# Secure shared data-space Coordination Languages: a Process Algebraic survey [⋆]

Riccardo Focardi

*Dipartimento di Informatica, Università Cà Foscari di Venezia*
*Via Torino 155, I-30172 Mestre (Ve), Italy*
*e-mail:focardi@dsi.unive.it*

Roberto Lucchi Gianluigi Zavattaro

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

**Abstract**

Shared data-space coordination languages, which provide a means to program interactions between decoupled entities abstracting away from their internal behavior, represent a powerful framework for programming network applications over the Internet and, in general, in open systems where the entities involved are not known *a priori*. In this context, where programs may run in an untrusted environment, new challenges come into play such as to provide a means to support security. In this paper we outline the most significant security threats emerging in this context and we present a survey, in a process algebraic setting, of the most interesting shared data-space coordination languages.

## 1 Introduction

New networking technologies are calling for the definition of models and languages adequate for the design and management of new classes of applications. Innovations are moving towards two directions. On the one hand, the Internet is the candidate environment for supporting the so-called wide area applications. On the other hand, smaller networks of mobile and portable devices, such as mobile ad-hoc networks or peer-to-peer systems, support applications

[⋆] Research partially funded by EU Integrated Project Sensoria, contract n. 016004.

based on entities or components which interact according to a dynamically reconfigurable communication structure. In both cases, the challenge is to develop applications without knowing, at design time, the overall structure of the system as well as the entities that will be involved. Indeed, these systems are usually open to new entities or components which are unknown beforehand.

Furthermore, our society is becoming more and more dependent on computer networks: the enormous amount of data that is elaborated, transmitted or stored needs some form of protection. Hence, in order to guarantee some security properties, several procedures based on cryptography (see, e.g. [17]) have been proposed in the literature. The goals of these protocols cover a large area of applications, e.g. privacy and authentication of the messages, personal identification, digital signatures, electronic money transfer, credit card transactions and many other critical applications. Actually, security protocols and the applications exploiting them are typically based according to a channel based topology.

Coordination models and languages, which advocate a distinct separation between the internal behaviour of the entities and their interaction, represent a promising approach for the development of applications for this class of dynamic and open systems. For instance, we assist to a renewed interest in data-driven coordination infrastructures originated by Linda [8] as exemplified by recent commercial products, such as JavaSpaces [18] and TSpaces [20], which are two coordination middlewares for distributed Java [11] programming proposed by Sun Microsystem and IBM, respectively. Both proposals exploit the so-called *generative communication* [8]: a sender communicates with one or more receivers through a shared tuple space (TS for short), where emitted tuples are collected; a receiver can read or consume the tuples from the TS; a tuple generated by an agent has an independent existence in the tuple space until it is explicitly withdrawn by a receiver; in fact, after its insertion in TS, a tuple becomes equally accessible to all entities, but it is bound to none. This form of communication is referred to as generative communication because when a datum is produced, it has an existence which is independent of its producer, and it is equally accessible to all entities.

In this paper we outline the main security threats which occur when the Linda coordination model is used in untrusted environments. Then we classify the most interesting solutions available in the literature in two classes by proposing two abstract calculi which implement access control mechanisms in two different manners: *entity-driven* approach and *knowledge-driven* approach. Essentially, the former one exploits the notion of entity to express access permissions and to govern the accesses to the resources according with system capabilities, while the latter one uses some reserved information that are required when accessing the resources.

It is interesting to notice that the classification that we propose for secure extensions of Linda is reminiscent of a recent taxonomy of probabilistic extensions of Linda [4] that considers two main approaches: the *schedule-driven* approach initiated in [6] and the *data-driven* approach of [3]. In the former, probability distributions are used to specify the expected schedule of the processes accessing the tuple space. In the latter, weights are associated to tuples in order to quantify their relevance: as greater is the weight of a tuple, as higher is the probability for that tuple to be retrieved. Schedule-driven is similar to entity-driven because it focuses on processes, while data-driven is similar to knowledge-driven because it assumes that the relevant information, i.e. the weights or the capabilities (or the like), are stored within tuples.

The paper in organized as follows. Section 2 presents the Linda calculus, Section 3 an overview of the security threats emerging when the Linda coordination model is exploited in open environments, Section 4 describes two possible approaches for dealing with security issues and Section 5 reports the most interesting actual proposals already available in the literature. Finally Section 6 concludes the paper with some final comments and by reporting related work.

## 2 The Linda coordination model

The coordination primitives that we have in Linda 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. In literature a number of Linda-based languages [14,15,18] support multiple spaces, thus in the model we are going to describe, named `LinCa`, we enrich the syntax of the primitives with an additional paramater indicating the space where the operation is to be performed.

Our modeling of the Linda coordination primitives does not consider the $eval(e)$ operator usually supported by most of the Linda implementations (see for instance [7]). This primitive is used to emit an active tuple; an active tuple is a tuple in which some of the fields must be computed by processes that are spawn in parallel, and only at the end of the computation of all of these processes the new tuple is actually inserted in the TS. In our model, new parallel processes can be emitted simply using the parallel composition as detailed in the following.

More precisely, Linda 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 [7]): 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 we are going to present fields are not typed.

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

Tuples, denoted by $e$, $e'$, ..., are finite and ordered sequences of data fields, whilst templates, denoted by $t$, $t'$, ..., 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 $e$.

Let $Sp$, ranged over by $s$, $s'$, $\ldots$, be the set of tuple space names. In the following, we denote with $TS_s$ the tuple space whose name is $s$. Processes, denoted by $P$, $Q$, $\ldots$, are defined as follows:

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

A process can be a terminated program $\mathbf{0}$, a prefix form $\mu.P$, the parallel composition of two programs, or the replication of a program. The prefix $\mu$ can be one of the following coordination primitives: i) $\mathit{out}\ e@s$, that writes the tuple $e$ in the $TS_s$; ii) $\mathit{rd}\ t(\vec{x})@s$, that given a template $t$ reads a matching tuple $e$ in the $TS_s$ and stores the return value in $\vec{x}$; iii) $\mathit{in}\ t(\vec{x})@s$, that given a template $t$ consumes a matching tuple $e$ in the $TS_s$ and stores the return value in $\vec{x}$. In both the $\mathit{rd}\ t(\vec{x})@s$ and $\mathit{in}\ t(\vec{x})@s$ operations $(\vec{x})$ is a binder for the variables in $\vec{x}$. The parallel composition $P \parallel Q$ of two processes $P$ and $Q$ behaves as two processes running in parallel, whilst the replication operator $!P$ denotes the parallel composition of an unbounded number of copies of $P$.

We use $P[d/x]$ to denote 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 also say that a process is *closed* if it has no free variable. In the following, we consider only processes that are closed and well formed; *Process* denotes the set of such processes.

Let $\mathit{DSpace}$, ranged over by $DS$, $DS'$, $\ldots$, be the set of possible configurations of the TSs, that is $\mathit{DSpace} = \{TS_s \mid s \in Sp, \quad TS_s \in \mathcal{M}_{fin}(\mathit{Tuple})\}$, where $\mathcal{M}_{fin}(S)$ denotes the set of all the possible finite multisets on $S$. The set $\mathit{System} = \{[P, DS] \mid P \in \mathit{Process}, DS \in \mathit{DSpace}\}$, ranged over by $sys$, $sys'$, $\ldots$, denotes the possible configurations of systems. Given $sys = [P, DS]$ we assume that the indexes of the data-spaces in DS are all pairwise distinct.

The semantics we use to describe processes interacting via `LinCa` primitives is defined in terms of a transition system $(\mathit{System}, \longrightarrow)$, where $\rightarrow \subseteq \mathit{System} \times \mathit{System}$. More precisely, $\longrightarrow$ is the minimal relation satisfying the axioms and rules of Table 1 (symmetric rule of (4) is omitted). $(sys, sys') \in \longrightarrow$ (also denoted by $sys \longrightarrow sys'$) means that a system $sys$ can evolve (performing a single action) in the system $sys'$.

$$(1)\quad [out\ e@s.P, DS \cup \{TS_s\}] \longrightarrow [P, DS \cup \{TS_s \oplus e\}]$$

$$(2)\quad \frac{\exists e \in TS_s : e \rhd t}{[in\ t(\vec{x})@s.P, DS \cup \{TS_s\}] \longrightarrow [P[Rv(e)/\vec{x}], DS \cup \{TS_s - e\}]}$$

$$(3)\quad \frac{\exists e \in TS_s : e \rhd t}{[rd\ t(\vec{x})@s.P, DS \cup \{TS_s\}] \longrightarrow [P[Rv(e)/\vec{x}], DS \cup \{TS_s\}]}$$

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

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

Table 1
Semantics of `LinCa`

Axiom (1) describes the output operation that produces a new occurrence of the tuple $e$ in the space $TS_s$ ($TS_s \oplus e$ denotes the multiset obtained by $TS_s$ 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 $TS_s$, it is returned at the process invoking the operation and, in the case of *in*, it is also removed from the space ($TS_s - e$ denotes the removal of an occurrence of $e$ from the multiset $TS_s$). More precisely, the return value is indicated with $Rv(e)$ which represents, in this case, the entire sequence of data fields in $e$. Rule (4) represents a local computation of processes, whilst (5) the replication operator that produces a new instance of the process and copies itself.

## 3 Security threats

Recent distributed applications such as Web services, applications for Mobile Ad Hoc Networks (MANETs), Peer to Peer Applications (P2P) are inherently open to processes, entities, components that are not known at design time. When the Linda coordination model is exploited to program the coordination inside this class of applications (see e.g. [13] for Web services, Lime [14] in the context of MANETs and PeerSpaces [5] for P2P applications) new critical aspects come into play such as the need to deal with a hostile environment which may comprise also untrusted components. In this case an entity may enter the system and, according to the data-driven approach, can access the repository in order to read/remove data, as well as maliciously produce new data. More precisely, each entity, provided it can access the TS, can perform any primitive on the space. In particular, the main problem is that any agent can read/remove any tuple stored into the TS simply by exploiting formal fields, that acts as wildcards. For instance, any process is allowed to perform $in\ \langle null\rangle(\vec{x})@s$ thus removing any tuple (which is nondeterministically selected) with one data field which is available in $TS_s$. The threat in this case is that a malicious process can interfere with the entities which collaborate by using the space TS. As far as the output operation is concerned, it can be the cause of denial of service attacks in which a process maliciously overwhelms the tuple space with an enormous number of new tuples to be stored.

**Example 3.1** ***Group communication -*** *As an example of security threat, we consider the case of group communication realized through a shared data-space. A group $G$ has an associated set of producers and a set of consumers. The producers can execute the operation $sendToGroup(G, d)$ to produce a datum $d \in Data$ which becomes available to the consumers that can consume it by executing the operation $consumeFromGroup(G, x)$. A trivial implementation of these two group communication primitives is based on the shared data-space $TS_{s_G}$ used as a repository for the exchanged messages:*

$$sendToGroup(G, d) := out\ \langle d\rangle@s_G$$

$$consumeFromGroup(G, x) := in\ \langle null\rangle(x)@s_G.$$

*Since all processes can access the tuple space, also a malicious process could perform the same operations by: i) inserting new tuples which are not produced by producers in G, and ii) reading/consuming tuples which were intended for consumers in G. It is rather easy to see that there is no way to avoid such kind of attacks. In the following sections we propose, for each language supporting security features that we describe, a solution guaranteeing a secure group communication.*

## 4 Linda with labels

The extension of Linda with labels we present is devoted to describe uniformly the several proposals for supporting some form of security in Linda-based languages which allow us to prevent the threat explained in the previous section. This uniform presentation represents also the first attempt, to the best of our knowledge, to provide a taxonomy of the solutions adopted in the literature in order to add security to shared data-space coordination languages. In particular, such proposals follow two different approaches: i) the *knowledge driven*, and ii) the *entity driven* approaches.

The idea behind the knowledge driven approach is that resources (i.e. tuple spaces, tuples and single data fields) are decorated with additional reserved data information and the processes can access the resources only in the case they prove to keep the knowledge of such a reserved information. In the case of the entity driven approach, instead, additional information associated to resources list the entities which are allowed to access the resources.

By comparing the two approaches we see that the knowledge driven one abstracts away from the entities involved in the system. Consequently, the entity driven approach is better suited when it is possible to know the whole set of entities involved in the system (e.g., within network where the access is limited to registered users) while in the opposite case, that is open environments, the knowledge driven approach is preferable.

Let $Lab$, ranged over by $l$, $l'$, ..., be the set of labels we use to describe access permissions and $Operation = \{out, rd, in\}$, ranged over by $op$, be the set of possible operations. Now we are interested in describing the idea at the basis of the mechanisms for managing system capabilities. In the following we will define the implementation of such labels used by the most interesting proposals available in the literature.

Labels are used to describe access permissions at the level of entire tuple space, tuple and data field. Formally, tuple spaces are associated with labels describing the permission which rule the access to the space; we denote with $TS_s[l]$ the space, identified by $s$, whose label is $l$. Tuples are now decorated with a label denoting the permissions which rule the access to the entire tuple and are syntactically expressed with $e[l]$. Finally, labels are associated also at the level of single data fields, the sequence of fields in a tuple is now defined as

$$\vec{d} ::= l : d \quad | \quad l : d; \vec{d}$$

where $l : d$ expresses that $l$ rules the access to $d$.

The following subsections are devoted to formalize the Linda extensions supporting knowledge-driven and entity-driven access control mechanisms named `KdLinCa` and `EdLinCa`, respectively.

The knowledge driven approach is based on the idea that the process which is willing to perform an operation on a certain space must provide the necessary knowledge for accessing the space, the tuple and the single data field. To this end, we extend the primitives by adding a label which represents the knowledge allowing to access the space as follows: $rd\ t(\vec{x})@s[l]$, $out\ e@s[l]$ and $in\ t(\vec{x})@s[l]$ where $l$ represents such a knowledge. In the same way, we extend the structure of templates in order to describe the knowledge to access a certain tuple and the single data fields. In this case, templates are extended as the labelled tuples defined above. Let $op \in Operation$ be an operation; in the following, abstracting away from the manner it is implemented, we use $l \rhd_{op} l'$ as the relation espressing that the label $l$ allows to access, by means of operation *op*, a resource labelled with $l'$. In the following we describe how such a relation is defined in existing proposals.

$$(1^K) \quad \frac{l \rhd_{out} l'}{[out\ e@s[l].P, DS \cup \{TS_s[l']\}] \longrightarrow [P, DS \cup \{TS_s[l'] \oplus e\}]}$$

$$(2^K) \quad \frac{l \rhd_{in} l' \qquad \exists e \in TS_s[l'] : e \rhd^K_{in} t}{[in\ t(\vec{x})@s[l].P, DS \cup \{TS_s[l']\}] \longrightarrow [P[Rv(e)/\vec{x}], DS \cup \{TS_s[l'] - e\}]}$$

$$(3^K) \quad \frac{l \rhd_{rd} l' \qquad \exists e \in TS_s[l'] : e \rhd^K_{rd} t}{[rd\ t(\vec{x})@s[l].P, DS \cup \{TS_s[l']\}] \longrightarrow [P[Rv(e)/\vec{x}], DS \cup \{TS_s[l']\}]}$$

Table 2
Semantics of knowledge-driven mechanisms

The matching rule of labelled tuples and templates, whose definition follows, is a conservative extension of the standard one.

**Definition 4.1** ***Knowledge-driven matching rule -*** *Let* $e = \langle l_1 : d_1; l_2 : d_2; \ldots; l_n : d_n\rangle[l]$ *be a tuple,* $t = \langle lt_1 : dt_1; lt_2 : dt_2; \ldots; lt_m : dt_m\rangle[lt]$ *be a template and* $op \in \{rd, in\}$ *be an operation; we say that* $e$ *matches* $t$ *when the operation* $op$ *is performed (denoted by* $e \rhd^K_{op} t$*) if the conditions in Definition 2.1 and the following hold:*

*(1)* $lt \rhd_{op} l$.
*(2)* $lt_i \rhd_{op} l_i$, $1 \leq i \leq n$.

The first condition verifies that the template has provided the right knowledge to access the tuple and the second one that the same holds for each labelled data fields.

The semantics of `KdLinCa` is defined as in Table 1 where the rules (1), (2) and (3) are replaced by the corresponding ones in Table 2. Informally, the operations can be performed only if they provide the right knowledge to access the tuple space and, in the case of data-retrieval, the specific matching tuple. More precisely, $out\ e@s[l]$ is performed provided that the label $l$ satisfies the policy on output of $l'$ which is the space where it intends to insert the tuple. Besides testing the access permissions at the level of tuple space, the data-retrieval operations $rd$ and $in$ verify access permissions at the level of tuples and single fields according with the definition of $\triangleright_{op}^{K}$.

Finally, a remark about the return value $Rv(e)$ of a data-retrieval operation accessing the entry $e$. In this case $Rv(e)$ could contain not only data fields but also some labels available in $e$, thus obtaining dynamic privileges acquisition. In the following we will see how this is actually permitted in the languages based on knowledge-driven mechanisms.

## *4.2 Entity-driven mechanisms*

The entity-driven mechanism is based on the idea that the access permissions (in this case by means of labels) somehow list the entities which are allowed to perform specific operations or to access certain tuples. To this end, the process description is enriched with an information indicating the identity of the entity executing the process. Formally let $Id$, ranged over by $id$, $id'$, $\ldots$, be the set of entity identifiers, the processes we consider are defined now by $P^E, Q^E, \ldots ::= id : P \mid P^E \parallel P^E$ where $P$ is a standard process as defined in Section 2. $id : P$ means that the entity identified by $id$ is willing to execute $P$ and $P^E \parallel Q^E$ is used to express that different entities can execute processes. In a term $P^E$ we assume that the identities are pairwise distinct.

Let $op \in Operation$ be an operation, in the following we use the relation $id \triangleright_{op} l$ to denote whether the entity identifier $id$ is allowed to access, during the operation $op$, the resource labelled with $l$ (obviously $op = out$ has a meaning only in the case $l$ is associated with a tuple space).

In the case of the entity-driven mechanism the matching rule, whose definition follows, takes into account the identity of the entity in order to verify it possesses the right permissions.

**Definition 4.2** ***Entity-driven matching rule*** **-** *Let* $e = \langle l_1 : d_1; l_2 : d_2; \ldots ; l_n : d_n\rangle[l]$ *be a tuple,* $t = \langle dt_1; dt_2; \ldots ; dt_m\rangle$ *be a template,* $op \in \{rd, in\}$

*be an operation and $id \in Id$ be an entity identifier; we say that $e$ matches $t$ when $op$ is performed by $id$ (denoted by $e \triangleright^{E}_{op,id} t$) if the conditions in Definition 2.1 and the following hold:*

*(1) $id \triangleright_{op} l$.*
*(2) $id \triangleright_{op} l_i$, $1 \leq i \leq n$.*

*The first condition verifies that the entity identified by $id$ has the right permission to access the tuple and the second one that the same holds for all data fields.*

In order to support dynamic privileges acquisition we assume that privileges, as in the knowledge driven approach, are contained in the tuple space thus an entity can update its access permission by performing data-retrieval operations. We use the function $upd(id, e, DS)$ to represent the new state of the data spaces after the access permission update for entity $id$ when it reads/consumes the tuple $e$.

The semantics of the calculus based on an entity-driven mechanism is reported in Table 3. The five rules are simple adaptations of the corresponding rules in Table 1.

## 5 Linda-like languages for untrusted environments

The most interesting proposals of coordination languages supporting security in untrusted environments will be listed and described in this section as instantiations of `KdLinCa` and `EdLinCa`. More precisely, we will describe the labels content and their granularity and, when necessary, the return value of data-retrieval operations.

### *5.1 KLAIM*

KLAIM [15] (Kernel Language for Agent Interaction and Mobility) is designed to program distributed applications that can interact using multiple tuple spaces and mobile code. The KLAIM operations basically are the same of Linda where in addition they allow the reference to a specific tuple space.

The basic element of the system is the location that is composed by a tuple space and a process. Locations are described by a term $l : [P, TS]$ where $l$ is a location while $P$ and $TS$ are, respectively, the process running at and the space contained in $l$. Systems consist of the parallel composition of locations.

$$(1^E)\quad \frac{id \rhd_{out}\, l}{[id : out\ e@s.P, DS \cup \{TS_s[l]\}] \longrightarrow [id : P, DS \cup \{TS_s[l] \oplus e\}]}$$

$$(2^E)\quad \frac{id \rhd_{in}\, l \qquad \exists e \in TS_s[l] : e \rhd^{E}_{in,id}\, t}{\begin{array}{l}[id : in\ t(\vec{x})@s.P, DS \cup \{TS_s[l]\}] \longrightarrow \\ \qquad [id : P[Rv(e)/\vec{x}], upd(id, e, DS \cup \{TS_s[l] - e\})]\end{array}}$$

$$(3^E)\quad \frac{id \rhd_{rd}\, l \qquad \exists e \in TS_s[l] : e \rhd^{E}_{rd,id}\, t}{\begin{array}{l}[id : rd\ t(\vec{x})@s.P, DS \cup \{TS_s[l]\}] \longrightarrow \\ \qquad [id : P[Rv(e)/\vec{x}], upd(id, e, DS \cup \{TS_s[l]\})]\end{array}}$$

$$(4^E)\quad \frac{[P_1^E, DS] \longrightarrow [P_2^E, DS']}{[P_1^E \parallel P^E, DS] \longrightarrow [P_2^E \parallel P^E, DS']}$$

$$(5^E)\quad \frac{[id : P, DS] \longrightarrow [id : P', DS']}{[id : !P, DS] \longrightarrow [id : P' \parallel !P, DS']}$$

Table 3
Semantics of entity-driven mechanisms

The entity driven approach is used to control the accesses: locations in this case correspond to entity identifiers of `EdLinCa` thus tuple spaces as well as entities are identified by locations.

The access permissions KLAIM supports have a granularity at the level of tuple spaces. Moreover it does not allow dynamic privileges acquisition: capabilities are completely defined at design time. Access permissions, expressed in the `EdLinCa` abstract model by means of labels, are in this case sets associated to tuple spaces and describe, for each location, the allowed operations. Moreover, if a process is allowed to perform *in* operations it inherits also the permission to access the space with *rd* ones.

Formally let *Loc*, ranged over by *loc*, be the set of locations. The set of entity identifiers and tuple space names are respectively defined as $Id := Loc$ and $Sp := Loc$, while the set of labels is defined as the set $Lab := \mathcal{F}_p(Loc, \mathcal{P}(Operations))$ of the partial functions from locations to sets of operations. Given a label $l$ that maps the location *loc* to the set of operations

$ao$, denote with $(loc \mapsto ao) \in l$, this means that the processes running on the location $l$ are allowed to perform on the space with location $loc$ only the operations in $ao$. For instance, $TS_{loc_1}[\{(loc_2 \mapsto \{rd, out\}), (loc_3 \mapsto \{in, out\})\}]$ means that processes running at location $loc_2$ can access $TS_{loc_1}$ only by using $rd$ and $out$ operations, while processes running at $loc_3$ only by using $in$ and $out$ ones. We assume that there exists a special label in $Lab$, denoted by $\perp$, that does not filter accesses to resources. Since KLAIM does not support labels at the level of single tuple and data field, we assume that tuples and data fields are associated to $\perp$.

Now we are ready to define the relation $\triangleright_{op}$, which verifies access control capabilities, and the function $upd$ used to update access permissions.

**Definition 5.1** ($\triangleright_{op}, upd$) **-** *Let $loc \in Id$, $op \in Operation$ and $l \in Lab$ be, respectively, an entity identifier, an operation and a label, $loc \triangleright_{op} l$ holds if one of the following conditions hold:*

- $l = \perp$.
- $\exists (loc' \mapsto ao) \in l \text{ s.t. } loc = loc' \text{ and } (op \in ao \text{ or } (op = rd \text{ and } in \in ao))$

*As there is no privileges acquisition, the funtion upd does not modify the data-space, that is:*
*$upd(id, e, DS) = DS$ for all $id \in Id$, $e \in Tuple$ and $DS \in DSpace$.*

The semantics of KLAIM is defined by the semantics of `EdLinCa` in Table 3 where $Id$, $Lab$, $upd$ and $\triangleright_{op}$ are the ones defined in the current section. The KLAIM model is also equipped with a type system which permits to verify whether a system behaves according with the capabilities. Types are used to express location capabilities, that is the capabilities associated to the processes running in that location. By exploiting these types, a static type-checking has been introduced for verifying whether the access permissions associated to the location where the processes run allow for the execution of the primitives declared by the processes.

**Example 5.2** ***Secure group communication*** **-** *We propose a secure solution to the problem of group communication described in Example 3.1.*

*The idea is that we use a certain location which contains the tuple space used to exchange data whose label allows: i) out operations for producers, and ii) in operations for consumers.*

*Let $G$ be the name of the group, $s_G$ be the location which identifies the tuple space used by group $G$, $P$ and $C$ be the sets of locations (entity identifiers) of producers and consumers, respectively, and $l := \{(lp \mapsto \{out\}) \mid lp \in P\} \cup \{(lc \mapsto \{in\}) \mid lc \in C\}$ be the label associated to $TS_{s_G}$.*

*Let $d \in Data$ be a datum, a producer can insert the datum $d$ by using the function $sendToGroup(G, d)$ defined as follows (we omit to denote the $\perp$ labels on tuples and data fields):*

$$sendToGroup(G, d) := out\ \langle d \rangle @ s_G$$

*while a consumer can acquire and store data in $x$ by using the function $consumeFromGroup(G, x)$ defined as follows:*

$$consumeFromGroup(G, x) := in\ \langle null \rangle (x) @ s_G.$$

*In this case the label associated to $TS_{s_G}$, $l$, guarantees that only producers can insert new tuples into the space and only consumers can consume that tuples. Indeed, a malicious process whose location (i.e. an entity) $l \notin P \cup C$ cannot access the space.*

## *5.2 Secure Lime*

This proposal [12] introduces security mechanisms in Lime [14] (Linda in a Mobile Environment) at the implementation level. It is a secure implementation of Lime that provides a password-based access control mechanism at the level of tuples and tuple spaces. More precisely, the password-based system on tuple space and tuples permits the access only to the authorized processes, that is those that know the password. In particular, password-based access permissions on tuples can be associated to the *rd* and to the *in* operations while, at the level of tuple space, a single password can be used to limit the access to the space. In the case a process is allowed to remove a certain tuple (i.e. it knows the password associated to the removal operations), it has also the permission of reading that tuple.

We model such a solution by means of `KdLinCa`. Indeed, passwords are not associated to entities thus any entity which keeps a password, say $pw$, can access resources protected with $pw$. Let $Passwd$, ranged over by $pw$, be the set of passwords. The set of labels is, in this case, composed of two sets: one for labeling tuples (with *rd* and *in* modalities) and the second one for labeling tuple spaces. Formally, $Lab := LabTuple \cup LabTS$ where $LabTuple := \{(pw, pw') \mid pw, pw' \in Passwd\}$ and $LabTS := Passwd$. $(pw, pw') \in LabTuple$ means that the tuple is protected with $pw$ and $pw'$ for *rd* and *in* accesses, respectively. $pw \in LabTS$ means that the tuple space is protected with password $pw$. Obviously processes when performing an operation will use a single password which, in the case of data-retrieval, will be associated to the operation it is executing. We assume that there exists a special label, denoted by $\perp$, whose meaning is that the resource is not protected; since the language does not support labels at the level of data fields, $\perp$ will be associated to data fields. The definition of the $\triangleright_{op}$ operator and of the return value follows.

**Definition 5.3** $(\triangleright_{op}, Rv)$ **-** *Let $pw, l \in Lab$ be two labels, $op \in Op$ be an operation, we say that $pw \triangleright_{op} l$ if one of the following conditions holds:*

- $l = (pw_1, pw_2) \in LabTuple$ *and* $(op = rd)$ *and* $((pw = pw_1) or (pw = pw_2))$.
- $l = (pw_1, pw_2) \in LabTuple$ *and* $(op = in)$ *and* $(pw = pw_2)$.
- $l = pw' \in LabTS$ *and* $pw = pw'$.
- $l = \perp$.

*Observe that, as in KLAIM, the first item ensures that the in capability implies also the rd capability. Let $e[l] \in Tuple$, the return value is defined as $Rv(e[l]) := e$ (i.e it does not return labels used to control the accesses).*

The semantics of the secure version of Lime is defined by the semantics of `KdLinCa` in Table 2 where $Lab$, $\triangleright_{op}$ and $Rv$ are the ones defined in the current section.

**Example 5.4** ***Secure group communication* -** *A solution to the problem of secure group communication described in Example 3.1 is presented.*

*The idea is that we use a certain tuple space to exchange data whose access is protected by a password which is known only to the entities involved in the group.*

*Let $G$ be the name of the group, $TS_{s_G}$ be the tuple space used by group $G$ and $pw_G$ be the password used to protect the access to $TS_{s_G}$. Let $d \in Data$ be a datum, a producer can insert the datum $d$ by using the function $sendToGroup(G, d)$ defined as follows (we omit $\perp$ labels on tuples and data fields):*

$$sendToGroup(G, d) := out\ \langle d \rangle @ s_G[pw_G]$$

*while a consumer can acquire and store data in $x$ by using the function $consumeFromGroup(G, x)$ defined as follows:*

$$consumeFromGroup(G, x) := in\ \langle null \rangle (x) @ s_G[pw_G].$$

*In this case, since only the entities in $G$ hold $pw_G$, only those ones can access the space $TS_{s_G}[pw_G]$ and no malicious entities can access the space. We would like to point out that, differently from Example 5.2, in this language consumers can also insert new tuples as well as producers can also consume tuples. This directly follows by the fact that the password mechanism is symmetric: for instance when a producer protects a tuple it defines the password thus it can also consume that tuple.*

## *5.3 SecOS*

SecOS [19] follows the knowledge-driven approach thus we exploit `KdLinCa` to describe the language and its semantics. Labels, in this case named keys, are available in two kinds: i) symmetric, and ii) asymmetric keys. The former one essentially defines the match between keys as the result of the comparison while in the latter case the idea that there exists a relation which permits to verify whether a certain key is the co-key of another one. Such an approach is the well known one used in the field of cryprography where the public-key mechanism guarantees that given a key it is not possible to guess its co-key.

The access keys can be associated to either the tuples (object locks) or single fields contained in tuples (field locks) since the model works on a single tuple space whose access is not restricted to anyone (consequently output operations are not controlled by means of labels). Data in the tuples can contain access key values, thus the shared space can be used to perform the distribution of access permissions. The mechanism used to control the accesses is the knowledge-driven one: keys are used to express access permissions and, in the case of symmetric keys, the same key is used to protect and access the information, while in the case of asymmetric keys it uses a pair of keys, one to protect and another one to access.

Let $Key$, ranged over by k, be the set of symmetric keys, and $AKey$, ranged over by $k_a$, be the set of asymmetric keys. We define $\overline{\cdot} : AKey \rightarrow AKey$ as the function that, given an asymmetric key, returns the corresponding co-key and such that $\overline{\overline{k}} = k$. For the sake of simplicity, we extend such a definition also for symmetric keys where the function $\overline{\cdot}$ is the identity one. Since SecOS uses only one unprotected tuple space, we assume that there exists a special value, denoted with $\perp$, which we use to label the tuple spaces as well as that all operations use $\perp$ as label for accessing the space.

In the case of SecOS the set of labels of `KdLinCa` is defined as $Lab := Key \cup AKey$. The definition of the relation $\triangleright_{op}$ between labels and of the return value follows.

**Definition 5.5** ($\triangleright_{op}, Rv$) **-** *Let $k, k' \in Lab$ be two access keys, $op \in Op$ be an operation, we say that $k \triangleright_{op} k'$ if $k' = \overline{k}$ (for the sake of simplicity we are assuming that $\overline{\perp} = \perp$).*

*The return value of data-retrieval operations is defined as follows: $Rv(e) := e$ for all tuples $e \in Tuple$.*

It is worth noting that *op* does not affect the definition of $\triangleright_{op}$. Indeed, the same access policy is associated to *rd* and *in* operations.

The semantics of SecOS is defined by the semantics of `KdLinCa` in Table 2 where $Lab$, $\triangleright_{op}$ and $Rv$ are the ones defined in the current section. It is worth noting that, since the return value contains both labels and data fields, SecOS allows for dynamic privileges acquisition and that, since the space is not protected by labels, output operations cannot be controlled thus losing the possibility to avoid denial of service attacks.

**Example 5.6** ***Secure group communication*** **-** *Here we use SecOS to present a solution to the problem of secure group communication described in Example 3.1.*

*The idea is to exploit an asymmetric key, that we use to protect tuples, which is known only to the producers while the consumers are the only ones which hold the corresponding co-key. In this way such tuples can be inserted and consumed only by producers and consumers, respectively.*

*Let* $G$ *be the name of the group,* $kp_G \in AKey$ *be the asymmetric key used to protect the access to* $TS_{s_G}$ *which is known only to the producers in* $G$ *while its co-key,* $\overline{kp_G} = kc_G$*, is known only to the consumers.*

*Let* $d \in Data$ *be a datum, a producer can insert the datum* $d$ *by using the function* $sendToGroup(G, d)$ *defined as follows (we omit* $\perp$ *labels on tuple spaces and data fields and the reference to the single space available):*

$$sendToGroup(G, d) := out\ \langle d \rangle [kp_G]$$

*while a consumer can acquire and store data in* $x$ *by using the function* $consumeFromGroup(G, x)$ *defined as follows:*

$$consumeFromGroup(G, x) := in\ \langle null \rangle\ (x)[kc_G].$$

*Besides guaranteeing that no malicious processes can insert or consume tuples used by the group, it is also guaranteed that receivers cannot insert new tuples as well as producers cannot consume available tuples.*

## *5.4 SecSpaces*

SecSpaces [10] follows the same idea introduced by SecOS. On one hand it refines the access permissions of SecOS by allowing to express different labels for $rd$ and $in$ operations (i.e. different access policies), on the other hand labels can be associated only at the level of tuple and not single data fields. Finally, the return value of a matching tuple does not contain labels, thus the only way to perform dynamic privileges acquisition is to insert labels within a data field.

Labels in SecSpaces are defined as $Lab := LabTuple \cup LabOp$ where $LabTuple := \{((k, ak)_{rd}, (k', ak')_{in}) \mid k, k' \in Key, ak, ak' \in AKey\}$ and $LabOp := \{(k, ak) \mid k \in Key, ak \in AKey\}$ ($Key$ and $AKey$ are the ones used to formalize SecOS which are defined in the previous section). Labels in $LabTuple$ are used to protect tuples: $((k, ak)_{rd}, (k', ak')_{in}) \in LabTuple$ means that access permissions for $rd$ and $in$ are expressed by means of the pair $(k, ak)$ and $(k', ak')$, respectively. Labels in $LabOp$ are used when performing an operation, they are not associated to a specific operation because it is dynamically determined during the execution. Since also SecSpaces models a single unprotected space and does not support data field labels, we assume that there exists a special value, denoted by $\perp$, which is associated to the space and to the data fields of tuples and templates.

**Definition 5.7** ($\rhd_{op}, Rv$) **-** *Let $(k, ak) \in LabOp$, $((k_1, ak_1)_{rd}, (k_2, ak_2)_{in}) \in LabTuple$ be two labels, $op \in Op$ be an operation, we say that $(k, ak) \rhd_{op} ((k_1, ak_1)_{rd}, (k_2, ak_2)_{in})$ if one of the following conditions holds:*

- $(op = rd)$ *and* $(k = k_1)$ *and* $(ak = \overline{ak_1})$.
- $(op = in)$ *and* $(k = k_2)$ *and* $(ak = \overline{ak_2})$.

*In the special case the label used to protect is $\perp$ we have that $l \rhd_{op} \perp$ for all $l \in Lab$ used to access. In the remaining cases the relation does not hold.*

*Let $e[l] \in Tuple$, the return value is defined as $Rv(e[l]) := e$ (i.e it does not return labels used to control the accesses).*

The semantics of SecSpaces is defined by the one of `KdLinCa` in Table 2 where $Lab$, $\rhd_{op}$ and $Rv$ are the ones defined in the current section. It is worth noting that SecSpaces refines the access control policies of SecOS from the point of view of data-retrieval operations while the space is still not protected by labels, thus output operations cannot be controlled. The model is also equipped with a testing equivalence [2] which permits to formalize, and verify, some of the main security properties (e.g., secrecy, authentication).

**Example 5.8** ***Secure group communication -*** *A solution based on SecSpaces supporting the secure group communication described in Example 3.1 is here presented.*

*The idea to solve the problem is, in this case, similar to the one proposed by using SecOS in Example 5.6. An asymmetric key, which is known only to the producers, is used to protect tuples while the consumers are the only ones which hold the corresponding co-key.*

*Let $kp_G, kc_G \in AKey$ and $G$ defined as in Example 5.6, $d \in Data$ be a datum, $k \in Key$ ba a symmetric key, a producer can insert the datum $d$ by using the function $sendToGroup(G, d)$ defined as follows (we omit $\perp$ labels on tuple* *spaces and data fields and the reference to the single space available):*

$$sendToGroup(G,d) := out\ \langle d\rangle[(k,kp_G)_{rd},(k,kp_G)_{in}]$$

*while a consumer can acquire and store data in x by using the function consumeFromGroup(G, x) defined as follows:*

$$consumeFromGroup(G,x) := in\ \langle null\rangle\ (x)[(k,kc_G)].$$

*Besides guaranteeing the same properties of the solution based on SecOS, it is worth noting that here it is also possible to refine access permissions on data-retrieval by distinguishing between the processes allowed to consume and read the tuples. Such an extension can be obtained by using different asymmetric keys for rd and in modalities.*

## 6 Conclusion and related work

In this paper we have outlined the main security threats occurring when the data-space coordination model is used in untrusted environments and presented a survey of the most interesting Linda-like languages supporting some form of security policies. To this end we have exploited two meta-models, `KdLinCa` and `EdLinCa`, that describe the idea and the semantics of the access control mechanisms based on the knowledge-driven and the entity-driven approaches, respectively.

We have also proposed a secure solution to the problem of secure group communication in all languages we have discussed. Obviously, as emerge by their semantics and in some cases by the discussed example, there exist significant differences among such languages. For instance, by considering the proposals based on the knowledge-driven approach, the secure version of Lime is the only one which permits to control output operations, and SecSpaces is the only one which permits to distinguish between the processes that can consume and the processes that can read a certain tuple.

Finally, we list some related works which follow different approaches w.r.t. the ones used by `EdLinCa` and `KdLinCa`. Two other proposals, muKLAIM and Klava, are extensions of the KLAIM model. Differently from KLAIM, muKLAIM [9] allows dynamic privileges acquisition. In particular, this can be done by performing in/rd operations with special template fields. While in KLAIM security policies do not change dynamically and then static type checking allows to test if running processes have the necessary access permissions to perform their operations, muKLAIM proposes a type system that evolves according to dynamic variations of security policies. The approach exploits a combination of static and dynamic type checking. In order to improve the system performance, the type inference rules perform a static type checking to those operations for which it is possible to statically test the presence of the necessary access permission.

Klava [1] is an extension of Klaim which exploits cryptography to allow only to authorized users access to the specific information stored inside the tuples, but nothing is done to restrict the access to the tuples stored in the TS. In particular it introduces encrypted messages into the fields of the tuples and the matching rule allows the evaluation of messages encrypted into fields; the encryption of messages ensures that they can be read only by the allowed clients.

Finally, a different approach is proposed in [16], where a general model for coordination middleware that exploits process handlers to control the behaviour of processes has been presented. More precisely, a context is associated to each process and it defines which operations are allowed to the process. In order to express allowed actions, a language derived from CCS is used to describe which operations the processes can perform.

## References

[1] L. Bettini and R. De Nicola. A Java Middleware for Guaranteeing Privacy of Distributed Tuple Spaces. In E. Astesiano N. Guelfi and G. Reggio, editors, *Proc. of FIDJI'02, Int. Workshop on scientific engineering of distributed Java applications*, volume 2604 of *LNCS*, pages 175–184. Springer-Verlag, 2003.

[2] M. Bravetti, R. Gorrieri, and R. Lucchi. A formal approach for checking security properties in SecSpaces. In *1st International Workshop on on Security Issues in Coordination Models, Languages and Systems*, volume 85.3 of *ENTCS*, 2003.

[3] M. Bravetti, R. Gorrieri, R. Lucchi, G. Zavattaro. Probabilistic and Prioritized Data Retrieval in the Linda Coordination Model. In *Proc. of 7th International Conference on Coordination Models and Languages (Coordination 04)*, volume 2949 of *LNCS*, pages 55–70. Springer Verlag, 2004.

[4] M. Bravetti, R. Gorrieri, R. Lucchi, G. Zavattaro. *"Quantitative Information in the Tuple Space Coordination Model"*, *Theoretical Computer Science*, 346:1, pages 28-57, Elsevier, 2005.

[5] N. Busi, C. Manfredini, A. Montresor, and G. Zavattaro. PeerSpaces: Data-driven Coordination in Peer-to-Peer Networks. In *Proc. of ACM Symposium on Applied Computing (SAC'03)*, pages 380–386. ACM Press, 2003.

[6] 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.

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

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

[9] Daniele Gorla and Rosario Pugliese. Resource Access and Mobility Control with Dynamic Privileges Acquisition. In *, Languages and Programming, 30th International Colloquium, ICALP 2003*, volume 2719 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2003.

[10] Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. Supporting Secure Coordination in `SecSpaces`. *Fundamenta Informaticae.* To appear.

[11] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[12] Radu Handorean and Gruia-Catalin Roman. Secure Sharing of Tuple Spaces in Ad Hoc Settings. In *1st International Workshop on on Security Issues in Coordination Models, Languages and Systems*, volume 85.3 of *ENTCS*, 2003.

[13] Roberto Lucchi and Gianluigi Zavattaro. WSSecSpaces: a Secure Data-Driven Coordination Service for Web Services Applications. In *Proc. of ACM Symposium on Applied Computing (SAC'04)*, pages 487–491. ACM Press, 2004.

[14] A. Murphy, G. Picco, and G.-C. Roman. A middleware for physical and logical mobility. In *21st International Conference on Distributed Computing Systems*, pages 524–533, 2001.

[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] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Formal Specification and Enactment of Security Policies through Agent Coordination Contexts. In *1st International Workshop on on Security Issues in Coordination Models, Languages and Systems*, volume 85.3 of *ENTCS*, 2003.

[17] B. Schneier. *Applied Cryptography.* Wiley, 1996.

[18] Sun Microsystems, Inc. *JavaSpaces$^{TM}$ Service Specification*, 2002. http://www.sun.com/jini/specs/.

[19] Jan Vitek, Ciarán Bryce, and Manuel Oriol. Coordinating Processes with Secure Spaces. *Science of Computer Programming*, 46:163–193, 2003.

[20] P. Wyckoff, S.W. McLaughry, and D.A. Ford. TSpaces. *IBM System Journal*, August 1998.