A data-oriented coordination language for distributed transportation applications

Mahdi Zargayouna[1], Flavien Balbo[1,2], Gérard Scémama[1]

[1] Gretia laboratory, The French National Institute for Transport and Safety Research
Site de Marne-la-Valle "Le Descartes 2"
2 rue de la Butte Verte
93166 Noisy le Grand Cedex, France
[2] CNRS-Lamsade laboratory, University of Paris Dauphine.
Place Marechal de Lattre de Tassigny,
75775 Paris Cedex 16, France.
{zargayouna,scemama}@inrets.fr, balbo@lamsade.dauphine.fr

Abstract. Data-oriented coordination languages rely on a shared space in which agents add, read and retrieve data. They are intuitively well suited 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 cannot express agents complex interaction needs. Indeed, in transportation applications, the interaction needs of the agents are driven by their current context and by *ambient* conditions, expressed in the form of constraints on the values taken by variables. In this paper, we propose LACIOS, a new data-oriented coordination language for designing and implementing distributed transportation applications, and we illustrate our proposal with two examples: a traveler information system and a demand-responsive transport system. LACIOS allows agents to express complex interaction needs, including agents states, system objects values, operators and functions, and is grounded on a model where the multi-agent system design is centered on the environment.

1 Introduction

The transportation domain is a privileged field for the multi-agent community. Indeed, it exhibits characteristics that makes it relevant to model transportation applications in the form of many physically and/or logically distributed interacting components that possess some level of autonomy. In [1], we have identified three recurrent issues when designing and implementing transportation applications: the knowledge processing, the space-time dimension of the problems and the dynamics of the real environment, and we have argued that the design of an environment-centered multi-agent system (MAS) is a solution to these issues. The multi-agent environment contains the recorded descriptions and supports their processing. In this paper, we propose a language to specify environment-centered multi-agent systems and we describe two applications designed and implemented with this language.

Our language is a data-oriented language, which were initiated by Linda [4], and has known several extensions (e.g. Klaim [3], Mars [2] and Lime [6]). Linda-like models are based on the notion of a shared data repository. 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.

LACIOS is a data-oriented coordination language designed to overcome several limitations in existing languages of the literature, and designed to facilitate their use in transportation applications. In this paper, we focus on two limitations: the poor expressive power of templates and the inaccessibility of agent states. In the literature, agents are black boxes: they don't have an observable state described by data. Indeed, data-oriented models describe what the agents *do*, not what they *are*. In the absence of agents' states, agents cannot condition their interaction with their current context, especially in the absence of a rich interaction mechanism.

This paper is organized as follows. Section 2 describes the LACIOS language that we propose and details its syntax. In section 3, we present the traveler information system based on LACIOS. Section 4 presents our proposal of a coordination environment for a demand-responsive transport service. Finally, section 5 concludes the paper.

2 The coordination language Lacios

2.1 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 a set of operators borrowed from Milner's CCS [5]. A 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*. Agents can *perceive* (read only) and/or *receive* (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 receive 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 [7].

2.2 Syntax

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*$\leftarrow$*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 1. 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. $\exists i \in \{1,\ldots,nbt\}$, $type_i = \{true, false, nil\}$

Definition 2. 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* $type(\pi) \in \{type_1, \ldots, type_{nbt}\}$.

A description is composed of properties and their corresponding values.

Definition 3. 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 every description, the set of properties* $\{\pi \mid d(\pi) \neq nil\}$ *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 ($\mathcal{O}$ is the set of objects), while an agent is defined by a description and a behavior ($\mathcal{A}$ is the set of agents).

Definition 4. 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. $\exists i \in \{1,\ldots,nbt\}$, $type_i = \Omega \cup \{nil\}$.

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

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

Definition 5. Operators *Each operator op of the language is defined by:*

- *(i)* $arity(op)$ *The number of parameters of the operator,*
- *(ii)* $par(op) : \{1, \ldots, arity(op)\} \rightarrow \{1, \ldots, nbt\}$*,* $par(op)(i)$ *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 $ret(\texttt{and}) = 1$.

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, it designates the destination of a. 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 6. Expressions *Exp is the set of expressions. An expression* $e \in Exp$ *is generated via the grammar of table 1.*

$e ::= nil$	
$\mid v$	, 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 1. Syntax of an expression

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

Matching Since we consider a data structure richer than tuples, we also use a matching mechanism richer than templates. The matching in LACIOS materializes what we call a *contextual interaction*, which is the type of interactions that use the state of the agent and the state of objects in the environment to access a set of objects, instead of only one like in Linda 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.

Definition 8. 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 expression:
$t.destination =$ "$Uppsala$" $\wedge\, t.price \leq budget \wedge p.decision =$ "$accepted$"
In this expression, t and p designate two objects, unknown for the moment, where t has to have as *destination* "Uppsala" and a *price* less than the *budget* of the agent, and where p must have as *decision* equal to "accepted" for the expression to be evaluated to *true*.

Agents' actions We define three primitives for LACIOS, two for the interaction with the environment (*add* and *look*) and one for agents' creation (*spawn*).

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

The primitive $spawn(P, sds)$ launches a new agent that behaves like the process P and that has as a description the result of the evaluation of sds (its transformation to a description ds). The primitive $add(sds)$ adds to Ω_{ENV} an object described by the evaluation of sds.

We choose to use a single primitive to access the environment. The primitive $look(sds_p, sds_r, e)$ allows both object perception and reception (perception and removal from Ω_{ENV}). It blocks until a set of objects becomes present in Ω_{ENV} such that the expression e is evaluated to *true*; the objects associated with the variables in sds_p are perceived and those associated with the variables in sds_r are received. For instance, the following instruction: $look(\{ticket \leftarrow t\}, \{paper \leftarrow p\},$ $t.destination =$ "$Uppsala$" $\wedge\, t.price \leq\ budget \wedge p.decision =$ "$accepted$") looks for two objects that will be associated with t and p. The object associated with t will be perceived while the object associated with p will be received. After the execution of this instruction, the two objects will be present in the local memory of the caller agent, the latter 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.

3 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 [8], 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.

3.1 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).

3.2 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

[3] Agent Traveler Information Server
[4] Simple Object Access Protocol, http://www.w3.org/TR/soap/
[5] Java API for XML messaging, http://java.sun.com/webservices/jaxm/

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.

3.3 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 $\{(line, number_{source}, number_{destination})_i \mid i \in I\}$, 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 these information entered, u adds his planning demand in the environment. A demand is an object described by his properties: *emitter*, *subject*, etc. Afterwards, u keeps on listening to messages that are addressed to him, this way: $look(\emptyset, \{message \leftarrow x\}, x.receiver = id)$. The agent representing the trip planning service is listening to messages asking for a plan: $look(\emptyset, \{request \leftarrow x\}, x.subject = \text{“}plan\text{”})$. 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: $add(\{emitter \leftarrow id, receiver \leftarrow request.emitter, body \leftarrow plan\})$. 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 :

$look(\emptyset, \{event \leftarrow x\}, \{x.subject = \text{“}alert\text{”}\}, x.line = line \wedge x.number \geq number \wedge x.number \leq line)$

The agent u is interested by the alerts concerning his transport plan, which is expressed by the preceding *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 *line* and *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 *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.

4 Coordination environment for demand-responsive transportation systems

We have proposed a demand-responsive transportation system (DRTS) as a MAS in which the agents' activities are coordinated through the environment, based on Lacios.

4.1 Demand-Responsive Transportation Systems

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.

4.2 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 1). A VA is described symbolically by his current position and his remaining available seats. A CA is described by his departure and arrival nodes, his time windows, the vehicle *veh*, and his successor (property *succ*) and predecessor (*pred*) customer in the route of *veh*. A CA that doesn't belong to any VA route has a property *veh* equal to $unknown_{veh}$, and a property *succ* and *pred* equal to $unknown_{cust}$.

In our application the environment structures the MAS components temporally and spatially, so that the interaction between agents is driven by their

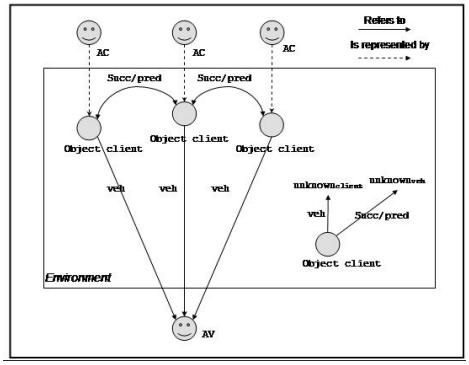

Fig. 1. Symbolic descriptions

perception of it. The boolean expressions contained in their *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 *veh*, *succ* and *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 (*spawn*), and is perceived by the available VAs with their current *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 computes 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 [7].

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.

5 Conclusion

In this paper, we have described the main part of the LACIOS coordination language while the complete definition can be found in [7]. The relevance of LACIOS for transportation applications is assessed by two applications: a traveler information server and a demand-responsive transport service.

In [7], we have implemented a script language on top of Java for the implementation of MAS in LACIOS (called Java-LACIOS). It consists of a compiler of a program written in LACIOS (text file) which generates a Java program. Therefore, system designers and programmers do not have to worry about the creation and synchronization of threads, or the matching process, while building their MAS. This new programming language makes it easier to build environment-centered applications, and is suitable for transportation applications, where the interaction needs of the agents (often based on mathematical constraints) are complex and contextual.

The multi-agent environment might be distributed over several hosts, but until now, this was done in an *ad hoc* way aiming at reducing communication cots 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.

References

1. F. Badeig, F. Balbo, G. Scémama, and M. Zargayouna. Agent-based coordination model for designing transportation applications. In *Proceedings of the 11th International IEEE Conference on Intelligent Transportation Systems*. IEEE Computer Society, 2008.
2. G. Cabri, L. Leonardi, and F. Zambonelli. Reactive tuple spaces for mobile agent coordination. In *MA'98: Proceedings of the Second International Workshop on Mobile Agents*, pages 237–248, London (UK), 1999. Springer-Verlag.
3. R. De Nicola, G. L. Ferrari, and R. Pugliese. Klaim: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
4. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
5. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989. 272 pages.
6. G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda meets mobility. In *International Conference on Software Engineering*, pages 368–377, 1999.
7. M. Zargayouna. *Modèle et langage de coordination pour les systèmes multi-agents ouverts. Application au problème du transport à la demande*. PhD dissertation, University of Paris-Dauphine, Paris (France), 2007. In French.
8. M. Zargayouna, F. Balbo, and J. Saunier. Agent information server: a middleware for traveler information. In O. Dikenelli, M.-P. Gleizes, and A. Ricci, editors, *Engineering Societies in the Agents World VI*, volume 3963 of *LNAI*, pages 3–16. Springer-Verlag, 2006.