An Experience in Using Components to Construct and Compose Agent Behaviors for Agent-based Simulations

Jean-Pierre Briot
Laboratoire d'Informatique de Paris 6 (LIP6)
Université Paris 6 - CNRS
Case 169, 4 place Jussieu, 75252 Paris Cedex 05, France
& CS Dept., PUC-Rio, Rio de Janeiro, Brazil
Email: Jean-Pierre.Briot@lip6.fr

Thomas Meurisse
Laboratoire d'Informatique de Paris 6 (LIP6)
Université Paris 6 - CNRS
Case 169, 4 place Jussieu, 75252 Paris Cedex 05, France
Email: Thomas.Meurisse@lip6.fr

Abstract— **This paper summarizes our experience in using a component model to help at construction of agents for agent-based simulations. In this model, named MALEVA, components encapsulate various units of agent behaviors or activities (e.g., follow gradient, flee, mate, reproduce). Among its specificities, it extends the principles of software composition to the specification of control, through the notions of control ports and of control components. Moreover, a notion of composite component allows complex behaviors to be constructed from simpler ones. Two examples, an ecosystem of situated agents and a microsimulation, are presented. We also discuss the benefits of our model for a fine grain control of activation and scheduling.**

Keywords— **agent, component model, behavior, composition, activation control.**

I. Introduction

Agent-based simulation is now established as one of the main approach for modeling and simulation of various phenomena. We believe that this is because the concept of agent is both structuring enough (unit of activity, of interaction, concept of organization...) and versatile enough (reactive or cognitive agents, various models of environments...).

An important issue is how to design and construct operational models of agents. Using software engineering principles should help at genericity and reuse of the models. A natural direction is thus to exploit the concepts of software components which already proved to be an effective approach for rationalizing composition, reuse, and deployment of software. It is important to note that, for simulation applications, the designers are not necessarily themselves expert programmers. Moreover, they usually want to quickly prototype and then update the behavioral properties of the various agents populating a simulation. We will see at Section IV how components, and the separate specification of their control, may help them.

In this paper, we relate on our experience in the design and the use of a component model for constructing agents for agent-based simulations. This component model, named MALEVA, helps at an incremental construction of an agent by composition of simple agent behaviors or activities (e.g., flee, follow gradient, mate, reproduce).[1] One of its specificities is that it extends the principles of software composition to the specification of control, through the notions of control ports and of control components. MALEVA has indeed been used as the foundation or as an inspiration for various agent-based simulation projects, applied to, e.g.: urban migration [22], traffic simulation [16], and fish tanks evolution.

Two examples will be described in the paper, for two different types of agent-based simulation. The first example is a variant of the classical prey/predator simulation example. The second one is a simplification of a real application, on micro-simulation of population evolution [14]. These examples illustrate how MALEVA can offer some potential for reuse and specialisation, through: structural composition of behaviors and specialisation of intra-agent scheduling policies.

II. The MALEVA Agent Component Model

The objective of the MALEVA component model is to help at incremental construction of agent behaviors (e.g., flee, follow gradient, reproduce), reified as software components. Therefore, we assume that there is a library of behavior components associated to the application domains targeted.

A component may be primitive (it is written in the underlying language, e.g., Java), or *composite* (i.e. defined as the encapsulation of a composition/assemblage of components).[2]

A. Data Flow and Control Flow

In MALEVA, a distinction is made between the activation control flow and the data flow connecting the components. As

[1]In this work and this paper, we focus on the issue of components at the *agent level*, to decompose the internal structure of one given agent. Multi-agent platforms often focus on the use of components at the *system level* (an agent is implemented as a component) as, e.g., in [17].

[2]The notion of composite component corresponds to a notion of *structural composition*, as opposed to, or rather *in addition to*, *functional composition* (simple assemblage). Architectures of sub-components may be encapsulated in composites, thus providing a hierarchical form of composition. A composite may also provide extra functionalities (and control specifications) at its higher abstraction level, making it a true component on its own. Another example of component model providing a notion of composite is the Fractal component model [6].

	Data	*Control*
Input port	Data consumption	Activation entry point
Output port	Data production	Activation exit point
Connexion	Data transfer	Activation transfer

TABLE I
DATA AND CONTROL PORTS

we will show in Section II-B, this characteristic and likely specificity of our model, decouples the functional architecture from the activation control architecture. The objective is to make components more independent of their activation logic and thus more reusable.

Consequently, we consider two different kinds of ports within a component:

- *data ports*. They are used to convey data transfer (one way) between components.[3]
- *control ports*. A behavior encapsulated in a component is activated only when it explicitly receives an activation signal through its input control port. When the execution of the behavior is completed, the activation signal is transferred to its output control port.

As shown in Table I, in addition to the specific *semantic* distinction between data ports and control ports, MALEVA adopts the common *architectural* distinction between input ports and output ports, as in, e.g., UML2 or CCM [19].

B. An Introductory Example

Figure 1 shows a first and very simple example of assembly/composition of components: a sequence of two components. Component B is activated after the computation of component A completes. Regarding data, component B will consume the data produced by component A only after computation of A completes. In this figure, as well as the following ones, data flow connexions are shown in solid lines, and control flow connexions in dotted lines.

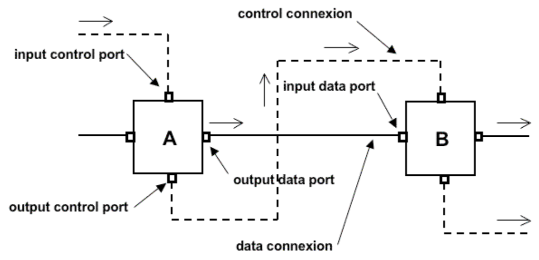

Fig. 1. Sequential activation of two components

Figure 2 recombines the two same components, but this time activated concurrently.[4] This simple example is a first illustration of the possibilities and flexibility in controlling activation of components. One may describe active autonomous components (with an associated thread), explicit sequencing or any other form of combination. Flow of control is specified outside of the components, which provides more genericity on the use of components,[5] and, as we will discuss in Section IV, also a fine grained control over activation policies.

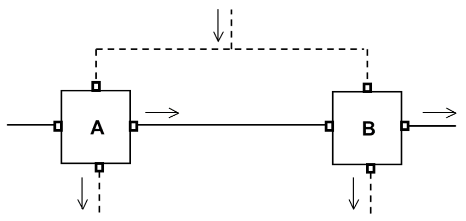

Fig. 2. Concurrent activation of two components

III. AN EXAMPLE OF SITUATED AGENTS

Our first example will define behaviors of situated agents within an ecosystem. Thus, the first step is to design a general architecture for situated agents.[6]

A. Abstract Architecture of a Situated Agent

A *situated agent* senses its environment (e.g., position of the various agents near by, presence of obstacles, presence of pheromones...) through its sensors. These data are used by its (internal) behavior to produce data for its effectors, which will act upon the environment (e.g., move, take food, leave a pheromone, die...). The general architecture of a situated agent usually follows the computational cycle:

$$\textit{sensors} \rightarrow \textit{behaviour} \rightarrow \textit{effectors}$$

and is shown at Figure 3.

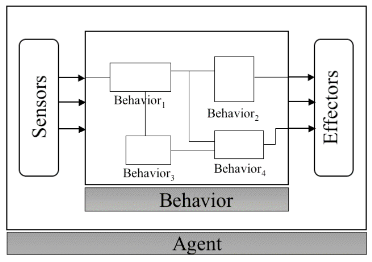

Fig. 3. Abstract architecture of a situated agent

B. Basic Behaviors

We will now define and construct the behaviors of preys and predators agents. By following a bottom-up approach, we first define a set of elementary components, representing the basic

[3]Data ports are typed, as further discussed in Section V-C.

[4]The control connexions have been changed accordingly, but not the data connexions. The semantic is analog to the *pipes and filters* [20] architectural style: component B consumes what A produces while they are both active simultaneously.

[5]An additional dimension, the mode of activation of components, which could be asynchronous or synchronous, will be addressed in Section V-B.

[6]For other applications, e.g., in Section IV on micro-simulation, agents are not necessarily situated (within an environment) and thus do not use any sensor/effector.

behaviors of preys and predators, that we name: `Flee` (fleeing a predator) ; `Follow` (following a prey) ; and `Exploration` (exploration through a random move, which represents the default behavior). Then we compose them, to represent the following agent behaviors: `Prey` and `Predator`.

C. Control Components

A prey flees the predators being located within its field of perception. If no predator is close (sensed), the prey explores its surroundings by moving randomly. Thus, we construct the `Prey` behavior as the composition of the following three components: `Flee`, `Exploration`, and a control component named `Switch`.[7]

The `Switch` control component reifies the standard conditional structure into a special kind of primitive component. The condition is the presence or absence of an input data. The behavior of `Switch`, once being activated (receiving an activation signal), is as follows:

```
IF data is received through If (input data port)
THEN transfer control through Then (output control port)
AND send data through Then (output data port)
ELSE transfer control through Else (output control port)
```

D. Prey Behavior

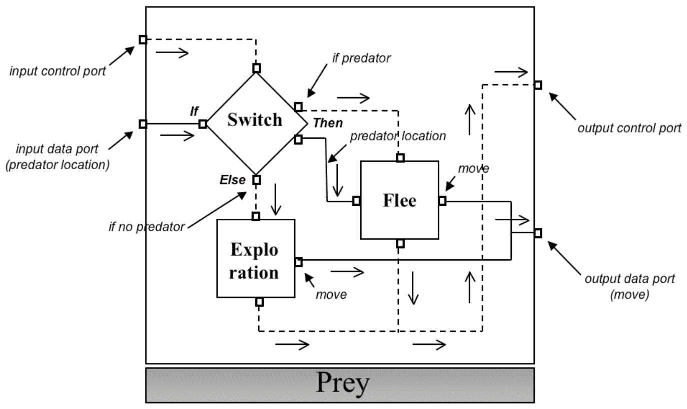

Fig. 4. Architecture of a prey

The behavior of a prey, named `Prey`, is defined as follows. If it detects a predator (some data representing the predator location has been received on its input data port), the `Switch` component transfers the control through its `Then` output control port, which activates the `Flee` behavior. Then `Flee` can compute a move data based on the location of the predator, and send it through its output data port. The move data arrives to `Prey` output data port and then to the effector, to produce a move of the agent on the environment. If no predator has been detected, `Switch` transfers control through its `Else` output control port, which activates `Exploration` behavior.[8] The result is shown at Figure 4.[9]

E. Predator Behavior

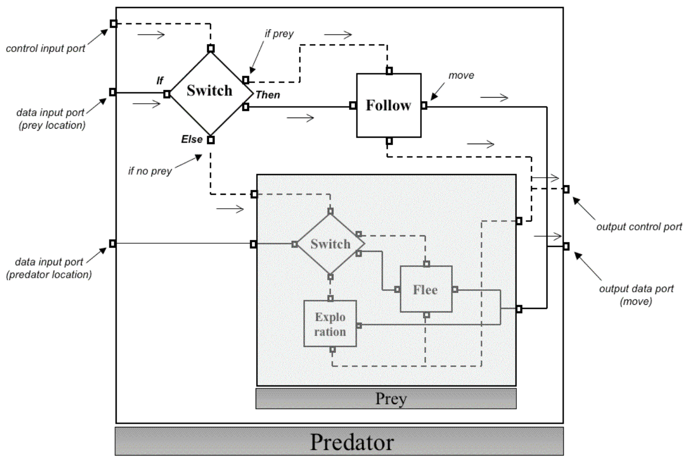

Fig. 5. Architecture of a predator (with prey as a sub-component)

We may now reuse the `Prey` behavior component to construct the behavior of a predator, which follows the preys while fleeing his fellows predators, and otherwise explores its surroundings. The behavior of a predator may be defined as the behavior of a prey (it flees other predators and otherwise carries out an exploration movement), to which is added a behavior of predation (it follows preys that he could perceive). According to our compositional approach, we define the `Predator` behavior component as a new composite behavior embedding – *as it is* – the existing `Prey` behavior component (see the result in Figure 5).[10]

An alternative to a bottom-up approach is a top-down approach. An example is the top-down design of the behavior of an ant, which involves a larger amount of components and also some design patterns. Because of space limitation, it is not included in this paper but is detailed in [5].

IV. An Example of Population Microsimulation

This second example will help at illustrate another merit of making explicit the control flow. It is inspired from an existing application of demography microsimulation (named Destinie) [14] and conducted at the French national statistics institute (INSEE). In this (simplified) example, we consider a virtual specie of agents, in which mating of two agents is necessary for reproduction, but without considering sexual differences (i.e. agents are hermaphrodite). We consider three basic behaviors: `Mate`, `Separate` and `Reproduce`. Behaviors may be seen as state changes, governed by probabilistic

[7]The MALEVA standard library includes other control components, analog to standard control structures (e.g., repeat loop) or synchronisation operators (e.g., barrier synchronisation), see [18].

[8]`Exploration` does not need a data input to produce a move data.

[9]We assume the input data port (perception of a predator in the environment) and the output data port of the `Prey` behavior to be connected to the corresponding sensor and effector data ports, along the general architecture of a situated agent, shown at Figure 3.

[10]In this design, hunger (predation) has priority over fear (fleeing), as `Prey` is activated by `Predator`. Other combinations could be possible.

transition laws. To activate a behavior evaluates if there is a state change, according to the associated probability. Although not independent, these 3 behaviors are not necessarily bound to a specific sequence of activations, as all possible interleavings are valid.

In general [11], behaviors are ordered sequentially, as proposed by domain experts. Indeed, it is not easy to realize a priori the impact of the possible interleavings. But the issues are: In what order ? And with what impact on the simulation results ?

Let us consider the following combinations:

(a): { `Mate ; Separate ; Reproduce` }
(b): { `Mate ; Reproduce ; Separate` }
(c): { `Mate || Separate || Reproduce` }

The two first strategies *(a)* and *(b)* make explicit an order of activation (sequence) of behaviors. In the third *(c)* strategy, no temporal dependency constraint is specified (concurrency), leaving the scheduler of the runtime system free of actual scheduling decision.[11]

Figure 6 shows the resulting histogram. It displays the number of individuals/parents (*y* coord.) having a certain number of children (*x* coord.). Results follow the intuition: if reproduction is activated before separation *(b)*, this leads to more children than if reproduction is activated after separation *(a)*. A fully concurrent strategy *(c)* produces an average number.

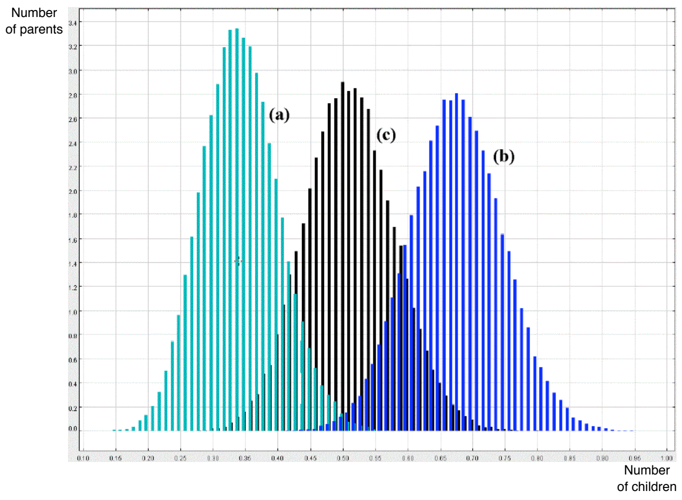

Fig. 6. Histogram of number of babies

This example shows the importance for the simulation designers to be able to experiment with various strategies for ordering behaviors, to compare results with the target models, and to quantify the impact on biases.[12] By considering control flow explicitly, MALEVA helps at specifying and controlling temporal dependencies between behaviors, and thus their possible orderings. This is realized via explicit control flow connexions, without any change to the code of behaviors (encapsulation is ensured), as shown at Figure 7.[13] This is notably useful for simulation applications, where the expert may incrementally specify temporal dependencies, independently of behaviors functionalities, and thus experiment with various strategies, compare results with the target models, and quantify the impact on biases [18]. A methodological guideline for multi-agent-based simulations, and how MALEVA may help in the incremental refinement from a design model to an operational model, are further discussed in [4].

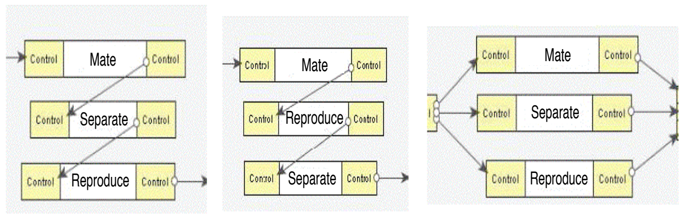

Fig. 7. Temporal dependency specifications with CGraphGen

V. Tools and Implementation

The MALEVA prototype CASE tool includes a library of components (behavioral components and control components) ; an editor of connexion graphs (named CGraphGen, which stands for *concurrent graph generation*) ; a graphical environment for constructing virtual environments for situated agents ; and a run time support for scheduling and activating components.

A. From Methods to Components

An interesting feature of CGraphGen is the importation of Java code and its reification into MALEVA components. The granularity considered is a Java method. After specifying the class, method name, and its signature, CGraphGen automatically generates a corresponding component whose data ports correspond to the method signature: one input data port for each parameter, and one optional output data port for the result (none in the case of `void`). Two control ports (one input and one output) are also implicitly added. CGraphGen allows graphical connexion of data-flow and control-flow between components, and the creation of composite components.

Note that XML-based descriptions may be used for importation or exportation. These simple characteristics (reification of existing code into components, manipulation of the temporal dependencies) turned out to be quite useful to help at reengineer existing simulation applications, specially when considering that the experts of the domains modelled and simulated are seldom programmer experts.

[11]To be more precise, the MALEVA runtime scheduler tries, for each simulation step, to maximise possible interleavings of behaviors activations.

[12]For instance, [15] show that results of simulations can be found biased in cases where the scheduling of the actions within an agent remains deterministic.

[13]Connexions are usually achieved interactively, see Section V-A. In this example, there are no data ports shown, because the behaviors considered in this case study are very simple, thus without parameters, nor results.

B. *Modes of Activation and Scheduling*

At the level of the general scheduler, two alternative modes (or strategies) of activation have been implemented: an *asynchronous* mode and a *synchronous* mode. In the *asynchronous mode*, the different agents (and components) evolve independently. It may be more efficient, specially in the case of distributed implementation. Meanwhile, unless the designer also uses explicit control connexions between agents, the different agents may not be synchronised (some can compute ahead of others), depending on their relative processing speed. In the *synchronous mode*, the scheduler sends next activation trigger once all behaviors have finished, which ensures but also imposes a global synchronization. The choice between the two modes depends on the requirements for the application (see, e.g., [15] and [18] for more discussion).

C. *Implementation*

After the initial Delphi implementation, the Java-based re-implementation of MALEVA[14] added typing to the components ports and connexions. This turned out to be useful for verifying interface compatibility between components. In addition, sub-typing helps at defining more abstract components. Java also supports inspecting various information about a component, thanks to its introspection facilities (API and tools). Thus, the designer can easily query a component to obtain its internal information. The Java implementation also improved the possibility of architectural dynamic evolution, which turned out to be useful to model evolving behaviors, such as, e.g., ant metamorphosis (see Section VII).

The Java implementation, actually based on JavaBeans, also gave opportunity to compare our MALEVA prototype component model with an industrial component model. Note that the JavaBeans model conforms to a publish/subscribe communication model, *but* the implementation still relies on standard method call. In our implementation of MALEVA, a mailbox (FIFO queue of messages) is associated to each input data port and to the input control port, in order to decouple data transfer and actual activation.

D. *Performances*

Considering performance, it is obvious that a very fine grained decomposition of agent behaviors will have a cost. This brings us to the usual trade-off between genericity and performance. However, it is important to note that the exact decomposition granularity may be freely chosen by the application designer and locally for each agent. We are considering the possibility of providing a code generator for transformation (compacting) of a composite component into an atomic component, with additional optimisations (e.g., transform certain activations in synchronous method calls when intra-agent concurrency is not used).

[14]A re-implementation of MALEVA into C++ has also been realized, in order to conduct more efficiently (reduce computation time) large scale experiments [18]. Last, a complete re-implementation of Maleva in Smalltalk (Squeak), named MalevaST, has very recently been independently conducted by Noury Bouraqadi et al. at École des Mines de Douai, France.

VI. Related Work

CCM (Corba Component Model) [19] is an industrial general model of component, supporting input and output interfaces and also event-based communication. However it does not support a notion of composite. Also, its development cycle is portable but relatively complex.

The Fractal model of component [6] supports the notion of composite. A concept of controller is also supported but it mainly offers simple control interfaces for life cycle management or for structural reconfiguration. Last, the control flow is not explicitly specified. Note that MALEVA could benefit from Fractal introspection and dynamic reconfiguration capabilities.

DEVS (Discrete Event Simulation Formalism) [23] is a formalism to model simulations in a hierarchical and modular way. DEVS is based on an atomic model based on timed state transitions and a coupled model to construct complex models in a hierarchical fashion. Compared to DEVS, MALEVA focuses on a model of component without imposing a specific (powerful but also complex) formalism for describing computation. Also, note that DEVS modules are not usually implemented as software components (no explicit output interface/ports nor connectors), although there are some steps in that direction (see, e.g., some work about mapping DEVS onto the Microsoft COM component model [7]).

The JADE architecture [2] offers some basic support for the designer to construct an agent as a set of *behaviors* (instances of class `Behaviour`). Some subclasses, e.g., `CompositeBehaviour` and `ParallelBehaviour`, provide basic structures for constructing hierarchies of behaviors or/and for expressing control structures. For instance, a more advanced one, `FSMBehaviour`, relies on a finite state automaton. Meanwhile, JADE behaviors are not real components (no output interface/ports nor connectors), thus the architecture of an agent is still partly hidden within the code.

Like MALEVA, JAF (Java Agent Framework [13]), also based on JavaBeans, uses components to decompose behaviors of agents. JAF does not explicitly separate control flow from data flow. But it proposes some interesting match-making mechanism, where each component specifies the services that it requires. At component instantiation time, JAF looks for the best correspondence between the requirements specification and the components available. Another difference between JAF and MALEVA is at the level of behavior decomposition. JAF decomposition appears at a relatively high level, whereas MALEVA promotes a fine grain behavior decomposition, and its management through explicit control.

The MaSE methodology [8] includes a modular representation of agent behaviors as sets of concurrent tasks. Each task is described as a finite state automaton and implemented as an object with a separate thread. A task can communicate with other tasks, inside the same agent, or with another agent task, through event communication. The implementation of MaSE concurrent tasks does not use components with explicit input/output ports. Also, MALEVA provides more explicit control of activation, whereas MaSE concurrent tasks partly rely on implicit control (inter-tasks implicit concurrency and synchronous message reception). That said, the MaSE methodology is actually quite general and we may imagine using some of MaSE steps to design MALEVA components.

Because of space limitation, we also refer to [5] for a more extended comparison of various rationales and architectural styles for modular agent architectures, and to [3], for a general survey on architectures and languages for multi-agent systems.

VII. Conclusion

In this paper, we presented a component model, named MALEVA, to construct agents for agent-based simulations. This model is relatively original in the explicit management of activation through control ports and connexions, by applying the concept of component to the specification of control. It is behavior-oriented and supports composite components. Some experiments have illustrated how MALEVA may help at genericity and reuse of components, through: structural composition of behaviours, abstract components and design mini-patterns (as discussed in [5]), and at rationalizing control of intra-agent behavior scheduling, an important issue for simulation. MALEVA has been experimented in different application domains, such as: urban migration [22], automobile trafic simulation [16], and artificial societies, e.g., a reengineering [18] of Sugarscape model [9].

An issue is in providing rich libraries of components and abstract architectures, supporting the types of architectures and applications targeted, for agent-based simulation, but also in other application fields, as we believe that the MALEVA model of component has a wider potential scope.

Another issue is that, in case of large applications, the connexion graphs may become large, *although* they may be hierarchical and encapsulated in composite components. Some radical alternative approach to reduce the control graph complexity is to abstract it in an adequate formalism such as a process algebra, in order to allow the concise representation of complex activation patterns. Such formal characterization would also allow the semantic analysis of such specifications, e.g., through model checking (see more discussion in [5]).

A last issue is the possible dynamicity of behaviors, e.g., to model the metamorphosis of an ant: from an egg, to a larva, to a worker ant or to a queen [5]. Current implementation strategy relies on a specific meta-component to manage the reconfiguration and re-assemblage of behaviors. We are currently considering using a higher level mechanism, based on concepts of configurations, roles and policies, such as the MaDcAr prototype model of automatic reconfiguration [12].

In summary, we hope that this short presentation of the MALEVA component model has illustrated some of its specificities and abilities at composing and reusing agent behaviors, for agent-based simulation applications. More generally speaking, we believe that some features of our component model may be transposed, and that making control available at the composition level may help the use of components within frameworks of applications vaster than those in which they had been initially thought.

Acknowledgements: to Marc Lhuillier, Alexandre Guillemet, Grégory Haïk, and Frédéric Peschanski, for their contributions to the MALEVA project.

References

[1] R. Axtell, Effects of Interaction Topology and Activation Regime in Several Multi-Agent Systems, *Int. Workshop on Multi-Agent-Based Simulation (MABS'2000)*, No 1979, LNCS, Springer, 2000, pp. 33–48.

[2] F. Bellifemine, A. Poggi and G. Rimassa, Developing Multi-Agent Systems with a FIPA-compliant Agent Framework, *Software Practice and Experience*, (31):103–128, 2001.

[3] R. Bordini, L. Braubach, M. Dastani, A. El Fallah Seghrouchni, J.J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci, A Survey of Programming Languages and Platforms for Multi-Agent Systems, *Informatica*, (30):33–44, 2006.

[4] J.-P. Briot and T. Meurisse, A Component-based Model of Agent Behaviors for Multi-Agent-Based Simulations, *AAMAS'06 7th Int. Workshop on Multi-Agent-Based Simulation (MABS'06)*, Hakodate, Japan, May 2006, pp. 183–190.

[5] J.-P. Briot, T. Meurisse and F. Peschanski, Architectural Design of Component-based Agents: a Behavior-based Approach, *AAMAS'06 4th Int. Workshop on Programming Multi-Agent Systems (ProMAS'06)*, Hakodate, Japan, May 2006, pp. 35–49.

[6] E. Bruneton, T. Coupaye, M. Leclerc, V. Quema, and J.-B. Stefani, An Open Component Model and its Support in Java, *7th Int. Symposium on Component-Based Software Engineering*, No 3054, LNCS, Springer, May 2004, pp. 7–22.

[7] Y.I. Cho and T.G. Kim, DEVS Framework for Component-based Modeling/Simulation of Discrete Event Systems, *The 2002 Summer Computer Simulation Conference*, San Diego, CA, USA, July 2002.

[8] S.A. DeLoach, Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems, *Agent-Oriented Information Systems (AOIS'99)*, Seattle, WA, USA, May 1999.

[9] J. M. Epstein and R. L. Axtell, *Growing Artificial Societies: Social Science from the Bottom Up*, MIT Press, 1996.

[10] P. Fishwick, *Simulation Model Design and Execution*, Prentice-Hall, 1995.

[11] N. Gilbert and K.G. Troitzsch, *Simulation for the Social Scientist*, Open University Press, 1999.

[12] G. Grondin, N. Bouraqadi, and L. Vercouter, MaDcAr: an Abstract Model for Dynamic and Automatic (Re-)Assembling of Component-Based Applications, *9th Int. SIGSOFT Symposium on Component-Based Software Engineering (CBSE'2006)*, No 4063, LNCS, Springer, 2006, pp. 360–367.

[13] B.C. Horling, A Reusable Component Architecture for Agent Construction, *Technical Report* No 1998-49, Computer Science Dept., UMASS, MA, USA, October 1998.

[14] INSEE, Le modèle de microsimulation dynamique DESTINIE, *Technical Report* G9913, Division Redistribution et Politiques Sociales, Institut National de la Statistique et des Études Économiques, Paris, France, 1999.

[15] B. G. Lawson and S. Park, Asynchronous Time Evolution in an Artificial Society Mode, *Journal of Artificial Societies and Social Simulation*, 3(1), 2000.

[16] V. LeCerf and M. Pintado, An Adaptive Model of Camera-Driven Urban Intersections Observation, *Workshop on Dynamic Scene Recognition from Sensor Data*, edited by C. Tessier, ONERA, Toulouse, France, June 1997.

[17] F. Melo, R. Choren, R. Cerqueira, C. Lucena, and M. Blois, Deploying Agents with the CORBA Component Model, *2nd Int. Conference on Component Deployment (CD'2004)*, No 3083, LNCS, Springer, May 2004, pp. 234–247.

[18] T. Meurisse, Simulation multi-agent : du modèle à l'opérationnalisation, *Thèse de doctorat (PhD thesis)*, Université Paris 6, France, July 2004.

[19] OMG (Object Management Group), Corba Component Model (CCM), 2006. "`http://www.omg.org/technology/documents/formal/components.htm`".

[20] M. Shaw and D. Garlan, *Software Architectures: Perspective on an Emerging Discipline*, Prentice Hall, 1996.

[21] K.G. Troitzsch, Methods of empirical social research, *SICSS Summer School*, 2000.

[22] D. Vanbergue, J.-P. Treuil, and A. Drogoul, Modelling Urban Phenomena with Cellular Automata, *Advances in Complex Systems*, Vol. 3, 2000.

[23] B.P. Zeigler, Discrete Event Simulation Formalism for Model based Distributed Simulation, *1985 SCS MultiConference: Distributed Simulation*, San Diego, CA, USA, January 1985, pp. 3–7.