Language for Implementing Multiagent Transportation Applications

Mahdi Zargayouna
INRETS Institute,
Gretia Laboratory
Building “Descartes II”
2 rue de la Butte Verte
93160 Noisy le Grand Cedex
France
zargayouna@inrets.fr

Flavien Balbo
University of Paris Dauphine,
CNRS-Lamsade Laboratory
Place Maréchal de Lattre de Tassigny
75775 Paris Cedex 16
France
balbo@lamsade.dauphine.fr

Gérard Scemama
INRETS Institute,
Gretia Laboratory
Building “Descartes II”
2 rue de la Butte Verte
93160 Noisy le Grand Cedex
France
scemama@inrets.fr

ABSTRACT

The multiagent paradigm is well suited to designing transportation applications. However, when implementing these applications in highly dynamic environments, it is very costly to use languages relying on channel-based communication. Data driven coordination languages rely on a shared space in which agents add, read and retrieve data. They are intuitively relevant for distributed transportation applications, where different actors evolve in a highly dynamic and very constrained environment. However, existing coordination languages can hardly be used for transportation applications, because they fail to express agents complex interaction needs and to ensure secure data exchanges. Indeed, in transportation applications, the interaction needs of the agents are driven by their current context and by ambient conditions, and information security is usually important. In this article, we propose a data driven coordination language tackling these issues, and we define a programming language on top of Java allowing to use the language syntax while executing it in Java. We illustrate our proposal with two applications: a traveler information system and a demand-responsive transport service.

Keywords

Data Driven Languages, Applications, Security

1. INTRODUCTION

Transportation applications rely more and more on networks of mobile devices such as vehicle and mobile ad-hoc networks. These applications are increasingly based on entities which interaction follows a dynamically reconfigurable communication scheme. These relatively new classes of applications are calling for new models and languages adequate for their design, management and implementation. Indeed, developing such logically distributed applications, without knowing either the overall structure of the system is a challenge and needs specific models and languages to simplify their implementation.

Based on our previous developments (e.g. [1, 18]), we identify three recurrent issues when dealing with transportation applications. The first is related to knowledge management. In many problems such as urban network regulation [1], the knowledge is incomplete or is only known by human experts. The knowledge issue is also related to the coordination of many information sources as in the traffic regulation problem [9]. The multiagent paradigm is well suited to deal with this issue. This paradigm gives the necessary abstraction level to take into account incomplete knowledge. For Parunak [10], “Agent-based modelling is most appropriate for domains characterized by a high degree of localization and distribution”.

The second issue is the dynamics of the real environment, which impacts the quality of the information, makes direct communication difficult and costly and/or implies that mobile entities appear and disappear. The data driven coordination models are well suited to deal with this dynamics. The principle is to use a shared dataspace and to let agents opportunistically get the information they need, without having to maintain any knowledge about the other agents of the system.

The last issue is the security issue, since users usually need to avoid being tracked or localized by other agents or authorities. For example, in the design of location-based applications, the user has to be sure that his needs remain personal and can not be used by other people.

LACIOS\(^1\) is a data driven coordination language designed to take into account these three issues. It builds on top of data driven languages enabling the programmer to design and implement secure multiagent systems for transportation applications. To illustrate our proposal and the syntax of LACIOS, a transportation 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.

This paper is organized as follows. Section 2 presents the existing solutions to these three issues. Section 3 gives an overview of the language that we are proposing. Section 4 describes LACIOS and details its syntax. In section 5, we detail the security management in LACIOS. Section 6 presents the programming language JAVA-LACIOS. In section 7, we present the traveler information system based on LACIOS.

\(^1\)Language for Agent Contextual Interaction in Open Systems
and the coordination environment for a demand-responsive transport service. Finally, section 8 concludes the paper.

2. STATE OF THE ART

Existing multiagent programming languages (such as Agent0 [13], AgentSpeak [12], dMars [3] and Claim [14], to cite only a few) gives the necessary abstraction level in the implementation of the agents. However, the fact that they rely on a channel-based communication makes it hard to design and implement open systems in which agents join and leave the system freely. Instead, data driven coordination languages, with the pioneer language Linda [6] and its extensions such as IBM’s TSaPces\textsuperscript{®} [16] and Sun’s JavaSpaces\textsuperscript{®} [5], 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.

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 contents) rather than by address. Every tuple is a sequence of one or more typed values. Communication in Linda is said to be generative: an agent generates a tuple and its life cycle is independent of the agent that created it. The tuplespace is manipulated by three atomic primitives: out to add a tuple, rd to read and in to take a tuple (read it and remove it from the tuplespace). The parameter of out is a fully instantiated tuple (sequence of values), and the parameter of an in or a rd primitive is a template: a tuple with potentially one or more formal fields (variables). A tuple and a template match if they have the same arity and if every field in the tuple is either equal to the corresponding value or of the same type of the corresponding variable in the template. The primitives in and rd are blocking: if no tuple matches their parameter template, the caller agent is suspended until a matching tuple is present in the tuplespace. An additional primitive, eval(P), launches a new agent that will run in parallel with the caller.

Nevertheless, systems adhering to data driven coordination models have two main limitations concerning the language expressiveness and the security management. Agents in the coordination languages literature are stateless processes, the language describe what they do, but not what they are. In the absence of agents’ states, agents interaction cannot be conditioned by their own current state or context. The interaction conditions are also poor (limited to the type, position and value of the components of the tuples) and unable to express agents’ complex concepts and interaction needs.

Besides, in the literature the tuple data structure is enriched on the implementation level (object oriented [5], relational [16], logic [8]) but not on the model level. The result is that the matching mechanism used remains based on templates, which limits the expressive power offered to agents to express their interactional needs. For example, the poor expressiveness of Linda-like languages materializes in the cases where an agent desires to condition its interaction by the status of data that are different from the data it wants to access. In a traveler information system for instance, a traveler agent could be interested by bus tickets if the weather is sunny, and underground tickets if it is rainy. The data it wants to access are tickets, but it is conditioned by other data (the weather conditions).

When used in untrusted environments, systems adhering to data driven coordination models encounter several security threats due to their information sharing. The authors in [11] classify security threats into: i) threat on confidentiality; ii) threat on integrity; iii) threat on availability; iv) threat on authenticity. For instance, consider a Traveler agent in a traveler information system, that adds a message to the shared data space, claiming that this message comes from a Train agent and informing about a delay. This information may mislead the other Traveler agents, which will miss the train if they get this information and act in consequence. The confidentiality threats are related to the interception by an agent of another agent’s confidential information or message, and the threats on availability concern the deletion of an agent’s information or message by another one. For instance, consider a Traveler agent t that tries to read or take a message from a Train agent to another Traveler agent with an information about the ticket price it is proposing. The agent t has to be prevented from reading (confidentiality) and taking (availability) this message. Finally, the integrity threats are related to the modification by other agents of another agents’ messages or information.

In the literature, several modifications have been made to Linda-like languages to tackle the security threats. In [4], the authors classify secure data driven languages into entity driven and knowledge driven languages. The idea behind the knowledge driven approach is that tuple spaces, tuples or single data fields are decorated with additional information. Agents can access the resources only in the case they prove their knowledge of this information. In the case of the entity driven approach, additional information associated to resources lists the agents that are allowed to access the resources. 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. Our proposal can be classified as an entity driven approach. However, instead of listing the agents that can access a datum, we propose a language in which these agents are described symbolically, i.e. their properties are defined without pointing them namely. Indeed, because of this listing requirement, the authors in [4] state that the knowledge driven approach is more suitable for open systems; In this sense, we propose an entity driven approach that is suitable for open systems. 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. Equipping data driven coordination languages is the objective pursued in this article. To this end, we perform several modifications of the shared space model, and propose a new language, called LACIOS, which is the linguistic embodiment of the modified model.

3. OVERVIEW

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. 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 in the system. 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 via a local agent. 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 (like agents 2, 3 and 4).

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. Distant agents (like agent_2) provide their new state to the local agent whenever it changes. 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.

**Figure 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 [15]:

**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: i) to support a global control by the environment of objects’ insertion by the agents in order to ensure that the new objects are not fraudulent (authenticity and availability), ii) 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, availability and integrity). 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 the data at language level. These states are defined as a set of property-value pairs (e.g., \{identifier ← 10, position ← node_i\}). 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). The second modification is the proposal of a rich interaction mechanism, using operators and variables, which allow to express the two levels of security management: local and global control. The next section is dedicated to the language specification.

## 4. The Coordination Language LACIOS

### 4.1 Overview

LACIOS is a coordination language that extends Linda for the design and implementation of MAS, defined in a suitable way for transportation applications. For the specification of agent behavior, we adopt four primitives inspired by Linda and Milner’s CCS [7]. A MAS written in LACIOS is defined by a dynamic set of agents interacting with an environment - denoted $\Omega_{\text{ENV}}$, which is composed of a dynamic set of objects. Agents can perceive (read only) and/or retrieve (read and take) objects from the environment. Agents are defined by a behavior (a process), a state and a local memory in which they store the data they perceive or retrieve from the environment.

The distinguishing features of LACIOS that we focus on in this paper can be summarized as follows. First, an agent can publish its state, update it and use it to condition its interaction with the environment. Second, the data structure (for exchanged data and for agents’ states) is based on typed property-value pairs. Finally, an agent can use complex conditions (using operators and functions) on its own state and on other shared objects in order to access the environment, with a single instruction.

We have proposed an operational semantics, unambiguously defining the behavior of a MAS written in LACIOS, which can be found in [17].

### 4.2 Syntax

#### 4.2.1 Data Structure

For LACIOS, we define a standard information system data structure: every datum in the system has a description, i.e. a set of property-value pairs, and all the properties of the language are typed. We define in the following the notions of a type, a property and a description.

**Definition 2 (Types).** The types of the language are defined as type_1, ..., type_{nb_t}. Every type_i is a set such that $\forall (i, j) \in \{1, \ldots, \text{nb}_t\}^2, i \neq j, \text{type}_i \cap \text{type}_j = \{\text{nil}\}$

**Remark 1.** We assume the existence of the boolean type in the language, i.e. $\exists d \in \{1, \ldots, \text{nb}_t\}, \text{type}_d = \{\text{true, false, nil}\}$

**Definition 3 (Property).** $\mathcal{N}$ is the property space, it is a countable set of properties. A property $\pi \in \mathcal{N}$ is defined by a type $\text{type}(\pi) \in \{\text{type}_1, \ldots, \text{type}_{\text{nb}_t}\}$.

A description is composed of properties and their corresponding values.

**Definition 4 (Descriptions).** $\mathcal{D}$ is the set of descriptions. A description is a function that maps properties to values, i.e. $d \equiv \{\pi \leftarrow v_\pi \mid v_\pi \in \text{type}(\pi)\}_{\pi \in \mathcal{N}}$. The mapping is omitted when $v_\pi = \text{nil}$. We use $d(\pi)$ in order to access the value $v_\pi$. For every description, the set of properties $\{\pi \mid d(\pi) \neq \text{nil}\}$ is finite.
A property evaluated to nil is considered undefined. In Lacios, every description is associated with an entity. An entity can be an object or an agent. An object is defined by its description (O is the set of objects), while an agent is defined by a description and a behavior (A is the set of agents).

**Definition 5 (Entities).** $\Omega = A \cup O$ is the set of entities of the MAS. Each entity $\omega \in \Omega$ has a description as defined above denoted by $d_\omega$. The value of the property $\pi$ of the entity $\omega$ 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 $\Omega$, i.e. $\exists \pi \in \{1, \ldots, nbt\}$, $\text{type}_\pi = \Omega \cup \{\text{nil}\}$.

For instance, let $a_1$ be an object, $d_{a_1}$ could be defined as follows: $(\text{id} \leftarrow \text{“01”}, \text{destination} \leftarrow \text{“Montreal”}, \text{from} \leftarrow \text{“Paris”})$. In this example, $d_{a_1}(\text{from})$ is equal to “Paris”.

### 4.2.2 Expressions

Expressions are built with values, properties and operators. We define an operator as follows.

**Definition 6 (Operators).** Each operator $\text{op}$ of the language is defined by:

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

For instance, let $\text{type}_1 \equiv \text{boolean}$. The operator and is defined as follows:

$\text{arity}($and$) = 2, \text{par}(\text{and})(1) = \text{par}(\text{and})(2) = 1$ and $\text{ret}(\text{and}) = 1$.

Expressions are used by agents to describe the data they manipulate, either locally or to interact with the environment. An expression may simply be a value, an operator, or a property. 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 the agent $a_1$, it designates the destination of $a_1$. If a property neighbor of the agent $a$ is of type reference, neighbor.destination designates the destination of the neighbor (an entity) of $a$.

**Definition 7 (Expressions).** Exp is the set of expressions. An expression $e \in \text{Exp}$ is generated via the grammar of Table 1.

We can now associate an expression with a property instead of a value in a description. 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 is a description that maps properties $\pi$ to expressions $e_\pi$, i.e. $\text{SDS} \equiv \{\pi \leftarrow e_\pi \mid \text{type}(e_\pi) = \text{type}(\pi)\}_{\pi \in \mathcal{N}}$.

<table>
<thead>
<tr>
<th>$e := $</th>
<th>nil</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mid v \mid$</td>
<td>, with $v \in \mathcal{T}\backslash \text{nil}$</td>
</tr>
<tr>
<td>$\mid \pi \mid$</td>
<td>, with $\pi \in \mathcal{N}$</td>
</tr>
<tr>
<td>$\mid \text{op}(e_1, \ldots, e_n) \mid$</td>
<td>, with $\text{op}$ an operator of the language, and nil doesn’t appear in any $e$</td>
</tr>
<tr>
<td>$\mid \pi.e \mid$</td>
<td>, with $\pi \in \mathcal{N}$ and $\text{type}(\pi) = \Omega$</td>
</tr>
</tbody>
</table>

**Table 1: Syntax of an expression**

### 4.3 Matching

Since we consider a data structure richer than tuples, we also use a matching mechanism richer than templates. To do so, we enhance the expressions’ syntax with entity variables, which designates objects not known by the agent, but will be discovered during the matching process and will be replaced by objects from the environment before their evaluation. Here follows 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 $\text{type}(x) \in \{\text{type}_1, \ldots, \text{type}_{\text{nbt}}\}$.

The syntax of an expression becomes:

| $e ::= $ | $\cdots | x.e$ with $x \in \mathcal{X} \land \text{type}(x) = \Omega$ |

For instance, consider the following boolean expression $e$: $t.\text{destination} = \text{“London”} \land t.\text{ind}_{\text{price}} \leq \text{budget}$. In this expression, $t$ designates an object, unknown for the moment, where $t$ has to have as destination “London” and an $\text{ind}_{\text{price}}$ (standing for individual price) less than the $\text{budget}$ of the agent for the expression to be evaluated true, in which case the agent executing look with $e$ as a parameter will perceive or retrieve the object.

In the previous example, a single object from the environment (unified with $t$) is needed for the expression to be evaluated. However, the introduction of variables allows for a richer matching. The matching in Lacios materializes what we call a contextual interaction, which is the type of interactions that uses the state of the agent and the state of several objects in the environment to access a set of objects, instead of only one like in Linda templates.

To illustrate the need for this type of interactions, consider the following scenario. Let’s say that train agents add objects representing tickets in the environment, designating remaining empty seats. These objects are of two types. The first type is “individual ticket” representing a single empty seat in the train. It’s described by the following properties: $\text{id}$ designating the identifier of the train, $\text{destination}$ designating his destination and $\text{ind}_{\text{price}}$ designating the price of the ticket. The second type is “group ticket” representing a group of empty seats in the train (we consider that group prices are applicable starting from three tickets bought together). Group tickets are cheaper than individual tickets and are described by the following properties: $\text{id}$ designating the identifier of the train, $\text{destination}$ designating it destination and $\text{group}_{\text{price}}$ designating the price of the ticket. However, in order to buy such a ticket, a single traveler agent cannot ask for it alone, and has to find two other agents that have the same destination and enough money to buy the group ticket together. Without contextual interaction, an agent has to first look for a ticket with a $\text{group}_{\text{price}}$ that
is lower than his own budget before to search individually for two agents with enough budget and the same destination as his. Meanwhile, the ticket could have been bought by another agent, or when the agent finds a first traveler and looks for a second, the first could’ve bought another ticket. A contextual interaction allows for an all-or-nothing request, including his own state and several objects from the environment. In a single pace, the agent would find the ticket and both agents that would share the ticket price with him. The corresponding expression to this example is the following:

\[(t.\text{destination} = \text{destination}) \land (\text{budget} \geq t.\text{groupprice}) \land (x.\text{destination} = \text{destination}) \land (x.\text{budget} \geq t.\text{groupprice}) \land (y.\text{destination} = \text{destination}) \land (y.\text{budget} \geq t.\text{groupprice})\]

The variable \(t\) designates the ticket, \(x\) and \(y\) represent the two needed agents. This expression needs three objects from the environment in order to be evaluated: a ticket and two agents’ descriptions. Either the agent will find the three needed objects and buy the ticket or it won’t, in which case it won’t launch any further useless interaction.

We can now provide the complete definition of the primitive \textit{look}. We choose to use a single primitive to access the environment (instead of \textit{in} and \textit{rd}). The primitive \textit{look}(\textit{sdsp}, \textit{sdss}, \(e\)), with \(sdsp\) and \(sdss\) symbolic descriptions, allows both object perception and retrieval (perception and removal from \(\Omega_{\text{ENV}}\)). It blocks until a set of objects \(C\) becomes present in \(\Omega_{\text{ENV}}\) such that the expression \(e\) is evaluated to true. When an agent executes \textit{look}(\textit{sdsp}, \textit{sdss}, \(e\)), the set of objects of the environment \(C\) is selected for matching with \(e\) (an object for each context variable), and \(e\) has to be evaluated to true for the consequence of the action to be effective. The objects associated with the variables in \(sdsp\) are perceived and those associated with the variables in \(sdss\) are retrieved. For instance, the following instruction: \textit{look}(\{\text{ticket} \leftarrow t\}, \{\text{paper} \leftarrow p\}, t.\text{destination} = \text{“London”} \land t.\text{budget} \geq \text{budget} \land p.\text{decision} = \text{“accepted”}) looks for two objects that will be unified with \(t\) and \(p\). The object associated with \(t\) will be perceived while the object associated with \(p\) will be retrieved. After the execution of this instruction, the two objects will be present in the local memory of the caller agent, who will have two additional properties of type reference: ticket that refers to the object associated with the variable \(t\) and paper that refers to the object associated with \(p\).

4.4 Interaction with External Systems/Users

Consider an agent having two properties \textit{destination} and \textit{budget} that are unknown before 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: \{\text{budget} \leftarrow b, \text{destination} \leftarrow d\}, where \(b\) and \(d\) are variables. When writing his/her program, the programmer doesn’t have to care about the provenance of this information, only the action of the external system will be observed on execution time, i.e. the assignment of values to the variables, while the external system itself is not modeled.

We enhance the syntax of an expression with free variables as follows:

\[e ::= \cdots | x \text{ with } x \in \mathcal{X}\]

The introduction of the variables for the interaction with an external system is interesting 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 \{\text{budget} \leftarrow b, \text{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.

5. 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 encountered with the original model. 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. The global level guards every object insertion in the environment. It verifies whether the agent’s properties gives it the right to insert the object. This check would notably guarantee the authenticity of the emitted messages, by checking that the agent is not trying to forge a message. The local level guards every object perception or retrieval. It verifies that the object’s owner allows the agent trying to read or take the object to do so. These checks guarantee the confidentiality, the availability and the integrity of the exchanged data. It is noteworthy that LACiOS manages security following the original Linda model, i.e. associatively. The communication remains anonymous and the security is managed symbolically, together with keeping a complete sharing of the data between the agents of the system.

5.1 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 objects insertions by the agents. A threat to authenticity (when an agent tries to forge a message for example) is an example of such a 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 \textit{position}, or with a new bid that is lower than the current one, etc.).

Let us consider for instance, the following action:

\[\text{add}(\{\text{from} \leftarrow \text{companion2}.id, \text{to} \leftarrow \text{companion1}.id, \text{subject} \leftarrow \text{“coallition”}\})\]

This action is fraudulent, since the agent tries to send a message to its first companion with a different \(id\) than its own \(id\) (\(id_{\text{from}} \neq \text{id}\)).

This first class of threats (which includes the authenticity threat) concerns the security rules that have to be checked when an \textit{add} is executed. To overcome threats resulting from the fraudulent insertion of objects in the environment, the system designer identifies the critical situations and specifies each one using a security rule \(s\) (\(s \in \mathcal{S}, \mathcal{S} \subseteq \mathcal{Exp}\) 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.

\[ e ::= \ldots | that.e \]

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 \( add\{from \leftarrow companion_2.id, to \leftarrow companion_1.id, subject \leftarrow \text{"coalition"}\} \), the security rule specified by the designer is evaluated to false, because \( d\{\text{from}\} \neq d_a(\text{id}) \), and the operation is canceled.

### 5.2 Local Control

The agents of the system know 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 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 inserted object is prohibited. To do so, the syntax of the primitive add is replaced as follows.

\[ \mu ::= \ldots | add(sds, e_p, e_r) \]

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

Consider an agent \( b \) with \( d_0(id) = 10 \) that executes \( look\{receiver \leftarrow r\}, \{message \leftarrow m\}, m.to = r.id \land r.destination = destination \)

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.

Aside from authenticity, confidentiality and availability, a fourth threat for information security is integrity. The integrity threats are related to the modification by other agents of another agents’ messages or information. However, since the update primitive is local in LACIOS, no agent can modify others’ information. The only way to do it - as it is the case for all data driven languages - is to take an object and to create a new one with different information. Taking an object is already guarded by security rules against interception. As a consequence, there is no need for a specific mechanism for the threats on integrity in LACIOS.

A fifth threat on information security is traditionally listed, which is the non-repudiation threat, implying that one party of a transaction cannot deny having received a transaction nor can the other party deny having sent a transaction. Due to the generative communication à la Linda, we cannot map a certain message to its sender or receiver. None of the data driven coordination languages tackle this issue. Nevertheless, when implemented, a system adhering to our model can use a log with such a mapping, that can be read by agents, to solve this problem. Provided that the authenticity is guaranteed with our global control mechanism, the content of this log is certified correct.

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 prohibit local agents from trying to change their identifiers with an update throughout the execution of the system. We enforce this prohibition by stating a single value in the definition domain of the concerned property; as a consequence, changing these properties would provoke a semantic error.

### 5.3 Specification of Agent Behavior

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

\[ \mu ::= add(sds, e_p, e_r) \mid look(sds_p, sds_r, e) \mid update(sds) \mid spawn(P, sds) \]

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: \( \forall i \in I, K_i \equiv P_i \), where every \( P_i \) is generated via the grammar in Table 2.

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 $0$ (usually omitted). It can also be a choice expression between programs $b[P] + b[P]$, 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 $\nu$ 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). $CS = (\Omega, d, \Omega_{ENV}, S)$

- $\Omega = A \cup O$ is the set of entities, composed of $A$ the set of agents and $O$ the set of objects,
  - $A \subseteq \Omega$ is the set of agents.
    * $\Omega_a$ is the private memory of agent $a$, $\Omega_a \subseteq O \cup \{a\}$, i.e. the agent has access to its own description,
    * proc($a$) is the process defining the behavior of $a$.
  - $O \subseteq \Omega$ 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 $o$.

- $d : \Omega \rightarrow (N \rightarrow T)$ is the description function of the MAS, each $d(\omega)$ is an entity description as described before (denoted by $d_\omega$ as well),

- $\Omega_{ENV} \subseteq O$ is the set of objects that are in the environment,

- $S \subseteq Exp$ is the set of predicates specifying the conditions that have to be verified, in order for an add to be executed.

Table 2: Process syntax

\[
P ::= 0 \quad \text{(null process)}
| \mu P \quad \text{(action prefixing)}
| b[P] + b[P] \quad \text{(choice)}
| P||P \quad \text{(parallel)}
| K_j \quad \text{(invocation)}
| \nu X(P) \quad \text{(variables linking)}
\]

\[
\mu ::= \text{spawn}(P, sds) \mid \text{add}(sds, e_p, e_r) \mid \text{look}(sds_p, sds_r, e) \mid \text{update}(sds)
\]

with $e$, $e_p$ and $e_r$ expressions, $sds, sds_p$ and $sds_r$ symbolic descriptions

6. THE PROGRAMMING LANGUAGE JAVA-LACIOS

We have defined a language that, following its operational semantics (cf. [17]), 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 adhere to the coordination model (e.g. Klava, which is associated to Klaim [2]). 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 of 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 2.

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 $S$. 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. 2 illustrates the execution with a Demand-Responsive Transport Service that we have implemented and that will be described in the following section. It is also possible, before the execution, to visualize the graphs (labeled transition systems) related to the agents’ behaviors as shown in Fig. 3.

The translation of a LACIOS program to a Java program is quite straightforward, but in order to respect the semantics of the language, a few details are noteworthy. On the one hand, the concurrent access to the environment objects necessitates a synchronization of the add and look calls. However, since an add call is non-blocking (following the original Linda model), an agent calling add has not to be blocked until the environment releases the lock. To this end, we have defined a buffer to which agents can add objects with-

\[\text{http://javacc.dev.java.net/}\]
out blocking, while emptying the buffer is synchronized with the look calls. On the other hand, 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.

Finally, 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.

7. APPLICATIONS

7.1 An Environment-Centered MAS for Traveler Information Systems

In this section, we describe an application based on LACIOS. We modeled and implemented a traveler information server [18], called ATIS\(^3\). The server purpose is to inform online travelers about the status of a part of the transport network that concerns them. Every traveler is mobile and has a specific objective during his connection on the server. Transport Web services are represented with agents in the server and their expertise domains constitute their properties. These transport service providers can give information in response to a request, or they may proactively send information concerning disturbances, accidents, events, etc. The problem in this kind of applications concerns the information flows that are dynamic and asynchronous. Indeed, each information source is hypothetically relevant. An agent cannot know a priori which information might interest him, since this depends on his own context, which changes during execution.

7.1.1 Context

Using LACIOS allows us to design an information server parameterized by its users. Indeed, if the information systems are adapted to the satisfaction of punctual needs (request/response), the management of the information followup assumes additional processing. These processing are difficult because the information sources are distributed and the management of the followup assumes a comparison of the available information.

We have defined two categories of agents, the first concerns the agents representing the users (that we call PTA for Personal Travel Agent) while the second concerns the agents representing the services (that we call Service Agent).

7.1.2 Technical Description

We have implemented a Web server for traveler information, where each Web service has a representant in the server, which is responsible of the convey of messages from the server to the port of the Web service and conversely. The exchange of messages between the server and the services are SOAP\(^4\) messages and the asynchronous communication is fulfilled via the JAXM API\(^5\) for the Web services supporting SOAP, and a FTP server otherwise, used as a sort of mailbox. These details are obviously transparent for the agents evolving in the environment i.e. they interact exactly the same way whatever the transport protocol that is used.

Every user is physically mobile and connects to the server via a Mobile Personal Transport Assistant (MPTA) and has during his connection a PTA agent representing him inside the server, which is his interlocutor during his session. The interaction of the MPTA with the PTA agents representing them is a sequence of HTTP requests/responses.

7.1.3 Execution Scenario

The context of the example is the following: inside the system, there is an agent representing a trip planning service and an agent representing a traffic service responsible of the emission of messages related to incidents, traffic jams, etc. These agents are persistent, since they are constantly in relation with the system providing the service. On the contrary, PTA agents representing the MPTA in the system are volatile, created on the connection of a user and erased at the end of his session i.e. when he arrives at destination. We have developed a trip planning service which role is to, first receive the trip planning demand (in the form of a SOAP message), then calculates the plan, wraps it in a SOAP message then sends it back to the local agent representing him in ATIS.

Every stop of the network is described by a line number line to which it belongs, and a number number reflecting his position on the line. A user u is also described by his current position in the network (the properties line and number). In a basic execution scenario, u has a path to follow during his trip i.e. a sequence of tuples \(\{(\text{line}, \text{number}_{\text{source}}, \text{number}_{\text{destination}}) \mid i \in I\}\), with I the number of transport means used by the traveler. Every tuple represents a part of the trip, without transfer. To receive his plan, the MPTA connects to the information server, and the agent u representing him is created. Then, the user is asked to specify his departure as well as his destination. Once this informa-

\(^3\)Agent Traveler Information Server


\(^5\)Java API for XML messaging, http://java.sun.com/webservices/jaxm/
tion entered, \( u \) adds his planning demand in the environment. A demand is an object described by his properties: \( \text{from, subject} \), etc. Afterwards, \( u \) keeps on listening to messages that are addressed to him, this way: \( \text{look}(\emptyset, \{ \text{message} \leftarrow x \}, x.\text{tor} = \text{id}) \). The agent representing the trip planning service is listening to messages asking for a plan: \( \text{look}(\emptyset, \{ \text{request} \leftarrow x \}, x.\text{subject} = \text{"plan"}) \). As soon as he receives the message, he creates a SOAP message addressed to the trip planning Web service and awaits for the response. When he receives the answer, a message is added to the environment addressed to \( u \) with the received plan as body: \( \text{add}(\{ \text{from} \leftarrow \text{id}, \text{to} \leftarrow \text{request.from, body} \leftarrow \text{plan}, \text{id} = \text{that.from, id} = \text{that.to}) \). The agent \( u \), when he receives the message, analyzes it and displays the result on the user’s MPTA. Then, the agent \( u \) restrains his interaction to the messages concerning events coming up on his way. To do so, he executes the following action:
\[
\text{look}(\emptyset, \{ \text{event} \leftarrow x \}, \{ x.\text{subject} = \text{"alert"}, x.\text{line} = \text{line} \land x.\text{number} \geq \text{number} \land x.\text{number} \leq \text{line} \})
\]

The agent \( u \) is interested by the alerts concerning his transport plan, which is expressed by the preceding \text{look} \ action. Let us assume that the agent representing the alert service adds an alert message concerning an accident on the way of \( u \) resulting on a serious delay for him. The traveler, via his representing agent \( u \), is notified concerning this alert event. Since the properties \( \text{line} \) and \( \text{number} \) are updated at each move of \( u \) (each time he moves from stop to stop), the segment concerned by the alert messages gets gradually reduced until the end of the trip. The use of the environment, the constant update of the properties of the PTA agents, together with the use of \text{look} \ actions allowed us to maintain a constant awareness of the traveler about problems occurring during his trip, without relying on continuous requests to the server.

### 7.2 Coordination Environment for Demand-Responsive Transporton Services

We have proposed a demand-responsive transport service (DRTS) as a MAS in which the agents’ activities are coordinated through the environment, based on LACIOS.

#### 7.2.1 Demand-Responsive Transport Services

A DRTS is a system designed to answer online customers that desire to be transported from one point in the network to another. Customers specify a time window associated with each point (departure and arrival) inside which they want to be visited. In our application the environment, via the use of LACIOS, structures the MAS components temporally and spatially, so that the interaction between agents is driven by their perception of it. The interaction between customers and vehicles is driven by their space-time positions, and the environment is modeled accordingly. A main issue in this application is the dynamics of the environment because when modeled as an MAS, DRTS are open MAS, where agents (e.g., customers and vehicles) join and leave the system freely. In such a dynamic environment, limiting communications is very important, since broadcasting all the available information is very costly. We use an implicit model on which LACIOS is grounded, in which communication is decoupled in space and time, so as to offset the loss in information in dynamic environments.

#### 7.2.2 System Description

We have designed a distributed model for a DRTS, in which two agent categories are defined: Vehicle Agents (VA) and Customer Agents (CA). Both VAs and CAs are generated dynamically: a new CA is associated to each new customer connected to the system, and a new VA is associated to each new vehicle creation (which occurs when no available vehicle can serve a new customer). The descriptions in this system are related to VAs and CAs (see figure 4). A VA is described symbolically by its current position and his remaining available seats. A CA is described by his departure and arrival nodes, his time windows, the vehicle \( \text{veh} \), and his successor (property \( \text{succ} \)) and predecessor (\( \text{pred} \)) customer in the route of \( \text{veh} \). A CA that doesn’t belong to any VA route has a property \( \text{veh} \) equal to \( \text{unknown}_{\text{veh}} \), and a property \( \text{succ} \) and \( \text{pred} \) equal to \( \text{unknown}_{\text{veh}} \).

In our application, the boolean expressions contained in the \text{look} \ actions are defined by VAs so that they will perceive only those customers they can insert in their route. The description of a new CA (with unknown \( \text{veh} \), \( \text{succ} \) and \( \text{pred} \)) is perceived when: (i) there are two nodes in the route of the same vehicle between which its departure node can be inserted without violating any of the time windows of the customers in that route, (ii) if there are remaining available seats when this insertion occurs, (iii) if there are two successor nodes in the route of this same vehicle between which the arrival node of the customer can be inserted, without violating any time window of the route. As a consequence, the use of LACIOS allows a new CA to discover the VAs that could be interested by its insertion, without knowing them in advance, while at the same time limiting communication in the system to only those agents that can reach an agreement (an insertion in the route of a vehicle). It is worth noting that the VAs that don’t perceive a CA can use their time to be candidates for the insertion of other customers.

The protocol followed in the MAS is a negotiation mechanism between CAs and VAs. When a new customer connects to the system, a CA is created (\( \text{spawn} \)), and is perceived by the available VAs with their current \text{look} \ actions (that is, which are not already involved in the insertion of another customer). The syntax of expressions that we introduced for LACIOS allows for the translation of the mathematical constraints on the insertion of a customer (vehicle capacity and space-time feasibility) onto LACIOS expressions, which is not possible for traditional Linda-like languages. Each VA com-
putes an insertion price for the insertion of this customer, and proposes it to the CA (add), which will choose the VA proposing the lowest price.

The interested reader about the complete definition of the agents actions and their expression parameters is invited to refer to [17].

In this application, the use of Lacios structured agent interaction and coordination, and made it more efficient to interact in a dynamic environment where agents appear and disappear without maintaining knowledge about the others and where communications can be disturbed and costly.

8. CONCLUSION

When designing transportation systems, we have observed several recurrent issues that call for new models and languages for their design, management and implementation. Although multiagent languages facilitate the design of such applications, they are not efficient with respect to communication costs in open and highly dynamic applications. In this paper, we suggest that using the environment as a common coordination medium is an efficient solution. We have defined the language LACIOS, which is a Linda-like language with distinguishing features making it easier to design and implement open highly dynamic multiagent transportation applications.

The multiagent environment might be distributed over several hosts, but until now, this was done in an ad hoc way aiming at reducing communication costs between the different hosts. We are currently working on automatic distribution of the environment, following the specification of entities’ properties, and based on their clustering in the form of Galois lattices.

9. REFERENCES


