Agents Secure Interaction in Data driven Languages

Mahdi Zargayouna[1]
[1] INRETS Institute,
Gretia Laboratory,
2, Rue de la Butte Verte
93166 Noisy Le Grand, France
Email: zargayouna@inrets.fr

Flavien Balbo[1,2]
[2] University of Paris-Dauphine,
Lamsade-CNRS Laboratory,
Place du Maréchal de Lattre de Tassigny
75775 Paris, France
Email: balbo@lamsade.dauphine.fr

Serge Haddad[3]
[3] École Normale Supérieure de Cachan,
LSV-CNRS Laboratory,
61, Avenue du Président Wilson
94235 Cachan, France
Email: haddad@lsv.ens-cachan.fr

***Abstract*—This paper discusses the security issues in data driven coordination languages. These languages rely on a data space shared by the agents and used to coordinate their activities. We extend these languages with a main distinguishing feature, which is the possibility to define fine-grained security conditions, associated with every datum in the shared space. Two main ideas makes it possible: the consideration of an abstraction of agents' states in the form of data at language level and the introduction of a richer interaction mechanism than state-of-the-art templates. This novel security mechanism allows both agents and system designers to prohibit undesirable interactions.**

I. Introduction

When designing logically distributed applications and open Multi-Agent Systems (MAS), developing applications without knowing either the overall structure of the system or the agents that will be functioning in it is a challenge. Data driven coordination languages, with the pioneer language Linda [1] and its extensions, provide a great deal of flexibility and are a promising approach to meet this challenge. These languages are based on the notion of a shared data repository composed of data used by the agents to interact and to synchronize their activities. Agents communicate by exchanging tuples via an abstraction of an associative shared memory called the *tuplespace*. A tuplespace is a multiset of tuples (tuples duplication is allowed) and is accessed associatively (by content) rather than by address, by specifying a template. Every tuple is a sequence of one or more typed values and every template is a sequence of one or more typed values or formal fields. Every tuple field matches with the corresponding template field if they have the same value or are of the same type.

Relying on a shared space for agent interaction naturally handles open systems design [2]. The advantage is that they provide the possibility for new agents to join the system and, since all the agents have a common interlocutor (the shared space), they don't have to manage an up-to-date address book of the other agents of the system. Nevertheless, the openness management implies a secure relationship between the agents and the shared memory. The data driven coordination model has to deal with the following security threats [3]: 1) threat on authenticity; 2) threat on confidentiality; 3) threat on availability. A threat on authenticity occurs when an agent acts instead of another agent. The data driven coordination model is designed to promote anonymous interaction, but if the tuples contain values related to the agents that insert, read or consume it then the authenticity of these data has to be validated. For example, in [4] the authors present a messaging service where the interaction between the agents is mediated by a broker that is grounded on a tuplespace. In this application, the authenticity of the agents related to a message exchange has to be guaranteed. The confidentiality threats are related to the interception by an agent of another agent's confidential information or message. Following the data driven approach, any agent can read/remove any tuple stored in the tuplespace simply by exploiting formal fields (variables) [1], which act as wildcards [5]. Therefore, a template having two wildcard fields can be used to read or remove any tuple containing two data fields [4]. The threats on availability concern the consequence of the deletion and insertion operations on the behavior of the tuplespace. The deletion of information is a consequence of the lack of confidentiality and implies that it is not possible to guarantee the correct behavior of the system. The threat related to the addition of information concerns malicious agents that can insert an unbounded number of tuples; in such a way, since the manager of the space has to handle any tuple's insertion operation, a process can generate a denial of service attack [4].

In data driven coordination languages, security is generally enforced by using multiple (logical) spaces or by stating "Interaction Laws". With multiple spaces (e.g. Klaim [6], RBAC and TucSon [7], SecSpaces [8] and SecOS [9]), two agents that wish to exchange confidential data use a space that is known only by the two of them. However, security with multiple spaces is defined in a coarse-grained way, since accessing one space gives the possibility to access all of its data, and being excluded from one space means not having access to any of its data. The laws in LGI [10] allow data to be secured by specifying conditions on the states of the agents and the content of the data. Tuple-space reactions are associated with agents' actions so as to always result in a coherent configuration. However, agents cannot manage the security of the data they add, and only the designer can specify security conditions.

We would like to equip data driven coordination languages with a security mechanism that allows for the protection of the exchanged data in a fine-grained way. We want to let agents specify, when they add a datum to the data space, the conditions under which it can be read or taken by others.

We also would like the designer of the system to specify the conditions under which an agent can or cannot add a certain datum to the space, following the application logic. To do so, we perform several modifications of the shared space model, and propose a new language, called LACIOS (Language for Agent Contextual Interaction in Open Systems), which is the linguistic embodiment of the modified model.

In [5], the authors classify secure data driven languages into entity driven and knowledge driven languages. The idea behind the knowledge oriented approach is that tuple spaces, tuples or single data fields are decorated with additional information and an agent can access the resources only in the case they prove their knowledge of these information. In the case of the entity oriented approach, additional information associated to resources list the entities which are allowed to access the resources. Our proposal can be classified as an entity oriented approach. However, instead of listing the agents that can access a datum, in LACIOS these agents are described *symbolically*, i.e. their properties are defined without pointing them namely.

The remainder of this paper is structured as follows. We give an overview of LACIOS in section II. Section III gives the basic syntax of our language; Section IV addresses the security issue, and section V provides the complete specification of agents' behaviors. Section VI presents the programming language JAVA-LACIOS. The proposal is discussed with respect to the state of the art in section VII before concluding with further lines of research.

II. OVERVIEW OF LACIOS

A MAS written in LACIOS is defined by a dynamic set of *agents* interacting with an *environment*, which is composed of a dynamic set of *objects*. To illustrate the syntax of LACIOS, an example application is used throughout the paper. In this example, human travelers are in a train station in which schedules, booking, payment services and information sources coexist. Two agent types are considered in here: Traveler agents represent travelers wishing to make a journey and Train agents represent trains, and generate information concerning future departures, arrivals, delays, etc. All these agents interact by exchanging data via a shared space in the same way as for all data driven coordination models.

A MAS written in LACIOS is an open system in two ways. As for every data driven language, agents in LACIOS can join and leave the system freely. In addition, external - non modeled - systems and users can interact with the MAS. As we will define it later, users (e.g. travelers) interact with the MAS by instantiating the values of certain variables in the code of the agents that represent them. External systems (e.g. trains) can interact with the MAS by instantiating variables with values as well. They can also execute agents that interact with the MAS Environment directly. The figure 1 illustrates the MAS architecture. The modeled MAS executes on a host, where (local) agents add, read and take objects to/from the MAS environment. Every agent is either independent (like agent 1), or representing a non-modeled system/user in the MAS.

The agents that are defined in a LACIOS program are usually the *local* agents. The users, external agents and external systems that are represented by an agent in the MAS are not modeled, only their actions are observed in the MAS, through the nondeterministic behavior of the local agent. An agent in LACIOS is then an entity, that has a state, a local memory and a nondeterministic behavior. As we will define it later, the whole behavior of the agent is not defined in LACIOS. An agent can have a complex behavior, by using additional operators, besides the standard operations defined in LACIOS.

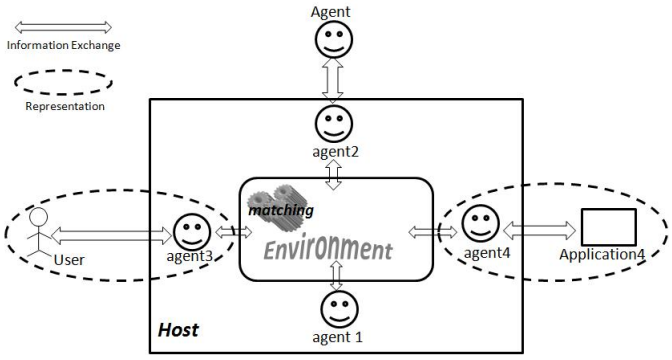

Fig. 1. LACIOS Architecture

Since agents in LACIOS don't interact directly, but via the environment, our definition of an agent is close to the general definition given by [11]:

Definition 1 (Agent): An agent is a computer system capable of autonomous action in some environment in order to meet its design objectives.

From the security point of view, LACIOS has two objectives: 1) to support a global control by the environment of the insertion by the agents of objects in order to ensure that the new objects are not fraudulent (authenticity, availability), 2) to support a local control by the agents that can specify *who* can access the object that they add to the environment in order to ensure their privacy (authenticity, confidentiality and availability). To do so, agents have to have a state defining who they are. This is the first modification we perform to the original model: the consideration of an abstraction of agents' states in the form of data at language level. These states are defined as a set of *property*←*value* pairs (e.g. $\{identifier \leftarrow 10, position \leftarrow node_1\}$). Agents' states in LACIOS are data representing the state of the agents that are accessed by the environment only for matching and security purposes (they are not directly accessible by the other agents).

III. BASIC SYNTAX AND INFORMAL SEMANTICS FOR LACIOS

LACIOS is a data driven language for the design and implementation of open and secure MAS. For the specification of agent behavior, four primitives inspired by Linda and a set of operators borrowed from Milner's CCS [12] have been used. An MAS written in LACIOS is defined by a dynamic set of *agents* interacting with an *environment* - denoted Ω_{ENV},

which is composed of a dynamic set of *objects*. Figure 2 illustrates the general principle of LACIOS. Agents are defined by a behavior (a process), a state (data) and a local memory in which they store the objects they perceive or retrieve from the environment. Agents can *perceive* (read only) and/or *retrieve* (read and take) objects from the environment. First the four primitives of LACIOS will be presented; their parameters will be defined along with the details of the language.

$$\mu ::= spawn \mid add \mid update \mid look$$

The primitive $spawn$ launches a new agent and provides it with an initial state and a behavior. An add action adds an object to the environment. The $update$ primitive changes locally the old values of the agent's state to the new ones.

Unlike traditional retrieval primitives, the $look$ primitive enables agents for both the perception and retrieval of objects as will be described below. The primitive looks for objects in the environment that satisfy the agents' conditions expressed as parameters. Agents can use their own states in the parameter expression of a $look$, which are accessible by the environment only, when the parameter expression is evaluated. Note that, the state of an agent cannot be accessed directly by the other agents (through a $look$ expression). In order to be observable to the others, an agent has to add an object representing itself to the environment autonomously (as in Fig. 2, where the agent decides not to publish a part of its state). Having data representing agents in the environment allows the agents of the system to discover each other by simply interrogating the environment *à la Linda*.

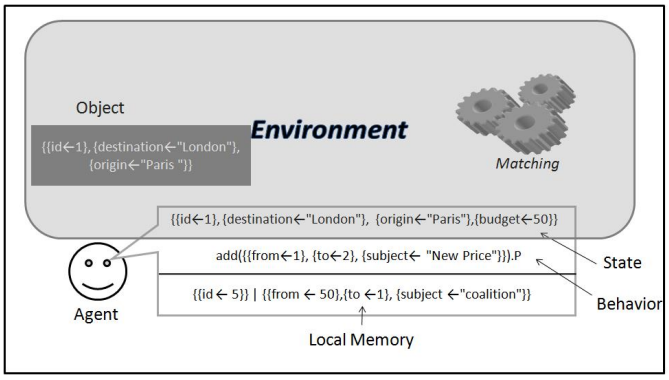

Fig. 2. General Schema

A. Data structure

For LACIOS, we define a standard information system data structure: every item of data in the system has a *description*, i.e. a set of *property←value* pairs, and all the properties of the language are typed. The notions of type, property and description are defined as follows.

Definition 2 (Types): The types of the language are defined as $type_1, \ldots, type_{nbt}$. Every $type_i$ is a set such that $\forall (i,j) \in \{1,\ldots,nbt\}^2, i \neq j, type_i \cap type_j = \{nil\}$

Remark 1: We assume the existence of the *boolean* type in the language, i.e. $\exists i \in \{1,\ldots,nbt\}$, $type_i = \{\text{true}, \text{false}, \text{nil}\}$

Notation 1: We denote the set of values supported by the language as $\mathcal{T} = \bigcup_{i=1}^{nbt} type_i$.

Definition 3 (Property): $\mathcal{N}$ is the property space, and is a countable set of properties. A property $\pi \in \mathcal{N}$ is defined by a type $type(\pi) \in \{type_1, \ldots, type_{nbt}\}$.

The value nil has a twofold use in the syntax of LACIOS. First, it represents every semantic error in a program. When a semantic error is encountered, the corresponding expression is set at nil. Second, a property whose value is equal to nil is considered as undefined (as if it is nonexistent), and is usually omitted.

Notation 2: We note $unknown_\pi$ a value of the type $type(\pi)$ that is defined but doesn't have a value. For instance, $unknown_{destination}$ is a value of the same type as the property $destination$, whose value is (temporarily) unknown.

A description is composed of properties and their corresponding values.

Definition 4 (Descriptions): DS is the set of descriptions. A description is a function that maps properties to values, i.e. $d \equiv \{\pi \leftarrow v_\pi \mid v_\pi \in type(\pi)\}_{\pi \in \mathcal{N}}$. The mapping is omitted when $v_\pi = nil$. We use $d(\pi)$ in order to access the value v_π. For each description, the set of properties $\{\pi \mid d(\pi) \neq nil\}$ is finite.

In LACIOS, each description is associated with an *entity*, which can be an *object* or an *agent*. Objects are defined by their descriptions ($\mathcal{O}$ is the set of objects), while each agent is defined by a description (their state), a behavior and a local memory ($\mathcal{A}$ is the set of agents).

For instance, let o_1 be an object representing a traveler, d_{o_1} could be defined as follows: $\{id \leftarrow$ "$o1$", $destination \leftarrow$ "$London$", $origin \leftarrow$ "$Paris$"$\}$. In this example, $d_{o_1}(origin)$ is equal to "$Paris$".

Definition 5 (Entities): $\Omega = \mathcal{A} \cup \mathcal{O}$ is the set of entities of the MAS. Each entity $\omega \in \Omega$ has a description as defined above denoted by d_ω. The value of the property π of the entity ω is denoted by $d_\omega(\pi)$.

Remark 2: We assume the existence of the type *reference* in LACIOS, a value of the type *reference* designates an entity in Ω, i.e. $\exists i \in \{1,\ldots,nbt\}$, $type_i = \Omega \cup \{nil\}$.

B. Expressions

Expressions are built with values, properties and operators, and are used by agents to describe the data they handle, either locally or to interact with the environment.

Definition 6 (Operators): Each operator op of the language is defined by:

- (i) $arity(op)$ The number of parameters of the operator,
- (ii) $par(op) : \{1,\ldots,\text{arity}(op)\} \rightarrow \{1,\ldots,nbt\}$, $par(op)(i)$ gives the index of the type of the i^{th} parameter of the operator op,
- (iii) $ret(op) \in \{1,\ldots,nbt\}$, the index of the type of the value resulting from the evaluation of op.

For instance, let $type_1 \equiv boolean$. The operator `and` is defined as follows:
$arity(\texttt{and}) = 2$, $par(\texttt{and})(1) = par(\texttt{and})(2) = 1$ and $ret(\texttt{and}) = 1$.

Besides basic operators, additional operators can be defined by the programmer, specifying complex agents' processes. LACIOS is then used mainly for coordination purposes, while the computational model remains non modeled.

An expression may simply be a value, an operator, or a property. For instance, $destination \neq$ "$Paris$" is a (boolean) expression. If an expression is a property, it refers to a property of the agent that is evaluating it. For instance, when $destination$ appears in the behavior of agent a as in the example above, it designates the destination of a. If a property $companion$ of agent a is of the type *reference*, $companion.destination$ designates the destination of the $companion$ of a.

Definition 7 (Expressions): Exp is the set of expressions. An expression $e \in Exp$ is generated via the grammar found in Table I.

$e ::= nil$	
$\mid v$	, with $v \in \mathcal{T} \backslash nil$
$\mid \pi$	, with $\pi \in \mathcal{N}$
$\mid op(e, \ldots, e)$	, with op an operator of the language, and nil doesn't appear in any e
$\mid \pi.e$	, with $\pi \in \mathcal{N}$ and $type(\pi) = \Omega$

TABLE I
SYNTAX OF AN EXPRESSION

In a description, an agent can associate an expression with a property, instead of a value. The result is a *symbolic description* which is transformed into a description when its associated expressions are evaluated.

Definition 8 (Symbolic descriptions): SDS is the set of symbolic descriptions. A symbolic description $sds \in SDS$ is a description that maps properties π to expressions e_π, i.e. $sds \equiv \{\pi \leftarrow e_\pi \mid type(e_\pi) = type(\pi)\}_{\pi \in \mathcal{N}}$.

Below is the definition of the add and $update$ primitives, together with their symbolic descriptions parameters.

$$\mu ::= \ldots \mid add(sds) \mid update(sds)$$

In order to add an object, an agent executes $add(sds)$, and an object whose description is the evaluation of sds is added to Ω_{ENV}. For instance, $add(\{ticket_{id} \leftarrow ticket.id, price \leftarrow ticket.price + 20, owner \leftarrow id\})$ adds an object to the environment whose $owner$ is equal to the id of the calling agent, $ticket_{id}$ is equal to the property id of an object in the memory of the calling agent, referred to by the property $ticket$, and whose $price$ is 20 more than the price paid by the agent. The primitive $update(sds)$ updates a set of properties of the agent with the evaluation of the expressions in sds. When $update(sds)$ is executed, the value of every property π in sds becomes equal to the evaluation of the corresponding expression e_π. For instance, if an agent a executes $update(\{budget \leftarrow budget - 20, destination \leftarrow$ "$Budapest$"$\})$, its *budget* is decreased by 20 and its *destination* becomes "Budapest".

C. Matching

Since we consider a data structure richer than tuples, we also use a matching mechanism richer than templates. To do so, the expressions' syntax is enhanced with *variables*, which designate objects not known by the agent, but which will be discovered during the matching process and will be replaced by objects from the environment before their evaluation. Below is the definition of a variable.

Definition 9 (Variables): $\mathcal{X}$ is the set of variables. A variable $x \in \mathcal{X}$ is defined by its type $type(x) \in \{type_1, \ldots, type_{nbt}\}$.

The syntax of an expression becomes:

$$e ::= \ldots \mid x.e \text{ with } x \in \mathcal{X} \wedge type(x) = \Omega$$

For instance, consider the following boolean expression e: $t.destination =$ "$London$" $\wedge\ t.price \leq budget$. In this boolean expression, t designates an object, unknown for the moment, where the property *destination* of t has to be "London" and the $price$ has to be less than the *budget* of the agent for the expression to be evaluated to `true`. In this case, the agent executing $look$ with e as a parameter will perceive or retrieve the object.

We can now provide the complete definition of the primitive $look$.

$$\mu ::= \ldots \mid look(sds_p, sds_r, e)$$

We choose to use a single primitive to access the environment. The primitive $look(sds_p, sds_r, e)$, with sds_p and sds_r symbolic descriptions, allows both object perception and retrieval (perception and removal from Ω_{ENV}). It blocks until a set of objects C becomes present in Ω_{ENV} such that the expression e is evaluated to `true`. When an agent executes $look(sds_p, sds_r, e)$, the set of objects of the environment C is selected for matching with e (each variable is unified with an object from C). The expression e has to be evaluated to `true` with this unification for $look$ to be executed. The objects associated with the variables in sds_p are perceived and those associated with the variables in sds_r are retrieved. For instance, the following instruction: $look(\{train \leftarrow tr\}, \{ticket \leftarrow tk\}, tr.destination =$ "$London$" $\wedge\ tk.price \leq budget \wedge tk.train = tr.id)$ looks for two objects that will be unified with tk and tr. The object associated with tr will be perceived while the object associated with tk will be retrieved. After this instruction has been executed, the two objects will be present in the local memory of the caller agent, which will have two additional properties of the type *reference*: $ticket$, which refers to the object associated with the variable tk and $train$, which refers to the object associated with tr. The object unified with tk won't be present in Ω_{ENV} anymore.

D. Interaction with External Systems/Users

Consider an agent having two properties $destination$ and $budget$ that are unknown before the execution. The values of these properties come from an external system (e.g. a Web server, a GUI, etc). Here is the description of this agent which properties will be defined during execution resulting from their instantiation by an external system: $\{budget \leftarrow b, destination \leftarrow d\}$, where b and d are variables. Only the action of the external system will be observed, i.e. the assignment of values to the variables, while the system itself is not modeled. We enhance the syntax of an expression with free variables as follows:

$$e ::= \ldots \mid x \text{ with } x \in \mathcal{X}$$

The introduction of the variables for the interaction with an external system is interesting insofar as it clearly separates the coordination aspect - what the MAS does - from the interaction with an external system aspect - the context in which the MAS is running. Thus, in the description $\{budget \leftarrow b, destination \leftarrow d\}$, regardless of which system is instantiating the variables b and d, the definition of the description and the behavior of the agent remain unchanged.

IV. Security Management

We have decided to maintain global sharing of the data between all the agents, and not to isolate them in private environments, thus following the original Linda model. However, this choice leads to the same security problems. More precisely, fraudulent data insertion and retrieval could occur and the agents and the system designer cannot prevent them. In LACIOS, the agents are responsible of the objects that they put in the environment. In order to avoid fraudulent use of these objects, the language supports two control levels, a global level for the designer of the system to control the insertion of objects and a local level for the owner of the objects to control how their object will be used.

A. Global Control

The designer of the system knows the conditions under which certain insertions of objects are fraudulent and we provide him/her with a global control of agents insertions of objects. A threat to authenticity (when an agent tries to forge a message for example) is an example of such fraudulent insertions. More generally, objects added to the environment might corrupt the coherence of the data according to the application logic (resulting in two agents with the same $position$, or with a new bid that is lower than the current one, etc.).

Let us consider for instance, the following action:
$add(\{from \leftarrow companion_2.id, to \leftarrow companion_1.id, subject \leftarrow \text{“}coalition\text{”}\})$

This action is fraudulent, since the agent tries to send a message with a different id than its own. Therefore, this first class of threats concerns the security rules that have to be checked when an add is executed. To overcome threats resulting from the fraudulent adding of objects to the environment, the system designer identifies the critical situations and specifies each one using a security rule s ($s \in \mathcal{S}, \mathcal{S} \subseteq Exp$ is the set of security rules of the system). An expression s in $\mathcal{S}$ is a boolean expression in which the designer specifies the conditions on the state of the agent executing add and the conditions on the description of the object that it adds. To do so, we add a specific key word $that$ in the syntax of an expression to designate, in a security rule, the object added by the agent.

$$e ::= \ldots \mid that.e$$

For instance, here is the expression preventing an agent from adding an object that has a property $from$ that is different from its own: $s \equiv that.from = id$, where id designates the identifier of the agent executing the add and $that$ designates the object added by the agent. When an agent a executes $add(\{from \leftarrow companion_2.id, to \leftarrow companion_1.id, subject \leftarrow \text{“}coalition\text{”}\})$, the security rule specified by the designer is evaluated to false, because $d(from) \neq d_a(id)$, and the operation is canceled.

B. Local Control

The agents of the system know best the conditions under which the perception or retrieval of an object they add is fraudulent, and we provide them with local control to manage the observability of their own objects. A confidentiality threat (e.g. the interception by an agent of another's confidential information or message), or a threat to availability (e.g. the deletion of the agent's information or message by another agent) are examples of such fraudulent access. We propose to allow agents to define the observability rules - on perception and on retrieval - and to let the environment check that these conditions are respected.

This is done by enabling an agent, when it adds an object, to manage its observability, i.e. to identify the situations where the perception or retrieval of the added object is prohibited. To do so, the syntax of the primitive add is replaced as follows.

$$\mu ::= \ldots \mid add(sds, e_p, e_r)$$

where e_p and e_r are boolean expressions. The expression e_p specifies the conditions that an agent has to satisfy to have the right to perceive the object described by sds, and e_r defines the conditions that an agent has to satisfy to have the right to retrieve it. When an agent executes $look(sds_p, sds_r, e)$, for each object $o \in C$ (the set of objects selected for matching from the environment) that is unified with a context variable in sds_p, the expression e_p associated to o has to be evaluated to true, and for each object o unified with a context variable in sds_r, the expression e_r associated with o has to be evaluated to true. Otherwise, the action $look$ cannot be executed with this set of objects. When the agent doesn't want to restrain the perception or the retrieval of the object described by sds, it assigns true to e_p or e_r respectively. For instance, let agent a (let's say that a's $companion.id = 5$) wants to prevent the message it has addressed to its $companion$ to be retrieved by others, and to be perceived by any agent but itself (the key word $that$ has the same semantics here, i.e. it designates the added

object): $add(\{from \leftarrow id, to \leftarrow companion.id, subject \leftarrow$ "coalition"$\}, id = that.from, id = that.to)$

Consider an agent b with $d_b(id) = 10$ that executes $look(\{receiver \leftarrow r\}, \{message \leftarrow m\}, m.to = r.id \wedge r.destination = destination)$. The agent b is trying to retrieve a message (object unified with m) and to perceive the object representing the agent to which m is addressed (object unified with r), if its destination is equal to its own. Thanks to the conditions associated with the added object, b won't be able to perceive a's message. Concretely, any matching that is trying to unify m with a's message is prohibited by the environment and is not considered.

Note that, in the development of the security management defined above, we only take into account the security between local agents and the environment. By doing so, we make two assumptions. On the one side, the $spawn$ of an agent representing an external system, user or agent, has to be fulfilled following a security protocol to ensure that this is indeed the agent with the claimed identifier. On the other side, we assume that local agents don't try to change their identifiers with an $update$ throughout the execution of the system, which is easy to check before the execution of the system. Otherwise, they could dupe the global control mechanism.

V. Specification of Agent Behavior

This section provides the complete definition of an open MAS written in LACIOS, starting with the complete definition of the primitives for LACIOS.

$\mu ::= add(sds) \mid look(sds_p, sds_r, e) \mid update(sds) \mid spawn(P, sds)$

We are now ready to define processes, which define agent behavior. The primitive $spawn(P, sds)$ launches a new agent that behaves like the process P and whose description is the result of the evaluation of sds (its transformation to a description ds). Below is the definition of a process, which defines agents' behaviors.

Definition 10 (Process): Given a set of process identifiers $\{K_i\}_{i \in I}$, a process definition is of the form: $\forall i \in I, K_i \stackrel{def}{=} P_i$, where every P_i is generated via the grammar in Table II.

$P ::= \mathbf{0}$	(null process)
$\mid \mu.P$	(action prefixing)
$\mid b\lfloor P \rfloor + b\lfloor P \rfloor$, where b is a boolean expression	(choice)
$\mid P \| P$	(parallel)
$\mid K_j$, for a certain $j \in I$	(invocation)
$\mid \nu X(P)$	(variables linking)
$\mu ::= spawn(P, sds) \mid add(sds) \mid look(sds_p, sds_r, e) \mid update(sds)$ with e an expression, sds, sds_p and sds_r symbolic descriptions	

TABLE II
PROCESS SYNTAX

Processes, ranged over by $P, Q, \ldots$ represent the programs of the MAS, and the behavior of its agents. A program can be a terminated process $\mathbf{0}$ (usually omitted). It can also be a choice expression between programs $b\lfloor P \rfloor + b\lfloor P \rfloor$, where each P is guarded by the evaluation of a boolean expression: when b is evaluated to true, the program P is executed. A program can also be a parallel composition of programs $P \| Q$, i.e. P and Q are executed in parallel. A program can be an invocation of another process whose identifier is the constant K_j, and which behaves like the process defined by K_j. A program may be a process prefixed by an action $\mu.P$. Actions are the language primitives, as defined earlier. The operator ν is introduced to link free variables in P. The process $\nu X(P)$ introduces nondeterminism in the agents' behaviors. Indeed, behaves like P where every free variable (in X) is nondeterministically linked with a value in its type.

A coordinated MAS is then defined as follows.

Definition 11 (Coordinated MAS): $CS = \langle \Omega, d, \Omega_{ENV}, \mathcal{S} \rangle$

- $\Omega = \mathcal{A} \uplus \mathcal{O}$ is the set of entities, composed of $\mathcal{A}$ the set of agents and $\mathcal{O}$ the set of objects,
  - $\mathcal{A} \subseteq \Omega$ is the set of agents.
    - Ω_a is the private memory of agent a, $\Omega_a \subseteq \mathcal{O} \cup \{a\}$, i.e. the agent has access to its own description,
    - $proc(a)$ is the process defining the behavior of a.
  - $\mathcal{O} \subseteq \Omega$ is the set of objects.
    - $e_p(o)$ returns the predicate specifying the perception conditions of o, i.e. which agents can perceive o.
    - $e_r(o)$ returns the predicate specifying the retrieval conditions of o,
- $d : \Omega \rightarrow (\mathcal{N} \rightarrow \mathcal{T})$ is the description function of the MAS, each $d(\omega)$ is an entity description as described before (denoted by d_ω as well),
- $\Omega_{ENV} \subseteq \mathcal{O}$ is the set of objects that are in the environment,
- $\mathcal{S} \subset Exp$ is the set of predicates specifying the conditions that have to be verified, in order for an add to be executed.

VI. The Programming Language JAVA-LACIOS

We have defined a language that, following its operational semantics (cf. [13]), could be implemented in any host language. The usual procedure in order to implement a coordination language is to provide libraries in a host programming language that can be used by any system wishing to follow the coordination model (e.g. Klava, which is associated to Klaim [6]). However, to take full advantage of the language semantics, it is more useful not to require the programmer himself/herself to respect the semantics in each system that he/she implements. This is possible by providing him/her with a tool allowing to write a program in LACIOS's syntax, and to generate a system ready to be executed, with the guarantee that it respects the language semantics. In particular, we want to use the operators prefixing, choice and parallel composition when defining the agents' behaviors. Java has been chosen as a target programming language in which a compiled LACIOS program is translated, because of the relative simplicity of Thread management, as well as the easy creation of parsers thanks to the parser generator JavaCC [1].

[1] http://javacc.dev.java.net/

A JAVA-LACIOS program is a file where both the behaviors and the initial state of the coordinated MAS are described. A coordinated MAS is defined by the set of initial agents, spawned when the program starts, together with the security rules $\mathcal{S}$. Programmers write their scripts which are parsed and compiled, generating a Java program. We have proposed a GUI for JAVA-LACIOS, which displays the ongoing execution, the current objects in the environment, the current agents' behaviors that are executed, etc. The Fig. 3 illustrates the execution with a Dial A Ride system that we have implemented [14]. It is also possible, before the execution, to visualize the graphs (labeled transition systems) related to the agents' behaviors.

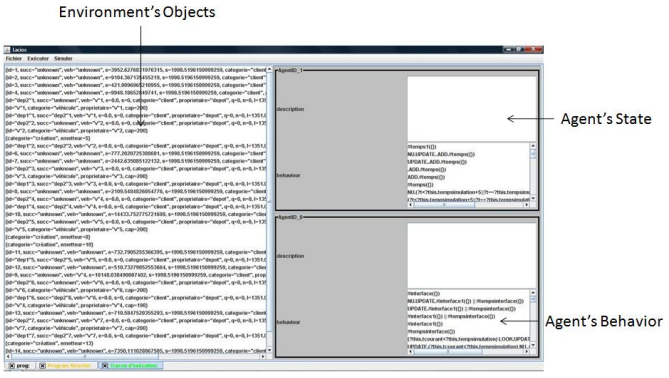

Fig. 3. Visualization of agents behaviors

The concurrent access to the environment objects with a $look$ necessitates a synchronization of the add and $look$ calls. However, an agent calling add has not to be blocked until the environment releases the lock. To this end, we define a buffer to which agents can add objects without blocking, while emptying the buffer is synchronized with the $look$ calls.

when a $look$ is called, the environment is locked while it is still looking for a matching, to guarantee that an agent does not access the environment in an incoherent state, and to be sure that a same object is not retrieved by more than one agent. If no matching is found, the calling process of the agent is blocked. The blocked processes are notified when an object is added to the environment. In this case, the notified process looks for a matching with the only newly added objects.

An $update$ modifies the agent's properties *locally*, but it however influences its interaction with the environment. Indeed, if a $look$ is currently executing, the matching have to be attempted with the current properties of the agent. When the properties of an agent change, and when they concern properties for which an ongoing $look$ has attempted to match, the execution of the $look$ is executed again, and the pending $look$ requests are notified since they might be concerned by the newly changed properties as well.

VII. DISCUSSION AND RELATED WORK

Security is generally enforced by using multiple (logical) spaces, by stating "Interaction Laws" or by defining roles and access rights associated to them. With multiple spaces (e.g. Klaim [6], SecSpaces [8], [4] and SecOS [9]), two agents, which wish to exchange confidential data use a space that is known only by the two of them. However, when security is guaranteed by isolating the data in private spaces, accessing one space gives the possibility to access all of its data, and being excluded from one space means not having access to any of its data. In LACIOS, agents have a state, and an agent can protect its data in a fine-grained way (at object level) without knowing the other agents, which allows secure interaction with complete data sharing. Roles and role access rights (like in the RBAC model associated to TuCSon [7]) are an additional layer on top of multiple spaces, and therefore security is also defined in a coarse-grained way.

In [8], specific cryptographic fields are added to the tuples to authenticate the producer of an item of data, for instance, as well as to identify the readers/takers of that item. This authentication is carried out in LACIOS thanks to agents' states and security rules, but it is nevertheless still possible to define a specific property for cryptographic fields.

The laws in LGI [10] allow data to be secured by specifying conditions on the states of the agents and the content of the data. Tuple-space reactions are associated with agents' actions so as to always result in a coherent configuration. Nevertheless, two points differentiate LACIOS from LGI. First laws in LGI are defined by the system designer only, whereas agents cannot do this. Second, laws are active rules, which poses the problem of non-termination of the matching process (action and a chain of endless reactions). In LACIOS, the rules cancel perceptions or retrievals but don't launch any reaction, so the problem of non-termination does not occur.

Tagged Sets [15] allow fine-grained protection of data added to the data space. However, neither agents' states nor powerful comparison operators are defined for it as in LACIOS.

In [5], the authors point out that "the secure version of Lime [[16]] is the only one which permits to control output operations, and SecSpaces [[8]] is the only one which permits to distinguish between the processes that can consume and the processes that can read a certain tuple". LACIOS allows for both insertions control and the distinction between reading a datum and taking it.

VIII. CONCLUSION AND PERSPECTIVES

The investigation of security issues in data driven coordination languages has lead us to propose a modified language allowing for fine-grained protection of exchanged data via the shared space. This paper has defined LACIOS, which can be used to model a large number of applications in which agents join and leave the system freely, where agents interact with external systems, and where security is crucial. Using LACIOS makes it easier for open MAS designers to translate the concepts manipulated by the agents and their interaction needs to LACIOS syntactic constructs, ensuring information security and expressing complex constraints. We have demonstrated this usefulness for a complex transportation application in our recent paper [14].

Our proposal is an entity oriented approach, and allows for the control of objects' insertion, perception and retrieval. It

distinguishes between objects' perception control and objects' retrieval control. The addition of agents' states, of *property-value* pairs data model, together with operators and variables lead us to propose a new language instead of building on top of an existing one. The formal operational semantics of LACIOS can be found in [13].

Our future works include the consideration of specific cryptographic properties to ensure authenticity. We are also investigating the addition of time constructs to LACIOS, inspired by the works of Busi *et al.* (e.g. [17]) and Linden *et al.* (e.g. [18]), to express temporary objects insertion and to define a deadline for $look$ before termination with no effect. Interaction over multiple hosts is very challenging, yet with simple spaces, and with contextual interaction and the security mechanism, this becomes even more difficult to fulfill. Since we don't program mobile agents (as in Klaim [6] or Claim [19]), and since as a consequence the agents' locations are transparent at the language level, the concern of the environment distribution is to be tackled at the implementation level. Our ongoing research investigates the definition of architectures and strategies providing guidelines for environment distribution for LACIOS implementation.

REFERENCES

[1] D. Gelernter, "Generative communication in linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[2] P. Ciancarini, "Coordination languages for open systems design," in *Proceedings of the International Conference on Computer Languages (ICCL'90)*. New Orleans, LA (USA): IEEE Computer Society, 1990, pp. 252–260.

[3] C. P. Pfleeger and S. L. Pfleeger, *Security in Computing*. Prentice Hall Professional Technical Reference, 2002.

[4] M. Bravetti, N. Busi, R. Gorrieri, R. Lucchi, and G. Zavattaro, "Security issues in the tuple-space coordination model," in *Formal Aspects in Security and Trust*, T. Dimitrakos and F. Martinelli, Eds. Springer, 2004, pp. 1–12.

[5] R. Focardi, R. Lucchi, and G. Zavattaro, "Secure shared data-space coordination languages: A process algebraic survey," *Sci. Comput. Program.*, vol. 63, no. 1, pp. 3–15, 2006.

[6] R. De Nicola, G. L. Ferrari, and R. Pugliese, "Klaim: A Kernel Language for Agents Interaction and Mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 315–330, 1998.

[7] A. Omicini, A. Ricci, and M. Viroli, "RBAC for organisation and security in an agent coordination infrastructure," *ENTCS*, vol. 128, no. 5, pp. 65–85, 2005, proceedings of the 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems (SecCo 2004).

[8] N. Busi, R. Gorrieri, R. Lucchi, and G. Zavattaro, "Secspaces: a data-driven coordination model for environments open to untrusted agents," *Electr. Notes Theor. Comput. Sci.*, vol. 68, no. 3, 2003.

[9] J. Vitek, C. Bryce, and M. Oriol, "Coordinating processes with secure spaces," *Sci. Comput. Program.*, vol. 46, no. 1-2, pp. 163–193, 2003.

[10] N. H. Minsky, Y. Minsky, and V. Ungureanu, "Safe tuplespace-based coordination in multiagent systems," *Applied Artificial Intelligence*, vol. 15, no. 1, pp. 11–33, 2001.

[11] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.

[12] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1989, 272 pages.

[13] M. Zargayouna, "Coordination model and language for open multiagent systems. application to the dial-a-ride problem," PhD Dissertation, University of Paris-Dauphine, Paris (France), 2007, in french.

[14] M. Zargayouna, F. Balbo, and G. Scmama, "A data-oriented coordination language for distributed transportation application," in *The third International KES Symposium on Agents and Multi-agent Systems Technologies and Applications (KES-AMSTA'09)*, ser. Lecture Notes in Artificial Intelligence. Uppsala (Sweden): Springer-Verlag, 2009, vol. 5559, pp. 283–292.

[15] M. Oriol and M. Hicks, "Tagged sets: a secure and transparent coordination medium," in *Proceedings of the International Conference on Coordination Models and Languages (COORDINATION)*, ser. Lecture Notes in Computer Science, J.-M. Jacquet and G. P. Picco, Eds., vol. 3454. Springer-Verlag, April 2005, pp. 252–267.

[16] A. L. Murphy, G. P. Picco, and G.-C. Roman, "LIME: A coordination model and middleware supporting mobility of hosts and agents," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 3, pp. 279–328, 2006.

[17] N. Busi and G. Zavattaro, "Expired data collection in shared dataspaces," *Theoretical Computer Science*, vol. 298, no. 3, pp. 529–556, 2003.

[18] I. Linden, J.-M. Jacquet, K. D. Bosschere, and A. Brogi, "On the expressiveness of timed coordination models," *Science of Computer Programming*, vol. 61, no. 2, pp. 152–187, 2006.

[19] A. Suna, "Claim & sympa : An environment for programming intelligent and mobile agents," PhD Dissertation, University of Paris VI, 2005, in french.