An Experience in Using Components for a Modular Construction of Agents for Agent-based Simulations

Jean-Pierre Briot[1,2], Thomas Meurisse[1], Frédéric Peschanski[1]

[1]Laboratoire d'Informatique de Paris 6 (LIP6), Université Paris 6 - CNRS
75252 Paris Cedex 05, France

[2]Laboratório de Engenharia de Software (LES), DI, PUC-Rio
Rio de Janeiro, RJ 22451-900, Brazil

{Jean-Pierre.Briot,Frederic.Peschanski}@lip6.fr

Meurisse.Thomas@wanadoo.fr

Resumo. *Este artigo resume nossa experiência de uso de um modelo de componente para conceber e construir agentes para simulações baseadas em agentes. Neste modelo, chamado MALEVA, os componentes encapsulam varias unidades de comportamento de agentes (por ex: seguir um gradiente, fugir, morrer, reproduzir...). Entre suas especificidades, o modelo expande os princípios de composição de software para especificação do controle por meio de portas e componentes de controle. A noção de componente composto permite construir comportamentos complexos, a partir de outros mais simples. Alguns exemplos ilustram a capacidade do modelo em facilitar a construção progressiva de comportamentos de agentes e seu suporte a várias formas de reutilização potencial. Nós discutimos também os benefícios do modelo para um controle refinado de ativação e de ordenamento.*

Abstract. *This paper summarizes our experience in using a component model for constructing agents for agent-based simulations. In this model, named MALEVA, components encapsulate various units of agent behaviors or activities (e.g., follow gradient, flee, die, reproduce). Among its specificities, it extends the principles of software composition to the specification of control, through the notions of control ports and control components. A notion of composite component allows complex behaviors to be constructed from simpler ones. Some examples illustrate how our model may support a progressive construction of agent behaviors and also various forms of potential reuse. We also discuss the benefits of our model for a fine grain control of activation and scheduling.*

1. Introduction

Agent-based simulation (Sichman and Antunes 2006) is recognized as an important approach for the modeling and simulation of various phenomena (e.g., biological, ecological, social, economic...). We believe that the concept of agent (Ferber 1999) is both structuring enough (unit of activity, of interaction) and versatile enough (reactive or cognitive agents) for simulation applications, as well as for other types of applications. It is important to note that, for simulation applications, the domain specialists 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 believe that using software engineering principles could 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.

In this paper, we describe 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. Requirements for simulation had an influence on the initial design and evolution of MALEVA, as well as on the case studies conducted. MALEVA has indeed been used as the direct foundation or as an inspiration by several teams for various agent-based simulation projects, applied to, e.g.: urban migration, traffic simulation (LeCerf and Pintado 1997), fish tanks evolution, but also to: robot architectures (Bouraqadi and Stinckwich 2007) and distance learning (Aniorte 2003). Therefore, we believe that our component model may also be useful for other types of applications.

We describe three examples in the paper. The first example is a variant of a classical prey/predator simulation example (Ferber 1999). The second one is the re-engineering of a significant application, the MANTA ant colonies simulation framework (Drogoul et al. 1995). The third one is a simplification of another real application, a micro-simulation of population evolution (INSEE 1999). We hope that these three examples provide some hints on how MALEVA can support bottom-up as well as top-down design, and also how it offers some potential for reuse and specialisation, through: structural composition of behaviors, abstract behaviors and design patterns, and specialization of intra-agent scheduling policies. Two other papers, focus respectively on: a methodological framework for agent-based simulations and how MALEVA may help at closing a gap between the domain model of a thematician and the operational model of a computer scientist (Briot and Meurisse 2006) ; and an analysis of different architectural styles for agent architectures (Briot et al. 2007). This paper focuses on lessons learned from the examples and also on enginering concerns.

2. 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]

[1] In this paper, we focus on the issue of components at the *agent level*, to decompose the internal structure of one given agent through components. We do not address here the use of components at the *system level* (each agent is implemented as one component), as, e.g., in (Melo et al. 2004).

[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

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

Table 1. Data and control ports

2.1. Data Flow and Control Flow

In MALEVA, a distinction is made between the activation control flow and the data flow connecting the components. As we will show in Section 2.2, this characteristic and likely specificity of our model,[3] 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.[4]
- *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 1, 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 (OMG 2007).

2.2. An Introductory Example

As for an introduction, we start with a first and very simple example of composition of components: a sequence of two components, illustrated at the left side of Figure 1. 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 our graphical notation for components and connexions, data flow connexions are shown in solid lines, and control flow connexions in dotted lines.

The right side of Figure 1 recombines the two same components, but this time activated concurrently.[5] 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, and, as we will discuss in Section 5, also a fine grained control over activation policies.

---

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 (Bruneton et al. 2004).

[3]Note that, in the different context of project management applications, a distinction between data flow and control flow was introduced in SADT (Marca and McGowan 1987). But SADT introduced this distinction mostly at the design level, whereas MALEVA makes it also available at the implementation level.

[4]Data ports are typed, as further discussed in Section 6.2.

[5]The control connexions have been changed accordingly, but not the data connexions. The semantic is analog to the *pipes and filters* (Shaw and Garlan 1996) architectural style: component `B` consumes what `A` produces while they are both active simultaneously.

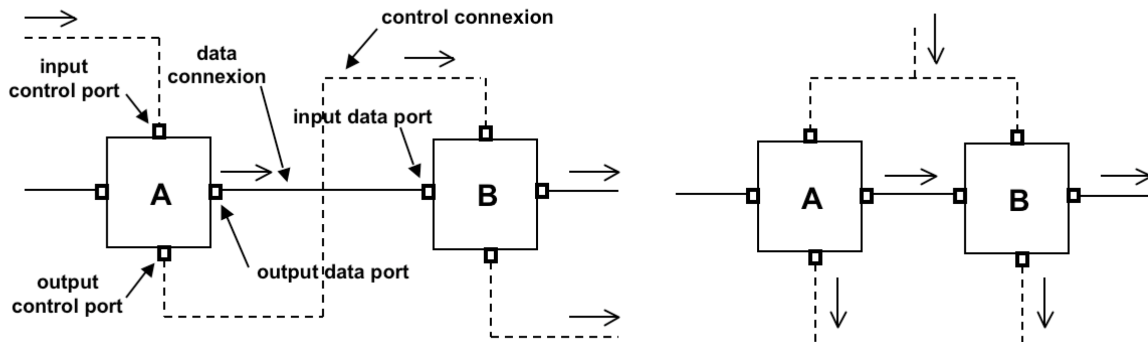

Figure 1. Sequential vs concurrent activation of two components

3. 1st Example: Bottom-Up Design of Prey and Predator

3.1. Abstract Architecture of a Situated Agent

Our first example defines behaviors of situated agents within an ecosystem. 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[6] (see Figure 2) usually follows the computational cycle: *sensors → behavior → effectors*.

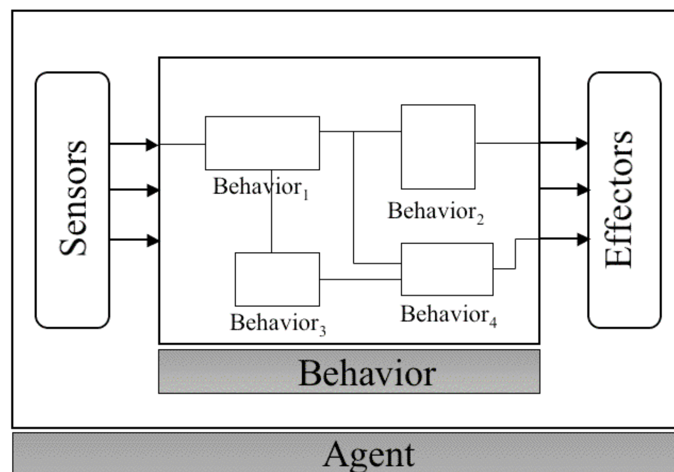

Figure 2. Abstract architecture of a situated agent

3.2. 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 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`.

[6]For other applications, e.g., in Section 5 on micro-simulation, agents are not necessarily situated (within an environment) and thus do not use any sensor/effector.

3.3. 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)

3.4. Prey Behavior

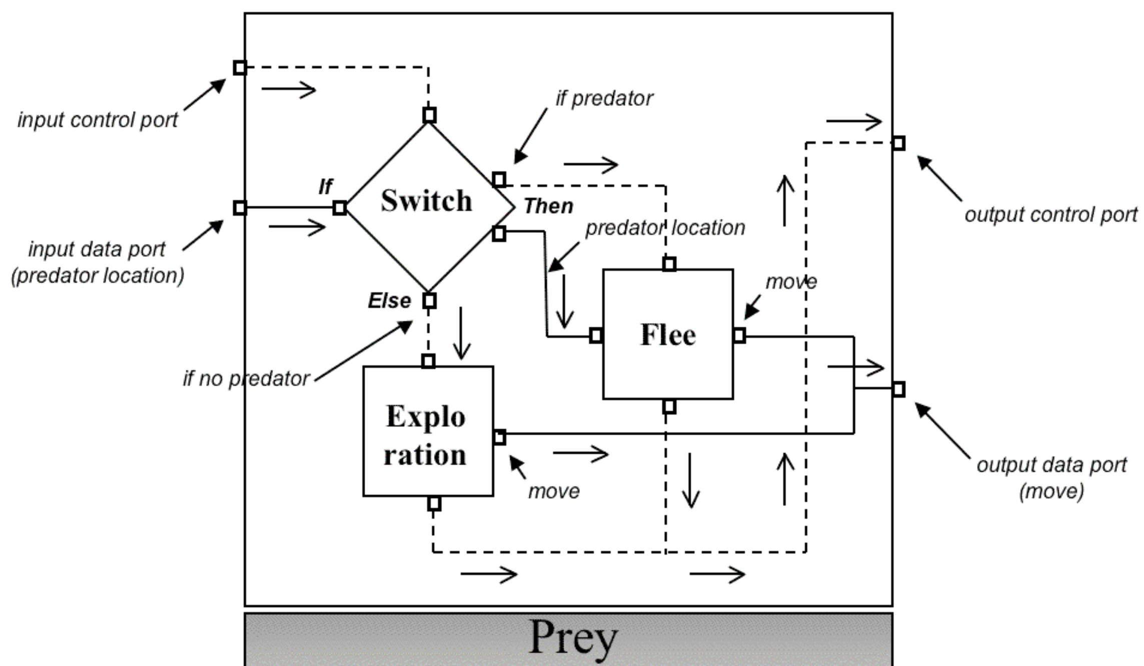

Figure 3. Prey behavior

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`

[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 (Meurisse 2004).

output control port, which activates `Exploration` behavior.[8] The result is shown at Figure 3.[9]

3.5. Predator Behavior

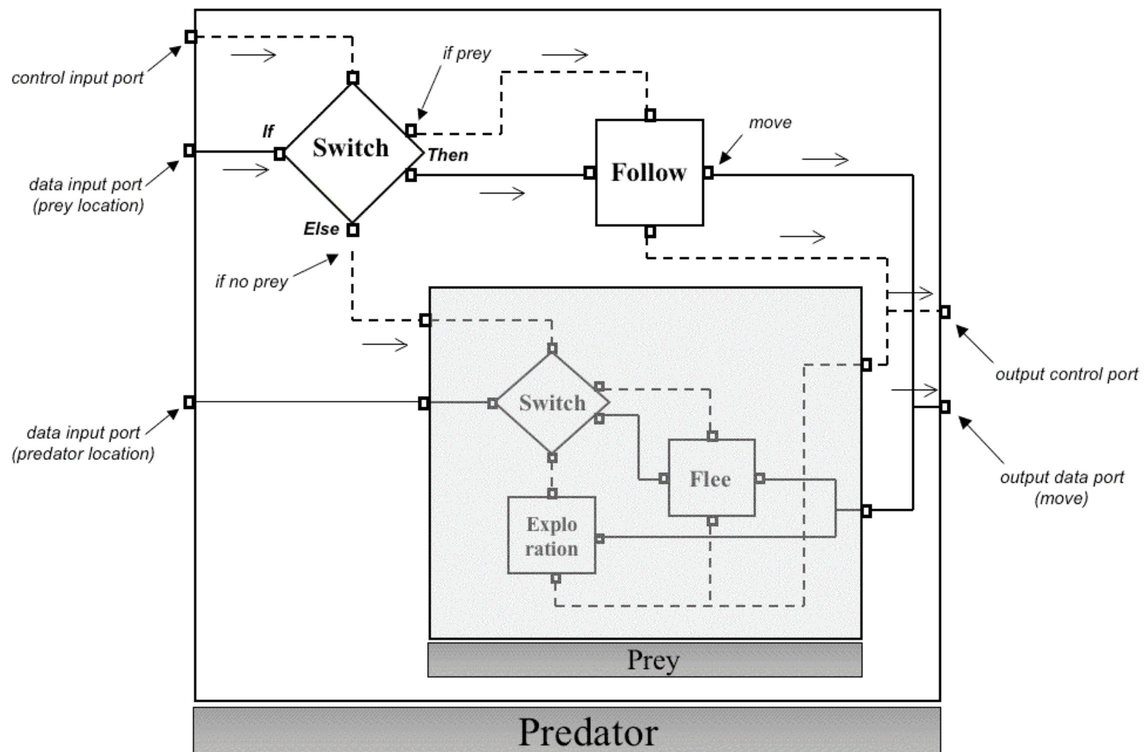

Figure 4. Predator behavior (with Prey as a sub-component)

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 a prey that he perceives). 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 4).[10]

4. 2nd Example: Top-Down Design of Ants

This second example illustrates a top-down design of agent behaviors. The complete application was the reengineering in MALEVA of the MANTA framework for simulation of ant colonies to study their sociogenesis (Drogoul et al. 1995). Various types of ant

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

[9] We assume that the input data port (perception of a predator in the environment) and the output data port of the `Prey` behavior have been connected to the corresponding sensor and effector data ports, along the general architecture of a situated agent, shown at Figure 2.

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

agents are considered: egg, larva,[11] ant worker, and queen. In this paper, we focus on the top-down design of the behavior of an ant worker.

4.1. The Living Abstract Behavior

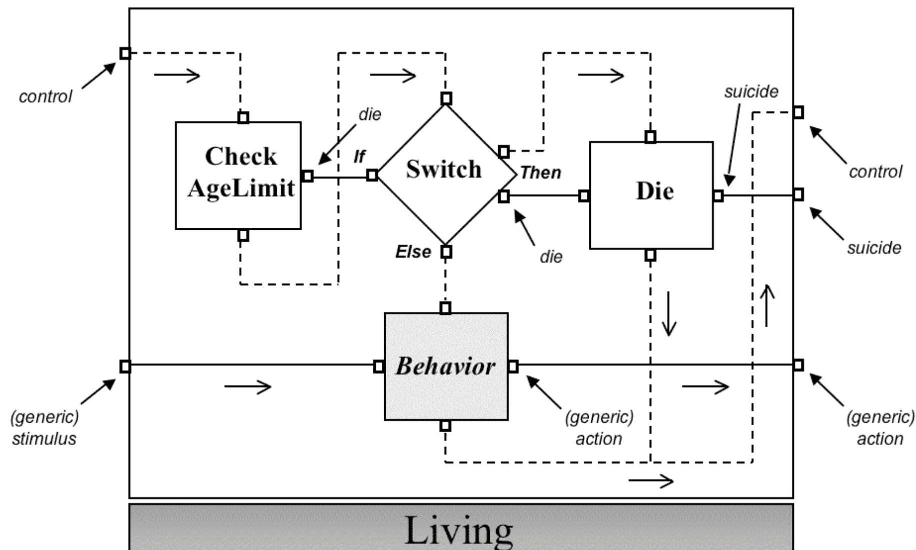

Figure 5. Living abstract behavior

The first step of our design identifies some feature common to each living agent, the ability to age (and ultimately to die). Therefore, we design a behavior, partly abstract, named `Living`, shown at Figure 5. It includes 4 sub-behaviors/components: `CheckAgeLimit`, which controls the ageing process (it includes a variable `age`, incremented for each activation step and compared with the agent age limit) ; `Die`, which implements the death of an agent ; `Behavior` *abstract* behavior (see below), shaded in the figure ; and a `Switch` control component. When the agent reaches its age limit, `CheckAgeLimit` emits a `die` data. Then `Switch` activates `Die`, which in turn emits `suicide` data, ultimately conveyed to the actuators (in practice, it may, e.g., remove the agent from the environment). Otherwise, `Behavior` is activated by `Switch`.

We found out this design to be useful in various contexts and thus extracted it as a recurring and "*pluggable*" design, in a similar way to the object-oriented principle of a *design pattern* (Gamma et al. 1995). Interestingly, the MALEVA mini-patterns encompass data and control features. `Living` implements that pattern as some kind of *"mini black-box framework"*, where the unique *hot spot* is the abstract component `Behavior`. To construct a specific agent behavior, we replace (instantiate) `Behavior` with a concrete behavior, specific to, e.g., an ant, egg, larva or queen. We have identified other patterns (not further detailed here), e.g., the *"exploration unless perception"* pattern used by `Prey`, described in Section 3. It will be reused by the `AntActivity` component, to be described in Section 4.2.

[11]Note that the metamorphosis process - from egg to larva and then to ant or queen - leads to the issue of *behavioral evolution* and to the corresponding *architectural dynamicity*. This will be discussed in Section 8.

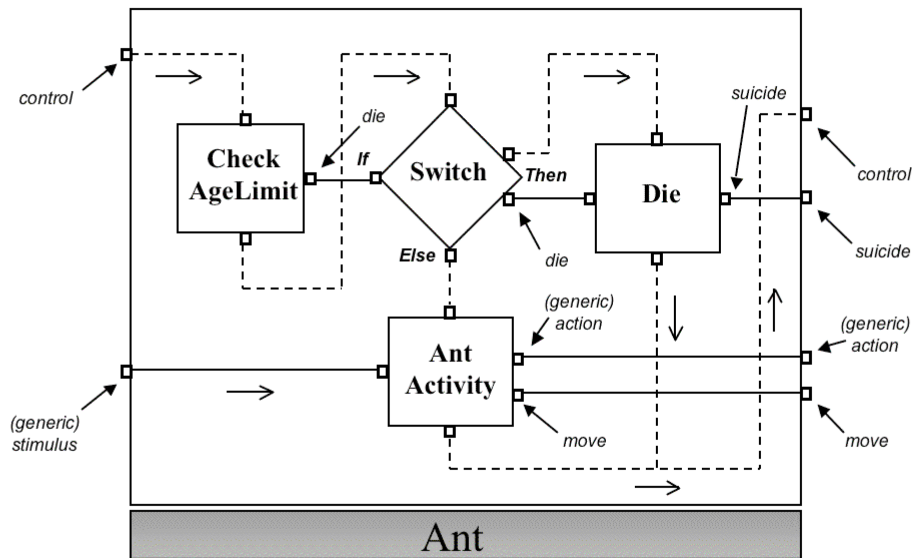

Figure 6. Ant behavior

4.2. Ant Behavior

A worker ant has a relatively complex behavior because its various activities: moving, pheromone following, egg carrying, egg caring. For the sake of understandability and concision, we describe here a simplification of the real application (Drogoul et al. 1995). First, we instantiate `Living` into a concrete behavior specific to ants, named `Ant`. In practice, the abstract sub-component `Behavior` is replaced by a concrete component, named `AntActivity`. The result is shown at Figure 6.[12]

We now define the internal behavior of an ant, named `AntActivity`. An ant explores its surroundings through a random movement (`Exploration` behavior), unless it perceives some stimulus (`ManageStimulus` behavior). Thus, `AntActivity` reuses the *"exploration unless perception"* pattern, already used for `Prey` and `Predator` behaviors (see Figures 3 and 4). Keeping with the top-down approach, we then further decompose `ManageStimulus` as follows. If the ant is located on a stimulus local maximum (`CheckLocalMax` behavior), an action associated to the type of stimulus (e.g., food, pheromone) is emitted (by `LocalMaxAction` behavior). Otherwise, the ant follows the gradient associated to the stimulus (`FollowGradient` behavior). The resulting complete architecture of an ant is summarized at Figure 7 and includes 3 levels and 14 components.

5. 3rd Example: Population Microsimulation

This last example will help at illustrate another merit of making explicit the control flow. It is inspired from an existing application of demography micro-simulation conducted at the French National Institute of Statistics (INSEE) and named Destinie (INSEE 1999). In this (simplified) example, we consider a virtual specie of agents, in which mating of

[12]One may note that `AntActivity` has an additional output data port, in order to distinguish the two possible outputs: action (e.g., leave a pheromone or take food) and move, and their associated effectors and types. An alternative simplification is to consider a single output data port including all types of actions (including move).

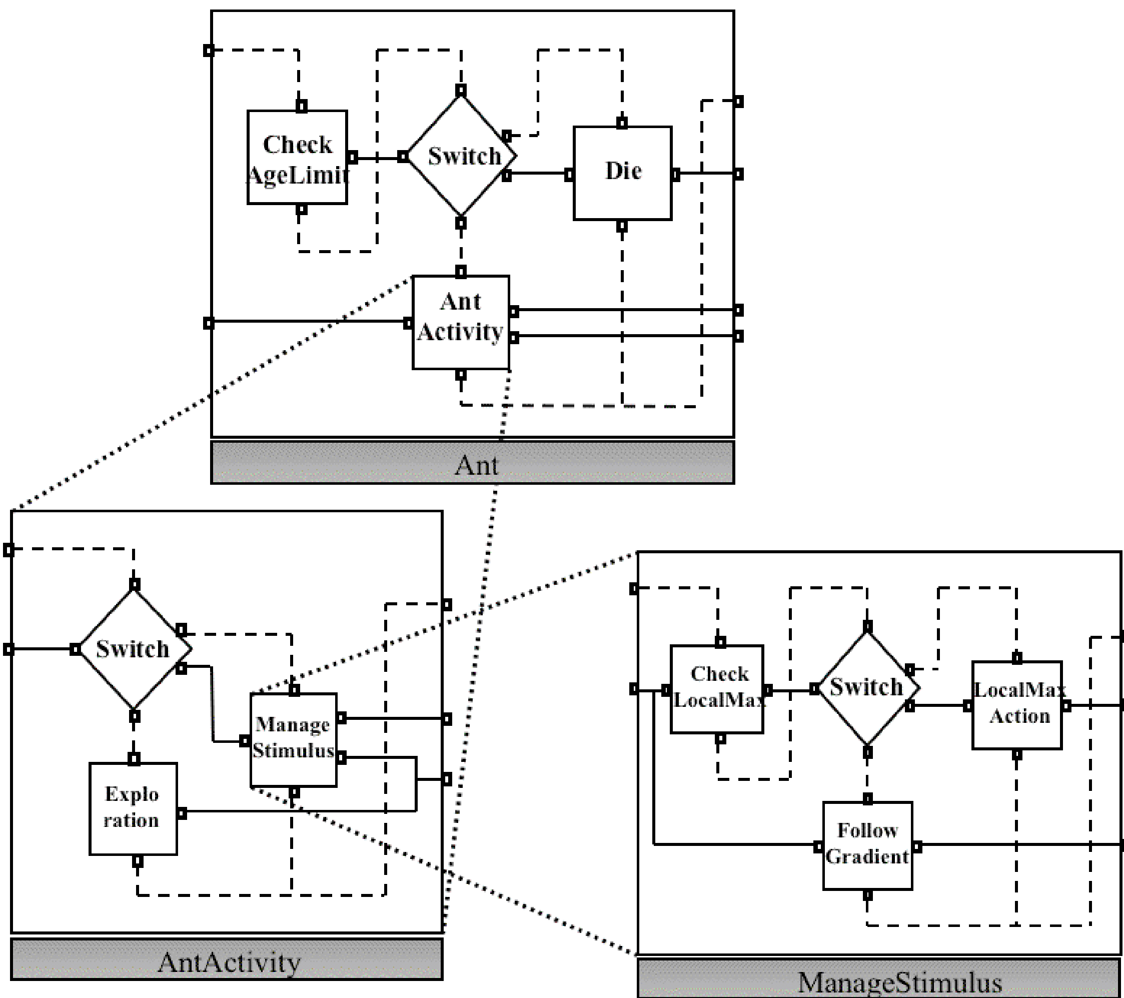

Figure 7. Ant behavior: complete decomposition

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 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 (Gilbert and Troitzch 1999), 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 de-

cision.[13] Figure 8 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.

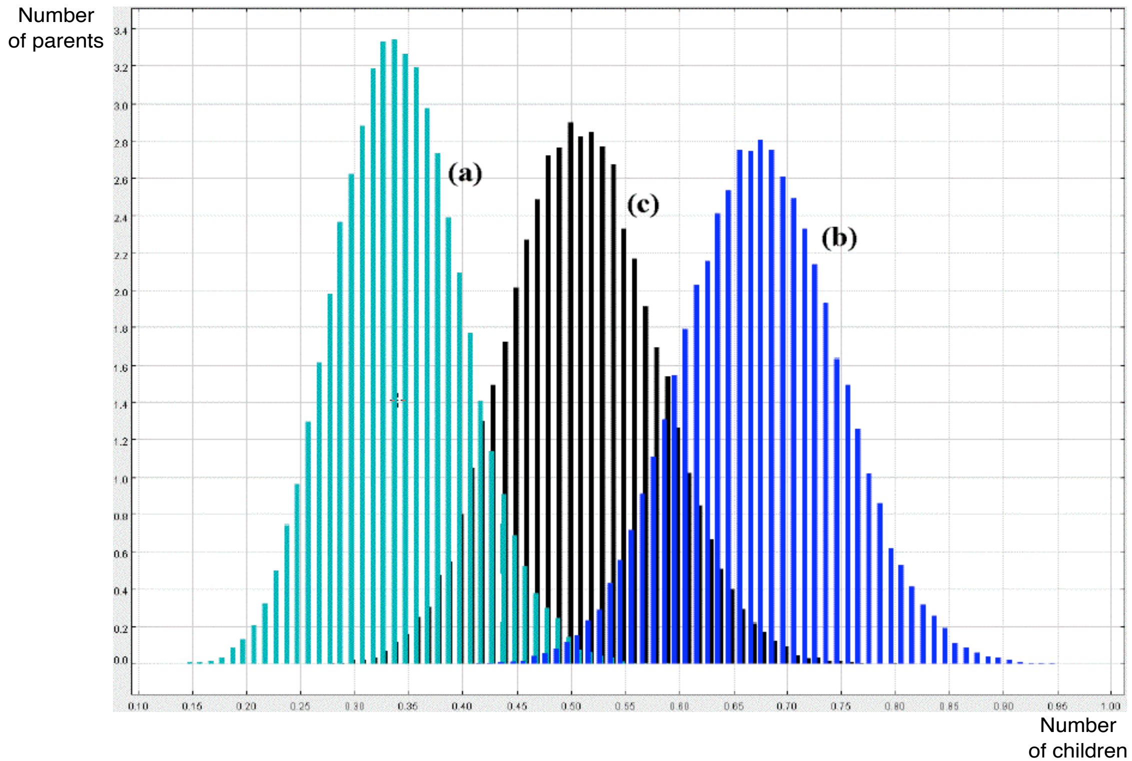

Figure 8. Histogram of number of babies

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.[14] By considering control flow explicitly, MALEVA helps at specifying and controlling temporal dependencies between behaviors, and thus their possible orderings.[15] This is realized via explicit control flow connexions, without any change to the code of behaviors (encapsulation is ensured), as shown at Figure 9.[16] This is notably useful for simulation applications, where the expert may

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

[14]For instance, (Lawson and Park 2000) show that results of simulations can be found biased in cases where the scheduling of the actions within an agent remains deterministic.

[15]Note that an additional dimension, not detailed here, is the mode of activation of components. In the *asynchronous mode*, the different agents (and components) evolve independently. This may be more efficient, specially for the case of distributed implementation, but, unless the designer also uses explicit control connexions between agents, the different agents may not be synchronized (some can compute ahead of others). 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., (Lawson and Park 2000) and (Meurisse 2004)).

[16]Connexions are usually achieved interactively, see Section 6.1. In this example, there are no data ports shown, because the behaviors considered in this case study are without parameters, nor results.

incrementally specify temporal dependencies, independently of behaviors functionalities, and thus experiment and compare various ordering strategies (Meurisse 2004). 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 (Briot and Meurisse 2006).

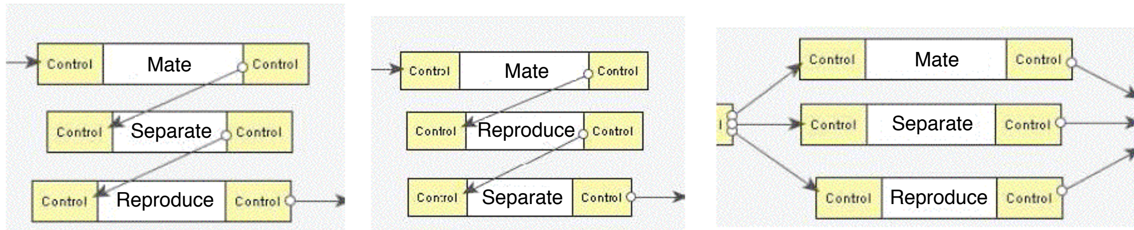

Figure 9. Temporal dependency specifications with CGraphGen

6. 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.

6.1. From Methods to Components

Some interesting feature of CGraphGen (illustrated at Figure 10) 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.

6.2. Implementation

After the initial Delphi implementation, the Java-based re-implementation of MALEVA 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

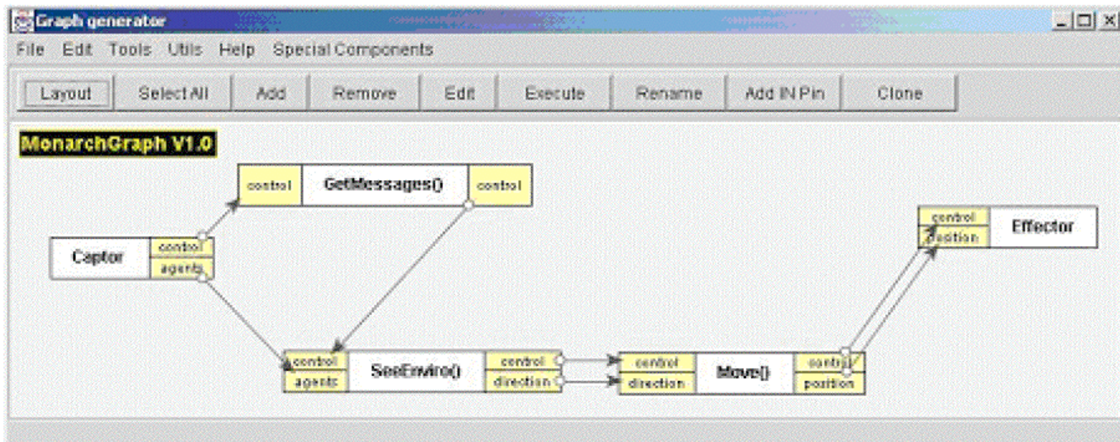

Figure 10. The CGraphGen tool

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 8).

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 (SUN 2007) 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.

A re-implementation of MALEVA into C++ has also been realized, in order to conduct more efficiently large scale experiments (Meurisse 2004). Last, a complete re-implementation of Maleva in Smalltalk (Squeak), named MalevaST (Bouraqadi and Stinckwich 2007), has recently been independently conducted by Noury Bouraqadi et al. at École des Mines de Douai, France, with, e.g., applications to robotics.

7. Related Work

CCM (Corba Component Model) (OMG 2007) 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 (Bruneton et al. 2004) 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) (Zeigler 1985) 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 recent attempts in that direction (see, e.g., some work on mapping DEVS onto Microsoft COM component model (Cho and Kim 2002)).

The subsumption architecture (Brooks 1986) for robots considers basic behaviors of the robot as the units of (de)composition. Behaviors (e.g., random move, obstacle avoidance) are simultaneously active and are organized within some fixed hierarchy and their associated priorities. In practice, a behavior may replace input data of the behavior situated below, as well as inhibit its output data (e.g., in case of close obstacle perception, the obstacle avoidance behavior may take control over lower ones). In MALEVA, the control architecture is completely explicit and arbitrary (through control flow), thus more flexible than the fixed hierarchical control model of the subsumption architecture.

The ABLE architecture (Bigus et al 2002) for Autonomic Computing is a good representative of a bottom-up orientation, adopting a toolbox approach where each tool is implemented as a component and an architecture is constructed by forming tool chains. Its set of tools components is organized along the types of tools and processing techniques (e.g., statistics, learning, search). All these components are implemented with JavaBeans (SUN 2007). The designer of a system (e.g., an autonomic load balancing mechanism for application servers) identifies processing steps, implements them through components and connects them.

The Component-Based Agent Framework (CBAF) (Goradia and Vidal 2003) is another set of building blocks for contructing agents, also based on JavaBeans. It is more aimed at cognitive agents and has been applied to robot soccer simulation. Like ABLE, CBAF proposes a classification of components (4 types: behavior, decision, activity and agent components). CBAF behavior components are close to the way MALEVA uses components to model behaviors. Also, CBAF simplest decision component, named `DifThenElse`, is actually similar to MALEVA's `Switch` control component. Building up on the experience of CBAF and of MALEVA looks as a promising path to provide libraries of components for various types of agent architectures and applications. Note that neither ABLE nor CBAF make a separation of data-flow and control-flow, thus they provide less flexibility for activation control.

The JADE architecture (Bellifemine et al. 2001) 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. 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 keeping the architecture of an agent partly hidden within the code.

The DESIRE methodology and component model (Brazier et al. 2001) is a component model and methodology for constructing agents. DESIRE is more high-level and knowledge-oriented than MALEVA and is more aimed at cognitive agents. It is based on a formal description considering separately a process (and component) level and a knowledge level. This approach enables some possibilities of verification, but at the cost of some added complexity in specifications. As opposed to MALEVA, DESIRE does not provide a fine grained control model for components.

Like MALEVA, JAF (Java Agent Framework (Horling 1998)), 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,[17] and its management through explicit control.

The MaSE methodology (DeLoach 1999) 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.

Last, we may also cite some work on using aspect-oriented programming (AOP) to facilitate integration and combination of agent properties (e.g., autonomy, learning, mobility) (Garcia et al. 2004). But this is not applied to the composition of fine grained behaviors as for the case of MALEVA. The reader may also refer to (Briot et al. 2007) for an extensive discussion of various rationales and architectural styles for modular agent architectures, and to (Bordini et al. 2006), for a general recent survey on architectures and languages for multi-agent systems.

8. Further Issues and Future Directions

An issue is the identification of a methodology and its companion tools to help at identifying and constructing agents with MALEVA components. If fact, most of multi-agent methodologies offer initial steps rather independent of a specific targeted architecture of agent. We may thus imagine reusing the initial steps of a methodology such as MaSE (see Section 7) to design MALEVA components. Regarding notations, we are aware that the initial diagrammatic notations of MALEVA could be adapted and reformulated using meta-modelling facilities of model-driven engineering. Last, regarding tools, we believe that CGraphGen (see Section 6.1) is a promising prototype and could be an inspiration for further tools more integrated into a standard interface development environment.

Another issue is in providing rich libraries of components and abstract architectures, such as behavior components (e.g., `Exploration`), abstract components (e.g., `Living`), control components (e.g., `Switch`) and "system" components (e.g., sensors).

[17]Considering performance, it is obvious that a very fine grained decomposition of agent behaviors will have a cost. But in MALEVA, this is the designer responsability for the exact decomposition granularity for each agent and thus to address the usual trade-off between genericity and performance. Meanwhile, we are considering the possibility of providing a code generator for transformation (compacting) of a composite component into a primitive component, with additional optimisations (e.g., transform some activations in synchronous method calls when intra-agent concurrency is not used).

They could support the types of architectures and applications targeted, for agent-based simulation, but also for other agent architectures and application fields, such as, e.g., interaction protocols for e-commerce and reasoning components for rational/cognitive agents, as we believe that the MALEVA model of component has a wider potential scope. The experience of CBAF (Goradia and Vidal 2003) is a very promising example in that direction. Note that the component-based design of agents and the support of CASE tools using possible information (e.g., typing) should help in assisting the designer to analyze existing designs and to create new ones.

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 (e.g., see Section 4). Some radical alternative approach to reduce the control graph complexity, and also to make it more accessible to formal analysis, is to abstract it in an adequate formalism. We think that a process algebra, such as CCS (Milner 1982), could allow a concise representation of complex activation patterns. Such formal characterisation would also allow the semantic analysis of such specifications, for example through model checking. The idea is close to coordination languages, but for very fine grained components. The starting point is to model data used for control as channels (e.g., presence of prey as `isPrey` channel) and synchronize activity of behaviors (e.g., `Flee`) on them. The reformulation of the control graph of the predator (Section 3) would be the following compact term: `isPrey.Follow || isPredator.Flee || (isNoPrey.Exploration + isNoPredator.Exploration)`

Another issue is the dynamicity of behaviors. An example is the metamorphosis process of ants (from an egg, to a larva, to an ant, see Section 4). For other types of applications (e.g., nomadic computing), reconfiguration of behaviors may also be externally triggered by needs for adaptation to a dynamic environment. MALEVA current implementation strategy relies on a specific meta-component to manage the reconfiguration and re-assemblage of behaviors. Based on the reconfiguration request of one of the current behavior component, the meta-component defines the new architecture, compares it with the current one, considers the sets of components to add and of components to remove, and makes the connexions. We are considering using a higher level mechanism, based on concepts of configurations, roles and policies, such as the MaDcAr prototype model of automatic reconfiguration (Grondin et al. 2006).

9. 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 examples have illustrated how MALEVA may help at genericity and reuse of components, through: structural composition of behaviors, abstract components and design mini-patterns, 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, automobile traffic simulation (LeCerf and Pintado 1997), artificial societies, robot architectures (Bouraqadi and Stinckwich 2007) and distance learning (Aniorte 2003). 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.

References

[Aniorte 2003] Aniorte, P. (2003) A Distributed Adaptable Software Architecture derived from a Component Model, *Computer Standards & Interfaces*, 25(3):275–282, June.

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

[Bigus et al 2002] Bigus, J.P., Schlosnagle, D.A., Pilgrim, J.R., Mills, W.N. and Diao, Y. (2002) ABLE: A Toolkit for Building Multiagent Autonomic Systems, *IBM Systems Journal*, 41(3):350–371.

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

[Bouraqadi and Stinckwich 2007] Bouraqadi, N. and Stinckwich, S. (2007) Towards an Adaptive Robot Control Architecture, *2nd National Workshop on Control Architectures of Robots (CAR'2007)*, Paris, France, May-June.

[Brazier et al. 2001] Brazier, F., Jonker, C., Treur, J. and Wijngaards, N. (2001) Compositional Design of a Generic Design Agent, *Design Studies Journal*, (22):439–471.

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

[Briot et al. 2007] Briot, J.-P., Meurisse, T. and Peschanski, F. (2007) Architectural Design of Component-based Agents: a Behavior-based Approach, *Programming Multi-Agent Systems - ProMAS 2006*, No 441, LNCS, Springer, pp. 73–92.

[Brooks 1986] Brooks, R.A. (1986) A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation*, 2(1):14–23, March.

[Bruneton et al. 2004] Bruneton, E., Coupaye, T., Leclerc, M., Quema, V. and Stefani, J.-B. (2004) An Open Component Model and its Support in Java, *Component-Based Software Engineering (CBSE'2004)*, No 3054, LNCS, Springer, pp. 7–22,

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

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

[Drogoul et al. 1995] Drogoul, A., Corbara, B. and Fresneau, D. (1995) MANTA: Experimental Results on the Emergence of (Artificial) Ant Societies, *Artificial Societies: the Computer Simulation of Social Life*, UCL Press, London, U.K.

[Ferber 1999] Ferber, J. (1999) *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, Addison-Wesley.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns - Elements of Reusable Object Oriented Software*, Addison-Wesley.

[Garcia et al. 2004] Garcia, A., Kulesza, U. and Lucena, C. (2004) Aspectizing Multi-Agent Systems: From Architecture to Implementation, *Software Engineering for Multi-Agent Systems III*, No 3390, LNCS, Springer, pp. 121–143.

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

[Goradia and Vidal 2003] Goradia, H.J. and Vidal, J.M. (2003) Building Blocks for Agent Design, *Agent-Oriented Software Engineering IV*, No 2935, LNCS, Springer, pp. 153-166.

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

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

[INSEE 1999] INSEE (1999) 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.

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

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

[Marca and McGowan 1987] Marca, D.A. and McGowan, C.L. (1987) *SADT - Structured Analysis and Design Technics*, McGraw-Hill.

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

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

[Milner 1982] Milner, R. (1982) *A Calculus for Communicating Systems*, Springer.

[OMG 2007] Object Management Group (2007) Corba Component Model (CCM), "`http://www.omg.org/technology/documents/formal/components.htm`".

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

[Sichman and Antunes 2006] Sichman, J.S. and Antunes, L. (Eds.) (2006) *Multi-Agent-Based Simulation VI*, No 3891, LNAI, Springer.

[SUN 2007] Sun Microsystems Inc. (2007) JavaBeans Specification, "`http://java.sun.com/products/javabeans/`".

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