorem 3.9, we exploit Propositions 3.7 and 3.8 which show that: for any transition performable by a `LinCa` system s' whose marking is legal in the P/T net corresponding to the system s there exists a transition in the net in which the reached marking is the decomposition of the state reached by s', and vice versa.

Proposition 3.7 *Let $s_l \in System$ be a `LinCa` system and $Net(s_l) = (S, T, m_0)$ be the corresponding P/T net. Let $s \in System$ be a system s.t. $dom(dec(s)) \subseteq S$, then if $s \longrightarrow s'$ there exists $\tau \in T$ s.t. $dec(s) \xrightarrow{\tau} dec(s')$.*

Proof *We proceed by induction on the derivation of the transition $s \longrightarrow s'$. We have the following base cases; the transition is proved using either (1), (2), (3) or (5) of Table 1. We report the analysis of the first case (the other cases are similar).*

If $s \longrightarrow s'$ is proved using directly the axiom (1) we have that $s = [out(e).P, DS]$ and $s' = [P, DS \oplus e]$. Since $dec([out(e).P, DS]) = \{out(e).P\} \oplus DS \subseteq S$, then

$OUT(e, P) \in T$. By performing $OUT(e, P)$ at marking $\{out(e).P\} \oplus DS$ the reached marking is $dec_p(P) \oplus e \oplus DS$. Finally, it is easy to verify that this is the decomposition of s'.
In the inductive case we have that the last rule to be applied in the derivation of $s \longrightarrow s'$ is (4) or its symmetric. In the first of these two cases we have that $s = [P|Q, DS]$ and $s' = [P'|Q, DS']$ as well as $[P, DS] \longrightarrow [P', DS']$. By induction hypothesis, we have that $dec([P, DS]) \xrightarrow{\tau} dec([P', DS'])$ for some $\tau \in T$. This transition can be fired also in any marking greater than $dec([P, DS])$, thus also for $dec([P|Q, DS]) = dec_p(P) \oplus dec_p(Q) \oplus DS$; the reached marking is $dec_p(P') \oplus dec_p(Q) \oplus DS' = dec([P'|Q, DS'])$. The analysis for the symmetric rule is similar. ■

Proposition 3.8 *Let $s_l \in System$ be a* `LinCa` *system and $Net(s_l) = (S, T, m_0)$ be the corresponding P/T net. Let $s \in System$ be a system s.t. $dom(dec(s)) \subseteq S$, then if $dec(s) \xrightarrow{\tau} m'$ there exists $s' \in System$ s.t. $s \longrightarrow s'$ and $dec(s') = m'$.*
Proof *We proceed by case analysis on the type of the transition τ. We discuss here only the case of transition of type $IN(t, e, Q)$.*
If $\tau = IN(t, e, Q)$ then $dec(s) = e \oplus \{in\ t(\vec{x}).Q\} \oplus m$, for some m, and $m' = dec(Q[e/\vec{x}]) \oplus m$. By considering the function dec the system s must be of the form $[in\ t(\vec{x}).Q|P, e \oplus DS]$ for some P and DS where $dec([P, DS]) = m$. According with Table 1, we have that $s = [in\ t(\vec{x}).Q|P, e \oplus DS] \rightarrow [Q[e/\vec{x}]|P, DS]$ whose decomposition corresponds with m'. ■

Theorem 3.9 *Let $s \in System$ be a* `LinCa` *system.*

- Soundness: *if $dec(s) \longrightarrow m_1 \ldots \longrightarrow m_n$ in $Net(s)$ then there exist $s_1 \ldots s_n$ such that $s \longrightarrow s_1 \ldots \longrightarrow s_n$ and $dec(s_i) = m_i$ for $i = 1 \ldots n$;*
- Completeness: *if $s \longrightarrow s_1 \ldots \longrightarrow s_n$ then $dec(s) \longrightarrow dec(s_1) \ldots \longrightarrow dec(s_n)$ in $Net(s)$.*

Proof *By induction on n: by exploiting Proposition 3.8 for Soundness and Proposition 3.7 for Completeness.* ■

Corollary 3.10 *Termination is decidable in* `LinCa` *systems.*
Proof *It directly follows from the decidability of deadlock in finite P/T nets, and by the fact that our encoding of* `LinCa` *systems into finite P/T nets defines a one-to-one mapping from finite maximal computations of the* `LinCa` *system into finite maximal firing sequences in the corresponding net.* ■

4 `ProbLinCa`: the Calculus with Probabilistic Data Retrieval

In this section we consider the probabilistic interpretation of the quantitative labels. More precisely, the quantitative information is interpreted as a *weight* that contributes in the definition of a probabilistic distribution indicating the probability for a tuple to be returned as the result of a data retrieval operation. The principle we follow is the usual one: *the greater is the weight of a tuple, the higher is the probability for that tuple to be retrieved.* We start by discussing the probabilistic model that we will adopt and then we formally define the semantics of the `ProbLinCa` calculus.

4.1 Probabilistic model

In Tuple Space coordination languages a probabilistic choice among entities reacting to a given communication request (e.g. tuples matching a "rd" or "in" request) requires a much more complex mechanism w.r.t. languages employing channel-based communication (like, e.g., those representable by standard process algebras). This is due to the greater complexity of the Linda-like matching-based communication mechanism w.r.t. such a simpler form of communication. If we had channel-based communication then we could just consider probability distributions (i.e. functions assigning probabilities that sum up to 1 to elements of a given domain) over the messages $a(\vec{d})$ actually available on the channel type a; when a receive operation is performed on the channel of type a, the channel would "react" to the request by simply choosing a message $a(\vec{d})$ for some d according to such a probability distribution (this is what would happen by adopting either the "reactive model" of probability [25] or the "simple model" of [22]). When we consider the Linda-like matching-based communication mechanism, we loose the separation above between the channel type (which decides the set of entities involved in the communication) and the datum d that is read. Since the set of matching tuples $\langle\vec{d}\rangle$ is now established from a template t on data that is chosen by the "rd" or "in" operation, it is unavoidable to deal with the situation in which the set of matching tuples is a proper subset of the domain of a probability distribution: in this case a re-normalization of "probability" values must be done in order to have them summing up to 1 (this is the same situation that arises for the restriction operator in generative models [25]). Note that the only way to avoid this would be to have an individual probability distribution for each datum $\langle\vec{d}\rangle$ that is present in the shared space (over the several instances of such datum), i.e. by treating each different datum $\langle\vec{d}\rangle$ in the same way as a channel type a in channel-based communication. However, since in this case the channel type would coincide with the datum that is read from the tuple space, reading (or consuming) different tuples having the same channel type (i.e. different instances of the same datum $\langle\vec{d}\rangle$) would have the same observable effect on the system, hence probability distributions would be useless.

As a consequence of this remark, since when the shared space is accessed the probabilities on matching tuples must be determined by using re-normalization (on the basis of the "selecting power" of the template in the "rd" or "in" operation), it is natural to express probability statically associated to tuples in the space by means of *weights* [24]. Usually, a weight is a positive real number which is associated to an entity that can be involved in a probabilistic choice: the actual probability that the entity assumes in the choice is determined from the associated weight depending on the context, i.e. from the weights of the other entities involved in the choice, by re-normalization. In our case, we use the quantitative label (that is a positive natural number) to identify the weight of a tuple.

Example 4.1 *We indicate the weight w of a tuple associating the notation $[w]$ to the tuple itself. Let us suppose that the tuple space contains three tuples $\langle m_1, m_2\rangle[w]$, $\langle m_1, m_2'\rangle[w']$ and $\langle m_1', m_2''\rangle[w'']$, then the following happens. If the operation $rd\ \langle null, null\rangle(x_1, x_2).P$ is performed, the variables x_1, x_2 are bound: to m_1, m_2 with probability $w/(w+w'+w'')$, to m_1, m_2' with probability $w'/(w+w'+w'')$, and to m_1', m_2'' with probability $w''/(w+w'+w'')$. If the operation $rd\ \langle m_1, null\rangle(x_1, x_2).P$ is performed, the variables x_1, x_2 are bound: to m_1, m_2 with probability $w/(w+w')$ and to m_1, m_2' with probability $w'/(w+w')$. If the operation $rd\ \langle m_1, m_2\rangle(x_1, x_2).P$ is performed, the variables x_1, x_2 are bound: to m_1, m_2 with probability $w/w = 1$.*

Moreover note that since the structure of the shared space is highly dynamic and tuples are introduced individually in the space, expressing weights associated to tuples seems to be preferable w.r.t. expressing a single probabilistic distribution over all tuples (generative approach of [25]) which is to be updated by re-normalization to value 1 every time a tuple is added or removed. Therefore, due to the inherent re-normalization behavior of Linda, and to the observations we have made, we adopt, like in [4,3], the approach above based on weights.

Example 4.2 Service registry with workload distribution - *We extend the Example 2.3 by programming a discovery service which supports a balanced workload distribution. Weights are used to express the current workload of services: the higher is the workload, the lower is the weight. Thus, the registration procedure must now take into account the workload of the service, represented by a weight w:*

$register(s, pl, w) \stackrel{def}{=} out(\langle s; pl\rangle[w])$

In order to discover a service supplying task s services can use the following procedure:

$discover(s) \stackrel{def}{=} rd\ \langle s; null\rangle(x_1; x_2)$

which is exactly the one defined in Example 2.3. In this way services with the lowest workload have the highest probability to be discovered, thus obtaining a workload distribution accordingly with the probability distribution on the accesses to the tuples.

4.2 The Probabilistic Semantics

$$(1)\quad [out\ (e).P, DS] \longrightarrow [P, DS \oplus e]$$

$$(2)\quad \frac{\exists e \in DS : e \rhd t}{[in\ t(\vec{x}).P, DS] \longrightarrow \rho^{p}_{in\ t(\vec{x}).P,DS}}$$

$$(3)\quad \frac{\exists e \in DS : e \rhd t}{[rd\ t(\vec{x}).P, DS] \longrightarrow \rho^{p}_{rd\ t(\vec{x}).P,DS}}$$

$$(4)\quad \frac{[P, DS] \longrightarrow \rho}{[P \mid Q, DS] \longrightarrow \rho|Q}$$

$$(5)\quad \frac{[P, DS] \longrightarrow \rho}{[!P, DS] \longrightarrow \rho|!P}$$

Table 3
Semantics of `ProbLinCa`.

The syntax of the `ProbLinCa` calculus is the same as the syntax defined in Subsection 3.1. The semantics replaces the standard non-deterministic choice of a tuple among the matching ones in the TS, with a probabilistic choice exploiting weights.

We consider probability distributions taken from the following set $Prob$:
$Prob = \{\rho \mid \rho : System \longrightarrow [0,1] \wedge supp(\rho) \text{ is finite } \wedge \sum_{s \in System} \rho(s) = 1\}$ where $supp(\rho) = \{s \mid \rho(s) > 0\}$.

The operational semantics of `ProbLinCa` is defined in terms of a probabilistic transition systems $(System, Prob, \longrightarrow)$, where $\rightarrow \subseteq System \times Prob$. More precisely, $\longrightarrow$ is the minimal relation satisfying the axioms and rules of Table 3. $(s, \rho) \in \longrightarrow$ (also denoted by $s \longrightarrow \rho$) means that a system s can reach a generic system configuration, say $s' \in System$, with a probability equal to $\rho(s')$. Note that, several probability distributions may be performable from the same state s, i.e. it may be that $s \longrightarrow \rho$ for several different ρ. This means that (like in the simple model of [22]) whenever the system is in state s, first a non-deterministic choice is performed which decides which of the several probability distributions ρ must be considered, then the next configuration is probabilistically determined by the chosen distribution ρ. Note that the non-deterministic choice may, e.g., arise from several concurrent rd operations which probabilistically retrieve data from the tuple-space. We use $s \longrightarrow s'$ to denote $s \longrightarrow \rho$, with ρ the trivial distribution which gives probability 1 to s' and probability 0 to all other configurations.

Table 4 defines: (i) the probability distributions $\rho^{p}_{in\ t(\vec{x}).P,DS}$ and $\rho^{p}_{rd\ t(\vec{x}).P,DS}$ used for in and rd operations, respectively; (ii) the operator $\rho|Q$ that, given ρ, computes a new probability distribution that accounts for composition with "Q". It is worth noting that $\rho^{p}_{in\ t(\vec{x}).P,DS}$ and $\rho^{p}_{rd\ t(\vec{x}).P,DS}$ are defined only for $t \in Template$ and $DS \in DSpace$ such that there exists $e \in DS : e \triangleright t$ (that is the condition reported in axioms (2) and (3)). The meaning of such probability distributions, that are used in axioms and rules of Table 3, is commented in the description of the operational semantics that follows.

Axiom (1) describes the output of the tuple e, after the execution an occurrence of e is added to the shared space DS and the process continues with P. Axiom (2) describes the behaviour of in operations; if a tuple e matching with template t is available in the DS, the in execution produces the removal from the space of e and then the process behaves as $P[\tilde{e}/\vec{x}]$. The probability of reaching a configuration where a matching tuple e contained in the DS is removed is the ratio of the total weight of the several instances of e in the DS, to the sum of the total weights of the several instances of the matching tuples currently available in the DS. In this way, the probability to reach a system configuration takes into account the multiple ways of removing e due to the several occurrences of e in the DS. The axiom (3) describes rd operations; if a tuple e matching with template t is available in the DS, then the process behaves as $P[\tilde{e}/\vec{x}]$. Differently from the previous axiom, rd operations do not modify the tuple space, i.e. reached states do not change the configuration of DS, therefore they are simply differentiated by the continuation $P[\tilde{e}/\vec{x}]$ of the reading process. For example, let us consider two different tuples $e = \langle d; d_c \rangle[w]$ and $e' = \langle d'; d_c \rangle[w']$. Let $P = rd\ \langle null; null \rangle(x_1; x_2).out(\langle x_2 \rangle[w])$ be the process performing the read; it is not possible to discriminate the selection of the two different tuples. Therefore, the probability of reaching a configuration s that is obtained by reading a tuple e matching with t in the DS (yielding value e) is the ratio of the sum of the total weights associated to the several instances of tuples e' matching with t in the DS such that the continuation of the reading process obtained by reading tuple e' is the same as the one obtained by reading e, to the sum of the total weights of the several instances

$$\rho^{p}_{in\ t(\vec{x}).P,DS}(s) = \begin{cases} \dfrac{QL(e) \cdot DS(e)}{\sum_{e' \in DS : e' \rhd t} QL(e') \cdot DS(e')} & if\ \ s = [P[\tilde{e}/\vec{x}], DS - e] \\ & \quad with\, e \in DS \wedge e \rhd t \\ & \\ 0 & o.w. \end{cases}$$

$$\rho^{p}_{rd\ t(\vec{x}).P,DS}(s) = \begin{cases} \dfrac{\sum_{e' \in DS : e' \rhd t \wedge P[\tilde{e'}/\vec{x}] = P[\tilde{e}/\vec{x}]} QL(e') \cdot DS(e')}{\sum_{e' \in DS : e' \rhd t} QL(e') \cdot DS(e')} & if\ \ s = [P[\tilde{e}/\vec{x}], DS] \\ & \quad withe \in DS \wedge e \rhd t \\ & \\ 0 & o.w. \end{cases}$$

$$\rho | Q(s) = \begin{cases} \rho([P', DS]) & if\ \ s = [P' \mid Q, DS], \\ & \quad P' \in Process \wedge\ DS \in DSpace. \\ & \\ 0 & o.w. \end{cases}$$

Table 4
Probability distributions.

of the matching tuples currently available in the DS. Rule (4) describes the behaviour of the parallel composition of processes (the symmetric rule is omitted): if configurations reachable from $[P, DS]$ are described by the probability distribution ρ, and P performs an action in the system $[P \mid Q, DS]$ (the process that proceeds between P and Q is non-deterministically selected), then the reachable configurations are of the form $[P' \mid Q, DS']$, for some $P' \in Process$ and $DS' \in DSpace$. The probability values of such configurations do not depend on Q (that is "inactive") and are equal to $\rho([P', DS'])$. Finally, rule (5) describes the behaviour of process replication operator: $!P$ behaves as an unbounded parallel composition of the process P.

4.3 Leader Election Protocol in `ProbLinCa`

We provide an expressiveness gap also between `ProbLinCa` and `LinCa`. Namely, we prove that in `ProbLinCa` the quantitative information associated to the tuples, interpreted as weights, permits to solve the leader election problem also in symmetric systems. This is not the case for the calculus `LinCa` without quantitative information.

Informally, the leader election problem consists in ensuring that all the processes inside a system will reach an agreement, about the leader of the system, in finite time. In symmetric systems (intuitively, systems composed of processes performing the same protocol) such a problem cannot be solved because a system with a leader is not symmetric, and asynchronous communication usually is not able to force the initial symmetry to break.

Tuple space communication is asynchronous because mediated by the tuple repository. The formal proof of the impossibility to solve the leader election problem in `LinCa` is reported in the Appendix A as it is a simple adaptation of an equivalent proof presented in [17] for the asynchronous π-calculus.

The addition of weights, on the contrary, permits to break the initial symmetry. This is proved by presenting a simple symmetric system (according to Definition A.3) composed of two processes only, which is also an electoral system, i.e. a system in which the leader election problem is solved.

Here we reformulate the protocol of [12,13] by using the calculus `ProbLinCa` instead of probabilistic asynchronous π-calculus as done in [12,13]. Let P_0 and P_1 be the two symmetric processes (see Appendix A) that must interact in order to elect a leader, and $w, w', w_\varepsilon \in QLab$ be weigths, where $w_\varepsilon = \varepsilon \cdot w$ with ε positive real number such that $\varepsilon < 1$. For the sake of simplicity, in the following we will omit weights associated to tuples when their value is not significant.

We assume that the space is initially partitioned in DS_0 and DS_1 which are defined as follows:

$$DS_0 = \langle 0; T; *\rangle[w] \oplus \langle 0; *; unlock\rangle[w_\varepsilon] \oplus \langle choice_0; 0\rangle[w'] \oplus \langle choice_0; 1\rangle[w'] \oplus \langle 0\rangle,$$
$$DS_1 = \langle 1; T; *\rangle[w] \oplus \langle 1; *; unlock\rangle[w_\varepsilon] \oplus \langle choice_1; 0\rangle[w'] \oplus \langle choice_1; 1\rangle[w'] \oplus \langle 1\rangle.$$

We assume that the leader is communicated to the environment by the two processes P_0 and P_1 via a tuple $\langle o; l\rangle$, where $l \in \{0, 1\}$ represents the leader. The idea is that tuples $\langle i; T; *\rangle[w]$ initially contained in the tuple space are "owned" by the corresponding process P_i. When execution starts, each process P_i tries to remove both its own tuple $\langle i; T; *\rangle$ and the tuple $\langle i +_2 1; T; *\rangle$ owned by the other process $P_{i+_2 1}$ (here we use $+_2$ as the sum modulo 2). If it succeeds, it is the leader and communicates this to $P_{i+_2 1}$, by inserting the tuples $\langle i; F; *\rangle$

and $\langle i +_2 1; F; * \rangle$ in the tuple space, and then to the external environment by inserting $\langle o; i \rangle$. If it does not succeed, it repeats such a procedure.

More in the details, since P_i may try to remove tuples $\langle i; T; * \rangle$ and $\langle i +_2 1; T; * \rangle$ only sequentially (we assume that we have no primitive for simultaneous tuple removal), it is fundamental to consider the order of removal of such tuples. The protocol that we will present operates as follows.

First it performs a probabilistic "blind" choice that determines whether the attempt to remove $\langle i; T; * \rangle$ will be done before the attempt to remove $\langle i +_2 1; T; * \rangle$ or, vice versa, the opposite ordering is undertaken. Such a choice is "blind" in the sense that the process commits to the choice of the ordering *before* knowing whether tuples are available. As shown in [12,13], such a commitment is essential for ensuring that the leader will be elected with probability 1 under every possible scheduling of the two processes. Technically, such a probabilistic choice is performed by P_i by reading a tuple matching with $\langle choice_i; null \rangle$: which yields 0 or 1 (which identifies the first tuple that the process will try to remove) in the second field with probability 1/2 and 1/2 (the same weight w' is associated with both $\langle choice_i; 0 \rangle$ and $\langle choice_i; 1 \rangle$ tuples).

Assuming that P_i is able to remove the first tuple, the idea is that now the protocol checks whether the second tuple is available: in the affirmative case the second tuple is removed and P_i declares himself to be the leader, otherwise P_i puts back the tuple he just removed in the tuple space and restarts its protocol. Such "idealized behavior" is however only approximated by the protocol that we consider. This because, removing the tuple $\langle i +_2 1; T; * \rangle$ (or $\langle i; T; * \rangle$ depending on the outcome of the initial probabilistic blind choice) in the case it is available and performing something else otherwise – technically we remove the tuple $\langle i +_2 1; *; unlock \rangle$ (or $\langle i; *; unlock \rangle$) – would require a prioritized mechanism where $\langle i +_2 1; T; * \rangle$ and $\langle i; T; * \rangle$ are assigned a higher priority than $\langle i +_2 1; *; unlock \rangle$ and $\langle i; *; unlock \rangle$. As in [12] we approximate this behavior by using a probabilistic choice where the unprioritized behavior has very low probability (we use weight w_ε) while the prioritized behavior has very high probability (we use weight w). The closer to zero the value of ε is, the more efficient the algorithm is: nevertheless $\varepsilon < 1$ is sufficient to guarantee that the system will eventually elect a leader with probability 1.

Whenever a process P_i succeeds in removing both tuples $\langle i; T; * \rangle$ and $\langle i +_2 1; T; * \rangle$, he declares himself to be the leader and communicates this event to the other process by putting the tuples $\langle i; F; * \rangle$ and $\langle i +_2 1; F; * \rangle$ in the tuple space. The other way around, whenever a process P_i reads a tuple $\langle i; F; * \rangle$ or $\langle i +_2 1; F; * \rangle$, he declares the other process to be the leader.

Concerning the use of special value "$*$" in the three-valued tuples: the second field is "$*$" (this tuple is not part of the information exchanged for establishing

$P_i =$

$!in\ \langle i \rangle (x).rd\ \langle choice_i ; null \rangle (x_1 ; x_2).$

$\quad$ if $(x_2 = i)$ then

$\quad\quad in\ \langle i; null; * \rangle (x_3; x_4; x_5).$

$\quad\quad$ if $(x_4 = T)$ then

$\quad\quad\quad in\ \langle i +_2 1; null; null \rangle (x_6; x_7; x_8).$

$\quad\quad\quad$ if $(x_7 = T)$ then

$\quad\quad\quad\quad out(\langle i; F; * \rangle).out(\langle i +_2 1; F; * \rangle).out(\langle o; i \rangle)$

$\quad\quad\quad$ else

$\quad\quad\quad\quad out(\langle i +_2 1; *; unlock \rangle [w_\varepsilon]).out(\langle i; T; * \rangle [w]).out(\langle i \rangle)$

$\quad\quad$ else

$\quad\quad\quad out(\langle o; i +_2 1 \rangle)$

$\quad$ else

$\quad\quad in\ \langle i +_2 1; null; * \rangle (x_3; x_4; x_5).$

$\quad\quad$ if $(x_4 = T)$ then

$\quad\quad\quad in\ \langle i; null; null \rangle (x_6; x_7; x_8).$

$\quad\quad\quad$ if $(x_7 = T)$ then

$\quad\quad\quad\quad out(\langle i +_2 1; F; * \rangle).out(\langle i; F; * \rangle).out(\langle o; i \rangle)$

$\quad\quad\quad$ else

$\quad\quad\quad\quad out(\langle i; *; unlock \rangle [w_\varepsilon]).out(\langle i +_2 1; T; * \rangle [w]).out(\langle i \rangle)$

$\quad\quad$ else

$\quad\quad\quad out(\langle o; i +_2 1 \rangle)$

Table 5
Leader Election Protocol for P_i.

the leader) whenever the third is "*unlock*"; the third field is "$*$" (this tuple is not used for avoiding deadlock) whenever the second field is "T" or "F".

In order to make more intelligible the protocol, we use a *if-then-else* construct which is just a shorthand for a simple `ProbLinCa` expression. In particular, assumed that x is a variable that can assume values v_1 and v_2 only, we use:
if $(x = v_1)$ then P else Q
to stand for:
$out(\langle cnt; x \rangle) \quad | \quad !in\ \langle cnt; v_1 \rangle (y_1; y_2).P \quad | \quad !in\ \langle cnt; v_2 \rangle (y_1; y_2).Q$
where "*cnt*" stands for "continuation" must be a different name for each syntactical occurrence of the construct *if-then-else* in the protocol.

The protocol for process P_i $(i = 0, 1)$ is defined in Table 5.

The system $[P_0 \mid P_1, DS_0 \oplus DS_1]$, where P_i with $i \in \{0, 1\}$ are defined as in Table 5 and DS_i with $i \in \{0, 1\}$ are the two initial spaces defined above, eventually elects a leader with probability one under every possible scheduling of processes P_1 and P_2. The proof of this fact can be found in [13] in a language independent format.

5 Conclusion

We divide this conclusive section in two subsections; the former reports a summary of the results proved in this paper, the latter discusses the relevance of our results with respect to some of the most significant tuple based coordination languages.

5.1 Summary of the results

We have investigated, from an expressiveness viewpoint, the impact of the introduction of quantitative information in the tuple space coordination model. More precisely, we have considered the possibility to decorate each tuple with a quantitative label. We have assumed two possible interpretations for such a label: either as a weight supporting a probabilistic access to the tuple space (the greater is the weight of a tuple, the higher is the probability for that tuple to be returned) or as a priority level supporting a prioritized access. After an informal discussion of applications for which this kind of tuple space access mechanisms seems particularly suited, we have formally discussed its expressive power.

A formal investigation of the expressive power requires a formal representation of the considered access mechanisms. We have achieved this representation by embedding the coordination primitives into a minimal process algebra. This algebra comprises only three operators: the prefix operator used to associate a continuation to the execution of a primitive; the parallel operator used to compose the processes that coordinate via the tuple space; the replication operator used as the unique infinite operator.

Namely, we have introduced three process algebras: `LinCa` which models the standard tuple space coordination model, `ProbLinCa` which introduces the probabilistic access mechanism, and `PrioLinCa` which consider the prioritized access. In this section we discuss also a fourth process algebra, that we do not formally define, which combines both the access mechanisms. This algebra,

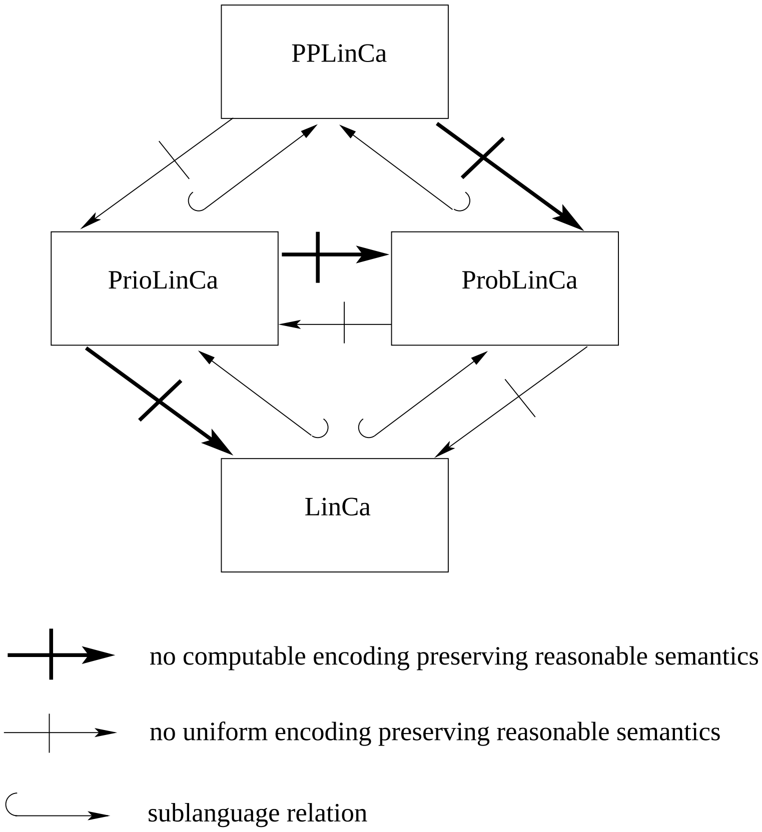

Fig. 1. Overview of the results.

that we denote with PPLinCa, is obtained associating two quantitative labels to the tuples, one interpreted as a priority and the other one as a weight. Data are retrieved probabilistically according to the weights of the tuples with the highest priority among the matching ones.

Figure 1 presents an overview of results regarding the impossibility to provide encodings among the several considered algebras. An encoding is a function from one algebra to another one that preserves some intended semantics. The remainder of this subsection is devoted to the discussion of these impossibility results.

We have proved that termination is decidable in LinCa while this is not the case in PrioLinCa. Moreover, we have proved that the leader election problem for symmetric systems can be solved in ProbLinCa while it cannot be solved in LinCa (see Appendix A).

Leader election has been exploited also in [17] to prove the impossibility to provide a *uniform encoding* from the synchronous to the asynchronous π-calculus that preserves any *reasonable semantics*. Informally, an encoding is uniform when it is a homomorphism w.r.t. parallel composition and preserves name substitution; a semantics is reasonable if it distinguishes two processes, say P and Q, whenever in some computation of P the relevant observable actions are different from those of any computation of Q. This impossibility result holds also between `ProbLinCa` (the calculus with probabilities) and `LinCa` (the calculus representing the standard tuple space model).

The gap of expressiveness that we have proved between the model with priorities and the standard tuple space model has the following consequence: there exists no computable encoding from `PrioLinCa` (the calculus with priorities) to `LinCa` that preserves any reasonable semantics. This follows from the fact that a reasonable semantics is able to discriminate between a faithful encoding of RAMs (such as the one we present in Subsection 3.3) and a nonfaithful encoding. It is enough to consider RAMs that make observable their termination and by formalizing faithfulness as follows: all the computations of the encoding of a RAM R terminate if and only if R terminates. Such an encoding cannot be provided in `LinCa` as we have proved that termination is decidable in this calculus.

Another consequence of our results is the impossibility to provide encodings from the algebra with priority to the algebra with probability, and vice versa. First of all, we observe that the leader election problem cannot be solved in symmetric systems neither in `PrioLinCa`; indeed, the proof reported in Appendix A applies also to the calculus with priorities. Thus we have that it is not possible to provide a uniform encoding that preserves any reasonable semantics from `ProbLinCa` to `PrioLinCa`.

Moreover, we have that the P/T net semantics we have used in Subsection 3.4 to prove that termination is decidable in `LinCa`, permits to prove that also reachability is decidable: given the systems s and s' the reachability problem consists in verifying whether there exists a computation from s to s'. The P/T net semantics can be easily adapted to the calculus with probabilities, thus proving that reachability is decidable also for `ProbLinCa`. On the other hand, reachability cannot be decided in `PrioLinCa` (this is a direct consequence of Theorem 3.3). As a reasonable semantics distinguishes between two processes that have different sets of reachable states, we have that there exists no computable encoding of `PrioLinCa` into `ProbLinCa` that preserves any reasonable semantics.

This incomparability result between the prioritized and the probabilistic models permits to conclude that the calculus `PPLinCa` comprising both of them is strictly more expressive than both `PrioLinCa` and `ProbLinCa`.

We have embedded the coordination primitives into a minimal process algebraic language. In this last subsection we analyse how our results can be applied to those real world coordination languages obtained embedding the coordination primitives in richer (Turing powerful) computational languages.

Let us consider the following notion of implementability of a coordination model M into a coordination language L. We say that M is implementable in L if any system designed according to the coordination model M can be implemented using the language L simply by translating the coordination actions of the system components into coordination actions expressed with the primitives provided by L. In other terms, the implementation should not alter the structure of the overall application and should not affect the internal behaviour of the components.

The impossibility to provide uniform encodings from the calculi `PrioLinCa` and `ProbLinCa` into `LinCa` implies that the tuple space coordination model enhanced with either prioritized or probabilistic tuple retrieval cannot be implemented into any standard Linda coordination language with only output, read, and input coordination primitives.

We now consider the Linda language in [9] that comprises also the *inp* and *rdp* primitives. These are non-blocking versions of *in* and *rd* which return the boolean value $false$ if no tuple can be actually retrieved (i.e. no tuple matches the considered template). In this way, the absence of tuples matching a certain template can be tested. The prioritized model could be programmed using test for absence primitives as follows. The level of priority could be associated to the tuples as an extra field. The processes that access the space could perform an inp/rdp taking into account the first level of priority, and passing to the subsequent levels only if no tuples are retrieved. Obviously, this approach is satisfactory only if few levels of priority are considered, because it is necessary to explicitly access one level at a time.

On the contrary, Linda languages with test for absence cannot implement the model with probability. This can be proved considering previous results in [26], where it is shown that a process calculus similar to `LinCa` is not expressive enough for solving the leader election problem in symmetric systems even when a test for absence operator is added.

We now consider another typical extension of Linda languages with group primitives. These primitives, such as the *collect* primitive of [21], permit to withdraw all the tuples satisfying a template. Group primitives permits to implement the probabilistic model. Each tuple could be decorated with an extra field that quantifies its weight. When a tuple retrieval operation is executed from a process, this process collects all the tuples satisfying the template, select the tuple according to their weights, and re-introduce the tuples in the space. This implementation is satisfactory only for small sized tuple spaces because it requires to move from the tuple space to the process (and back) all the tuples matching the template; moreover this double transfer of tuples should be executed in a transactional manner, thus requiring consistent locks.

On the contrary, Linda languages with group primitives cannot implement the model with prioritized access to the tuple space. This can be proved considering previous results in [6], where it is shown that a process calculus similar to `LinCa` is not expressive enough for modeling faithfully RAMs even when a *collect* operator is added.

The Linda extensions discussed so far permit to implement either the probabilistic or the prioritized model, but in a rather unsatisfactory way. The main problem is that the implementation of each probabilistic or prioritized access requires several (in some cases expensive) interactions between the processes and the tuple space. This problem can be fixed moving to coordination languages such as MARS [7] and TuCSoN [16] which permit to program the tuple space behaviour. More precisely, processes can introduce programs in the tuple space that are triggered by events such as the execution of a specific coordination primitive. Clearly, using coordination languages with this feature it is possible to program the space in order to react to the execution of an in or a rd primitive by executing a prioritized or a probabilistic tuple retrieval.

A final remark regards Probabilistic Klaim [19], the unique calculus to the best of our knowledge that addresses probabilities in the context of Linda-like languages. Klaim [15] is a distributed mobile version of Linda where tuple spaces and processes are associated to nodes in a net, and the processes may migrate from one node to another one. In Probabilistic Klaim probability comes into play at the global level (when the process to move must be selected) as well at the local level (when the specific action to be executed must be chosen). Moreover, the approach adopted in Probabilistic Klaim follows more faithfully the tradition of process algebras, according to which a probability is associated to each branch of a choice or to each process inside a parallel composed term. Our probabilistic model simply permits to associate a weight to the tuples when they are created; on the other hand, in Probabilistic Klaim the tuples are produced with an associated probability equal to one. As a future work, we plan to investigate whether the schedule-driven approach of Probabilistic Klaim is expressive enough for modeling our data-driven approach.

References

[1] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* John Wiley and Sons, Inc., 2004. Second Edition.

[2] JP. Banatre and D. LeMetayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, January 1993.

[3] M. Bravetti. *Specification and Analysis of Stochastic Real-Time Systems.* PhD thesis, Dottorato di Ricerca in Informatica. Università di Bologna, Padova, Venezia, February 2002. Available at `http://www.cs.unibo.it/~bravetti/`.

[4] M. Bravetti and M. Bernardo. Compositional asymmetric cooperations for process algebras with probabilities, priorities, and time. In *Proc. of the 1st Int. Workshop on Models for Time-Critical Systems, MTCS 2000,* State College (PA), volume 39(3) of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2000.

[5] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156(1-2):90–121, January 2000.

[6] N. Busi, A. Rowstron, and G. Zavattaro. State- and event-based reactive programming in shared dataspaces. In *Coordination Languages and Models (COORDINATION'02)*, volume 2315 of *LNCS*, pages 111–124. Springer-Verlag, 2002.

[7] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive tuple spaces for mobile agent coordination. In *2nd Int. Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 237–248. Springer-Verlag, 1998.

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

[9] Scientific Computing Associates. *Linda: User's guide and reference manual.* Scientific Computing Associates, 1995.

[10] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Pattern, and Practice.* Addison-Wesley, 1999.

[11] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[12] O.M. Herescu and C. Palamidessi. Probabilistic asynchronous pi-calculus. In *Proc. of FoSSaCS*, volume 1784 of *LNCS*, pages 146–160. Springer-Verlag, 2000.

[13] O.M. Herescu and C. Palamidessi. Probabilistic asynchronous pi-calculus. Technical report, Penn State University, 2000. Available at `http://www.cse.psu.edu/~catuscia/papers/Prob_asy_pi/report.ps`.

[14] M.L. Minsky. *Computation: finite and infinite machines.* Prentince Hall, Englewood Cliffs, 1967.

[15] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998. Special Issue: Mobility and Network Aware Computing.

[16] A. Omicini and F. Zambonelli. Coordination of mobile information agents in TuCSoN. *Journal of Internet Research*, 8(5):400–413, 1998.

[17] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Proc. of POPL*, pages 256–265, 1997. Full version appeared in Mathematical Structures in Computer Science, 13(5): 685-719, 2003.

[18] G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46:329–400, 1998.

[19] A. Di Pierro, C. Hankin, and H. Wiklicky. Probabilistic Klaim. In *Proc. of 7th International Conference on Coordination Models and Languages (Coordination 04)*, volume 2949 of *LNCS*, pages 119–134. Springer Verlag, 2004.

[20] C. Reutenauer. *The Mathematics of Petri Nets.* Masson, 1988.

[21] A. Rowstron and A. Wood. Solving the Linda multiple `rd` problem using the `copy-collect` primitive. *Science of Computer Programming*, 31(2-3):335–358, 1998.

[22] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995.

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

[24] C.M.N. Tofts. Processes with probabilities, priority and time. *Formal Aspects of Computing*, 6:534–564, 1994.

[25] R.J. van Glabbeek, S.A. Smolka, and B. Steffen. Reactive, Generative and Stratified Models of Probabilistic Processes. *Information and Computation*, 121:59–80, 1995.

[26] G. Zavattaro. Towards an Hierarchy of Negative Test Operator for Asynchronous Communication. In *Proc. Workshop of Expressiveness in Concurrency (EXPRESS'98)*, volume 16(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

A Unsolvability of the Leader Election Problem in `LinCa`

In this appendix we present the formal discussion about the unsolvability of the leader election problem for symmetric systems expressed in the `LinCa` calculus. We adapt to our context the proof reported in [17] developed for the asynchronous π-calculus.

In order to formalize the notion of symmetric systems we proceed as follows: we define a mechanism for associating a hypergraph to any `LinCa` systems, and we characterize symmetric systems in terms of the class of hypergraphs they are associated to.

We start reporting the basic notions about hypergraphs and automorphisms.

Definition A.1 ***Hypergraph and automorphism -*** *An hypergraph H is defined by a triple (N, X, t) where N and X are finite sets defining the set of nodes and of hyperarcs, respectively, and t is a function which maps each $x \in X$ into a set of nodes.*
An automorphism σ of H defined as above is a pair (σ_N, σ_X) where $\sigma_N : N \longrightarrow N$ and $\sigma_X : X \longrightarrow X$ are permutations such that, for each $x \in X$, if $t(x) = \{n_1, \ldots, n_k\}$ then $t(\sigma_X(x)) = \{\sigma_N(n_1), \ldots, \sigma_N(n_k)\}$. The composition of two automorphisms is still an automorphism.
The orbit of $n \in N$ generated by σ is defined as $O_\sigma(n) = \{n, \sigma(n), \ldots, \sigma^h(n)\}$ in which σ^k represents σ composed with itself k times and h is such that $\sigma^{h+1}(n) = id$.

Now, we describe how to associate an hypergraph to any `LinCa` system. In the following, we assume that natural numbers are contained in $Mess$. We also consider fixed partitionings of the data space and we use the notation ω to denote a partitioning such as $DS_1, DS_2, \ldots, DS_k$.

Definition A.2 ***Hypergraph associated to a system -*** *Let ω be the partitioning $DS_1, DS_2, \ldots, DS_k$ and let $s = [P_1|P_2|\ldots|P_k, DS_1 \oplus DS_2 \ldots \oplus DS_k]$ be a `LinCa` system. The hypergraph associated to s with partitioning ω is $H(s, \omega) = (N, X, t)$ where $N = \{1, \ldots, k\}$, $X = df(P_1|P_2|\ldots|P_k) \cup df(DS_1 \oplus DS_2 \ldots \oplus DS_k)$ and, for each $x \in X$, $t(x) = \{i \in N \mid x \in df(P_i) \cup df(DS_i)\}$.*

We recall that *df* extracts the data fields from processes as well as data spaces. Let $\sigma = (\sigma_N, \sigma_X)$ be an automorphism on the hypergraph associated with the system $[P, DS]$, we use $\sigma(P)$ (resp. $\sigma(DS)$) to denote the process (resp. the space) obtained by renaming each data field d occurring in P (resp. DS) $\sigma_X(d)$.

Definition A.3 ***Symmetric system -*** *Let ω be the partitioning $DS_1, DS_2, \ldots, DS_k$ and let $s = [P_1|P_2|\ldots|P_k, DS_1 \oplus DS_2 \ldots \oplus DS_k]$ be a `LinCa` system. Let σ be an automorphism on the hypergraph $H(s, \omega) = (N, X, t)$. We say that s is symmetric w.r.t. σ and ω iff, for each $i \in N$, $P_{\sigma(i)}$ is equal to $\sigma(P_i)$ (up to α-renaming) and $DS_{\sigma(i)} = \sigma(DS_i)$ hold. The system s is symmetric w.r.t. ω if it is symmetric w.r.t. all the automorphisms on $H(s, \omega)$.*

We use the term computation to identify a specific sequence of transitions of a system. For example $s \longrightarrow s_1 \ldots \longrightarrow s_n$ is a computation of s. Let C and C' be two computations of a system s; we say that C' extends C if $C' = s \longrightarrow s_1 \ldots \longrightarrow s_n \longrightarrow s_{n+1} \ldots \longrightarrow s'$ and $C = s \longrightarrow s_1 \ldots \longrightarrow s_n$. If C is an infinite computation there exists no *strict* extension of C.

Let $s = [P_1|P_2|\ldots|P_k, DS]$ be a system and C a computation of s; we use $Tuple(C, P_i)$ as the function which returns the set of tuples that are inserted by P_i (and its derivatives) in the space during the computation C, for $i \in \{1, \ldots, k\}$.

An electoral system guarantees that all the processes in the system eventually elect the same leader; the choice of the leader is done by producing the tuple $\langle o; m\rangle$ where m is the identifier of the selected leader.

Definition A.4 ***Electoral system*** **-** *Let $s = [P_1|P_2|\ldots|P_k, DS]$ be a system; s is an electoral system if for every computation C there exists an extension C' of C and $n \in \{1, \ldots, k\}$ (the 'leader') such that $\langle o; n\rangle \in Tuple(C', P_i)$ and there exists no extension C'' of C' in which $\langle o; m\rangle \in Tuple(C'', P_i)$ for any $i \in \{1, \ldots, k\}$ and $m \neq n$.*

We are now ready to prove the impossibility to have in `PrioLinCa` electoral systems that are also symmetric systems. The prove is restricted to a specific class of symmetric systems, namely those with only one orbit (the extension to all symmetric systems can be done as discussed in the Corollary 4.3 of [17]).

Proposition A.5 *Let ω be the partitioning $DS_1, DS_2, \ldots, DS_k$ and let $s = [P_1|P_2|\ldots|P_k, DS_1 \oplus DS_2 \ldots \oplus DS_k]$ be a* `LinCa` *system. If the associated hypergraph $H(s, \omega)$ admits an automorphism $\sigma \neq id$ with only one orbit, and s is symmetric w.r.t. σ and ω, then s cannot be an electoral system.*

Proof *We assume by contradiction that s is an electoral system. We will show that it is possible to build an infinite sequence of computations C_1, C_2, $\ldots$, C_n, $\ldots$, such that C_{i+1} extends C_i and the system reached by the computation C_i is symmetric w.r.t. σ and some ω_i (updated in order to take into account the possible data exchanged during the computation) and does not contain any tuple of the form $\langle o; n\rangle$, for any i and n, thus proving that s cannot be an electoral system.*

We proceed by induction on n. We consider the empty computation for the case $n = 1$. For the inductive case we consider C_n and we describe how to define C_{n+1}. Let $s_n = [P_1^n|P_2^n|\ldots|P_k^n, DS^n]$ be the system reached by the computation C_n, which is symmetric w.r.t. σ and the partitioning $\omega_n = DS_1^n, DS_2^n, \ldots, DS_k^n$. Since s is an electoral system and, by hypothesis, $\langle o; l\rangle \notin Tuple(C_n, P_i)$ for any i and l, there must exists an extension of C_n that produces the tuples indicating the leader. Now we exploit the fact that the hypergraph associated to s has only one orbit: in this case $O_\sigma(i) = \{i, \sigma(i), \ldots, \sigma^{k-1}(i)\} = \{1, \ldots, k\}$ for any

$i \in \{1, \dots, k\}$.
Let P_i^n be the process that moves and DS_j^n be the subspace in which it acts, that is there exists a transition $\left[P_i^n, DS_j^n\right] \longrightarrow \left[P_i^{n\prime}, DS_j^{n\prime}\right]$. Observe that P_i^n cannot perform the output of $\langle o; l\rangle$ for some l. If this is possible then by symmetry $P_{\sigma(i)}^n$ (which is equal to $\sigma(P_i^n)$) must be able to produce the tuple $\langle o; \sigma(l)\rangle$. Since s is an electoral system there must be $l = \sigma(l)$ and, given that σ generates one orbit only, $\sigma = id$ which is a contradiction.
By symmetry there exist also the following transitions:

$$\left[P_{\sigma(i)}^n, DS_{\sigma(j)}^n\right] \longrightarrow \left[{P_{\sigma(i)}^n}', {DS_{\sigma(j)}^n}'\right]$$
$$\left[P_{\sigma^2(i)}^n, DS_{\sigma^2(j)}^n\right] \longrightarrow \left[{P_{\sigma^2(i)}^n}', {DS_{\sigma^2(j)}^n}'\right]$$
$$\dots$$
$$\left[P_{\sigma^{k-1}(i)}^n, DS_{\sigma^{k-1}(j)}^n\right] \longrightarrow \left[{P_{\sigma^{k-1}(i)}^n}', {DS_{\sigma^{k-1}(j)}^n}'\right].$$

Note that the process P^n is the parallel composition of $P_i^n, P_{\sigma(i)}^n, \dots, P_{\sigma^{k-1}(i)}^n$. For the sake of simplicity we assume $P_n = P_i^n | P_{\sigma(i)}^n \dots | P_{\sigma^{k-1}(i)}^n$ (any different case is treated similarly). Moreover, $DS^n = DS_j^n \oplus DS_{\sigma(j)}^n \dots \oplus DS_{\sigma^{k-1}(j)}^n$. According with rule 4 of Table 1 the following computation can be performed:

$$\begin{aligned}
&\left[P_i^n | P_{\sigma(i)}^n \dots | P_{\sigma^{k-1}(i)}^n, DS_j^n \oplus DS_{\sigma(j)}^n \dots \oplus DS_{\sigma^{k-1}(j)}^n\right] \longrightarrow \\
&\quad \left[P_i^{n\prime} | P_{\sigma(i)}^n \dots | P_{\sigma^{k-1}(i)}^n, DS_j^{n\prime} \oplus DS_{\sigma(j)}^n \dots \oplus DS_{\sigma^{k-1}(j)}^n\right] \longrightarrow \\
&\quad\quad \dots \longrightarrow \\
&\quad\quad\quad \left[P_i^{n\prime} | {P_{\sigma(i)}^n}' \dots | {P_{\sigma^{k-1}(i)}^n}', DS_j^{n\prime} \oplus {DS_{\sigma(j)}^n}' \dots \oplus {DS_{\sigma^{k-1}(j)}^n}'\right].
\end{aligned}$$

The reached system is still symmetric w.r.t. σ and the partitioning $\omega_{n+1} = DS_1^{n+1}, DS_2^{n+1}, \dots, DS_k^{n+1}$ and does not contain any tuple of the form $\langle o; l\rangle$, for any $l \in \{1, \dots, k\}$. ■