Architectural Design of Component-Based Agents: A Behavior-Based Approach

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

[1] Laboratoire d'Informatique de Paris 6 (LIP6)
Université Paris 6 - CNRS
Case 169, 4 place Jussieu
75252 Paris Cedex 05, France
{Jean-Pierre.Briot,Thomas.Meurisse,Frederic.Peschanski}@lip6.fr
[2] Currently visiting CS Dept., PUC-Rio, Rio de Janeiro, Brazil

Abstract. This paper relates an experience in using a component model to design and construct agents. After discussing various rationales and architectural styles for decomposing an agent architecture, we describe a model of component for agents, named MALEVA. In this model, components encapsulate various units of agent behaviors (e.g., follow gradient, flee, reproduce). It provides an explicit notion of control flow between components (reified through specific control ports, connexions and components), for a fine grain control of activation and scheduling. Moreover, a notion of composite component allows complex behaviors to be constructed from simpler ones. Two examples, in the domain of multi-agent based simulation, are presented in this paper. They illustrate the ability of the model to facilitate both bottom-up and top-down approaches for agent design and construction and also to help at different types of potential reuse.

Keywords: component, agent, multi-agent systems, behavior, design, composition, architecture, simulation.

1 Introduction

Components and multi-agent systems are among current popular approaches for designing and constructing software. Both of them propose abstractions to organize software as a combination of software elements, with easier management of evolution (such as changing and adding elements). We consider that multi-agent systems push further the level of abstraction and the flexibility of component coupling, notably through self-organization abilities [5]. Meanwhile, we believe that the component concept and technology may help in the actual construction of multi-agent systems:

– at the *system level*, we may consider each agent as a component, to provide some support for integration, configuration, packaging and distribution of multi-agent systems,

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 73–92, 2007.

– at the *agent level*, by providing some support for structuration, (de)composition and reuse of its internal architecture.

In this paper, we focus on the second category. Indeed, we believe that the design and construction of an individual agent can benefit from the principles of software components (encapsulation, explicit connectors...). Our objective is to help in an incremental design of agents as the composition of simpler agent behaviors and activities (e.g., follow gradient, flee, reproduce...). Our main application field target is multi-agent-based simulation of phenomena (biological, ecological, social, economical...). Their specific requirements definitely influenced our design decisions, as well as our case studies and applications conducted. Meanwhile, we believe that the scope of the component model that we propose goes beyond the domain of multi-agent-based simulations, and that other application areas could benefit from some of its principles, e.g., making control available at the composition level.

After first discussing some rationales for the design of component-based agent architectures, and referring to related work, we describe a component model named MALEVA, which aims at encapsulating and composing units of behaviors to describe complex agent architectures. This component model does not impose a specific architectural style. One of its specificities is that it applies the principles of components and software composition to the specification of control, through the notions of control ports and control components. Two examples will be presented in the paper. They illustrate how MALEVA can support bottom up as well as top down design, and also how it offers some potential for reuse and specialization, through: structural composition of behaviors, abstract behaviors and design patterns.

2 Rationales and Styles for Agent Architectures

We consider an *agent architecture* as the description of the relations between the software (or sometimes hardware) modules that implement the various agent functions. Except for simple reactive agents, the architecture of an agent may be complex. It is thus useful to describe it in terms of simpler lower level components that interact with each other.

Inspired by the seminal work on software architectures by Shaw and Garlan [23], we tentatively propose a classification for agent architectures from the perspective of *architectural styles*.[1] In this paper, we focus on the rationales for decomposition and on their impact on the reuse of the architecture or/and of its components. It is important to note that we do not expect our typology to be exhaustive. Also note that, as for software architectures [23], a complex architecture (e.g., InteRRaP, see Section 2.3) may juxtapose and combine several architectural/decomposition styles.

[1] There is of course no unique typology for agent architectures, and we may find other classifications in the literature (e.g., in [20]), such as horizontal/vertical or reactive/cognitive/hybrid.

2.1 Cycle-Based Style

The architectural style based on the notion of cycle, among the simplest ones, follows the basic computational cycle of an agent situated within an environment: perception (of the environment), state update (data or/and mental state), generation of intentions (of actions), action. An example is a general architecture for situated reactive agents, introduced in Section 4.1. Another example is Yoav Shoham's Agent-Oriented Programming (AOP) architecture for cognitive agents [24] (see Figure 1).

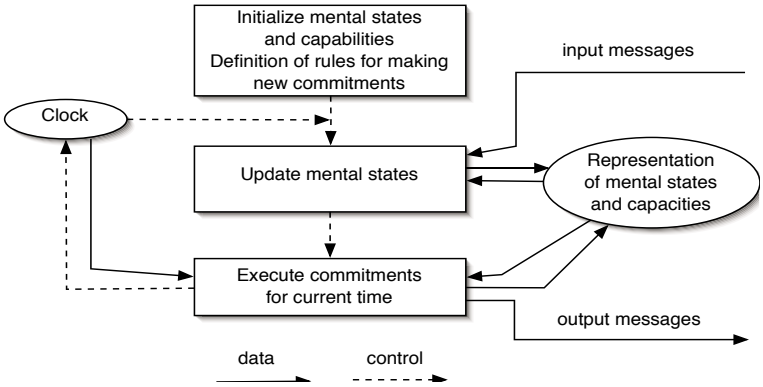

Fig. 1. AOP Architecture

2.2 View-Based Style

Another style of decomposition, more structural than computational, considers various view points (e.g., interaction, environment, organization...) and their respective units of processing (e.g., perception, communication, coordination...). An example is the VOLCANO architecture [21], which decomposes an agent along four dimensions: A (agent), E (environment), I (interaction) and O (organization). The architecture is actually a *framework* with components (named bricks) `A`, `E`, `I` and `O`. Figure 2, illustrates the central position of the `A` brick. Note that the designer needs also to implement inter-bricks adaptors/wrappers, respectively `AE`, `AI`, `AO`, `EI`, `EO` and `IO`.

Another example is the generic model of agent architecture (Generic Agent Model: GAM) [4], based on the DESIRE methodology and component model [3]. It includes a set of components (e.g., interaction management, information maintenance) each dedicated to a specific type of processing (see Figure 3, imported from [4]). The GAM generic model (also a framework) has been instantiated to model (retro-engineer) various agent architectures, such as BDI, and ARCHON.

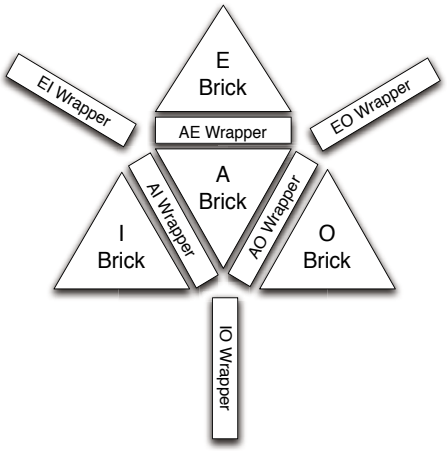

Fig. 2. VOLCANO architecture

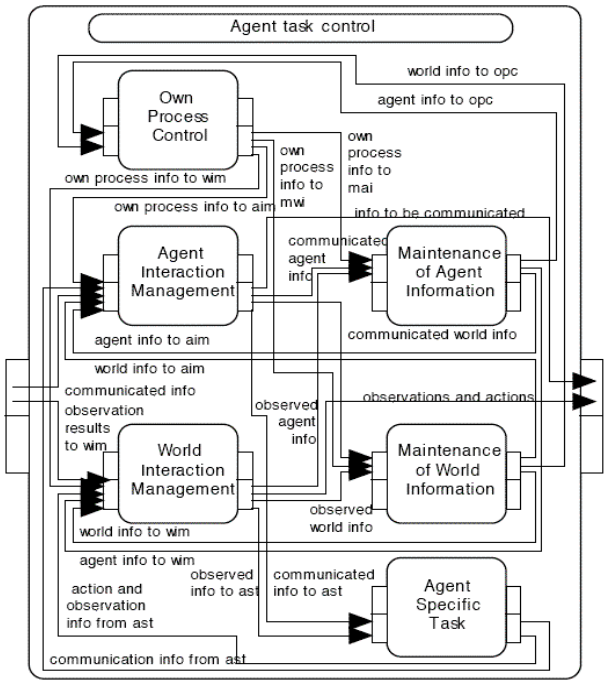

Fig. 3. DESIRE GAM architecture

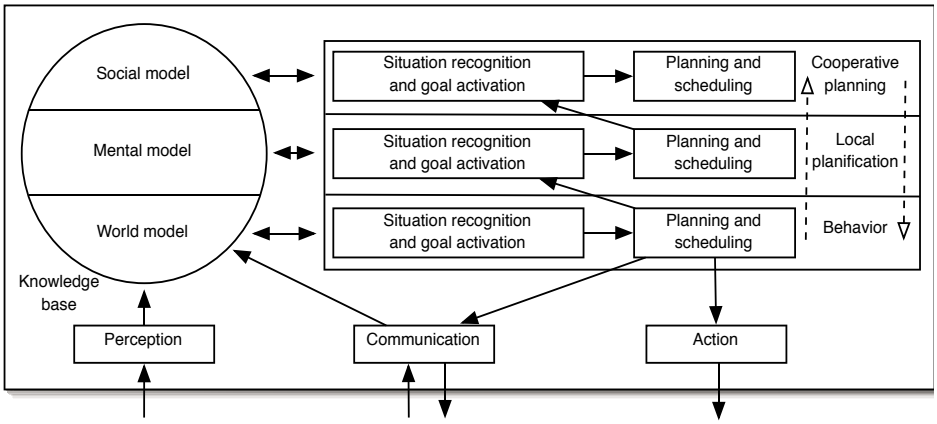

Fig. 4. InteRRaP architecture

2.3 Level-Based Style

Another approach considers different various levels (and models) of knowledge, reasoning and action, to structure the architecture, e.g., through the distinction between world model, self model, and social model. A representative example is the InteRRaP architecture [19] (see Figure 4, adapted from [19]). InterRRaP is structured as a hierarchy of three layers, concurrently active: cooperative planning, local planning, and reactive behaviour. The internal architecture of each level follows the same model, based on situation recognition and planning. A knowledge base structures the information manipulated by each layer, thus respectively: social model, mental model, and world model. Two dual control mechanisms between layers are considered: upwards activation request, to activate the layer above, and downwards commitment signal, to delegate execution of commitments to the layer below.

2.4 Behavior-Based Style

A more radical style of decomposition, considers basic behaviors of the agent as the units of (de)composition. An example is Rodney Brooks' subsumption architecture [7], in which various behaviors (e.g., random move, obstacle avoidance...) are simultaneously active. They are organized within some fixed hierarchy and their associated priorities (see Figure 5, adapted from [7]). In practice, a behavior may replace input data of the behavior situated below, as well as inhibit its output data (for instance, in case of close obstacle perception, the obstacle avoidance behavior may take control over other ones).

2.5 Discussion

The architectures that we surveyed are usually more tailored at a specific model of agent (e.g., cognitive collaborative agent for the InteRRaP architecture,

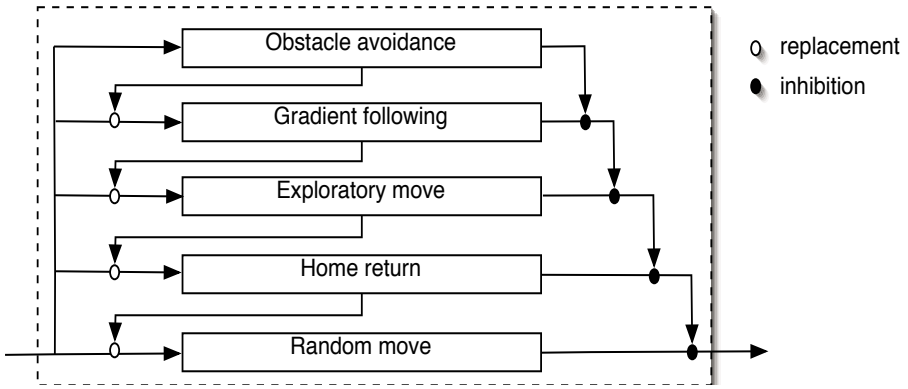

Fig. 5. Subsumption architecture

situated agent or robot for subsomption architecture). They aim at genericity, but, in practice, they may not provide enough flexibility. For instance, it is often uneasy, in some case almost impossible, to replace and add, and moreover to remove components. Also, the implementation of the architecture does not always follows the basic requirements of software components (output interfaces, explicit connectors...),[2] neither classical component models (e.g., JavaBeans). The VOLCANO architecture clearly separates the components, but in order to replace one component by another one, we are forced to re-implement the corresponding adaptors. The subsumption architecture actually represents an abstract model of architecture, instantiated for a specific robot and objective (e.g., see Figure 5). It is concise but also difficult to evolve (e.g., add a component), as the fixed hierarchy is the key of control between components.

We think that the behavior-based style of decomposition, as used in the subsumption architecture, is somewhat radical, but it is also the closest to the concept that we aim at decomposing: the *behavior* of the agent. A radical option is then to only offer a model of component, in a way similar to a general software component model such as JavaBeans, without a specific agent architecture. The main difference is that the control aspects, and not just the functionalities, must be also made composable in a flexible and open way, in order to replace the fixed hierarchical control model of the subsumption architecture. We propose that, by keeping in line with the idea of a component model, control is specified/reified through control ports and connexions (see Section 3.1), in order to represent arbitrary patterns of control flow.

[2] The JADE architecture [1] 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, e.g., the most advanced one, `FSMBehaviour`, relies on finite state automata. 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.

3 The MALEVA Model of Component

As has been explained above, the objective of the MALEVA agent component model is to help in incremental design and construction of agent behaviors by composing simpler behaviors, encapsulated as software components.

3.1 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 show in Section 3.2, this characteristic and specificity of our model, which decouples the functional architecture from the activation control architecture, makes 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. Note that data ports are typed, as discussed in Section 6.1.
- *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 transfered to its output control port.

In addition to the *semantic* distinction between data ports and control ports, specific to MALEVA, we find the common *structural* distinction between input ports and output ports. Table 1 summarizes that.

Table 1. Data and control ports

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

3.2 An Introductory Example

Figure 6 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.

Figure 7 recombines the two same components, but this time activated concurrently. Note that the control connexions have been changed accordingly, but not the data connexions. The semantic is analog to the *pipes and filters* [23]

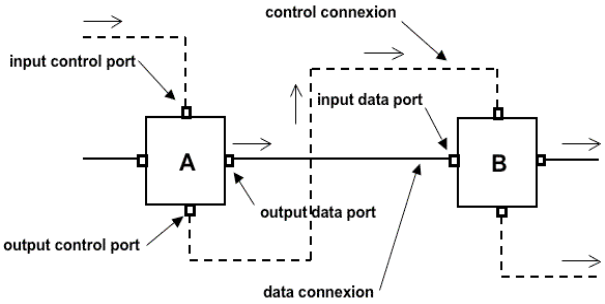

Fig. 6. Sequential activation of two components

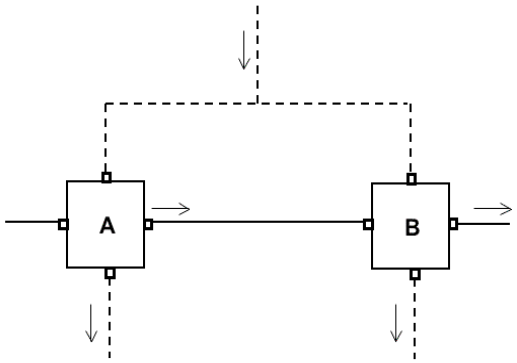

Fig. 7. Concurrent activation of two components

architectural style: component `B` consumes what `A` produces while they are both active simultaneously.

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.[3] Flow of control is specified outside of the components, which provides more genericity on the use of components. The designer of the application also has a fine grained control over activation policies between components. Making possible the control of temporal dependences between behaviors - which are usually left implicit -, independently of behaviors functionalities, helps experts at experimenting with various strategies, at comparing results with the target models, and at quantifying the impact on biases [6].[4]

[3] An additional dimension, the mode of activation of components, which could be asynchronous or synchronous, will be briefly addressed in Section 6.2.

[4] For instance, [14] shows that results of simulations can be found biased in cases where the scheduling of the actions within an agent remains deterministic.

3.3 Designing Agent Behaviors

Designing and constructing agents for a given application should ideally consist mostly in assembling existing behavior components. We therefore assume that there is a library of behavior components associated to the application domains targeted. A component may be primitive (the behavior is written in the underlying language, e.g., Java) or *composite*,[5] as the encapsulation of a composition (assembly) of components.

4 A First Example: Bottom-Up Design of Prey and Predator

The MALEVA model has been particularly targeted at and used for multi-agent-based simulation (MABS) applications [22], in various domains such as ecology, ethology, and economy. In multi-agent-based simulations, various elements of the phenomena modeled and their interactions are explicitly modeled and studied. The two examples described in this paper show some facets of design and of potential reuse.

Our first example will define behaviors of situated agents within an ecosystem. First step is thus to define a general architecture for situated agents.[6]

4.1 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:

$$sensors \rightarrow behavior \rightarrow effectors$$

and is shown at Figure 8.

4.2 Prey Behavior

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

[5] It corresponds to a notion of *structural composition* as opposed to, or rather *in addition to*, *functional composition* (simple assemblage). Such encapsulation of assemblages of components represents a very powerful abstraction principle. Of course, a composite may provide extra functionalities (and control specifications) at its higher abstraction level, making it a true component on its own. Another example supporting the notion of composite component is the Fractal component model [8].

[6] Note that for other applications, e.g., micro-simulation [6], agents are not necessarily situated (within an environment) and thus do not use any sensor/effector. Other applications agents could also use inter-agent communication (ACL) modules.

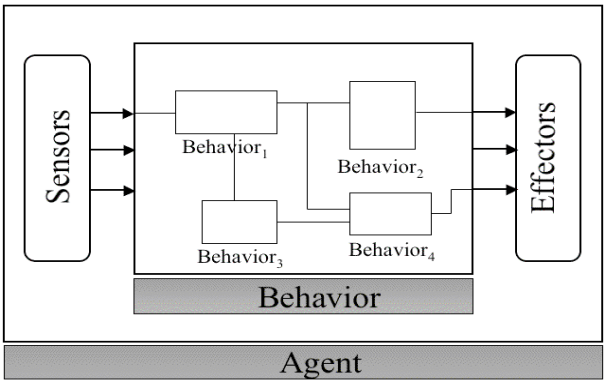

Fig. 8. General architecture of a situated agent

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 will compose them, to represent the following agent behaviors: `Prey` and `Predator`.

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`.

4.3 Control Components

The `Switch` control component reifies the standard conditional structure into a special kind of primitive component.[7] 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)

The architecture of the `Prey` behavior follows this pattern and is shown at Figure 9. If a predator has been detected (some data representing the predator loca-

[7] Note that the MALEVA standard library includes other control components, analog to standard control structures (e.g., repeat loop) or synchronization operators (e.g., barrier synchronization) [16]. They will not be described in this paper.

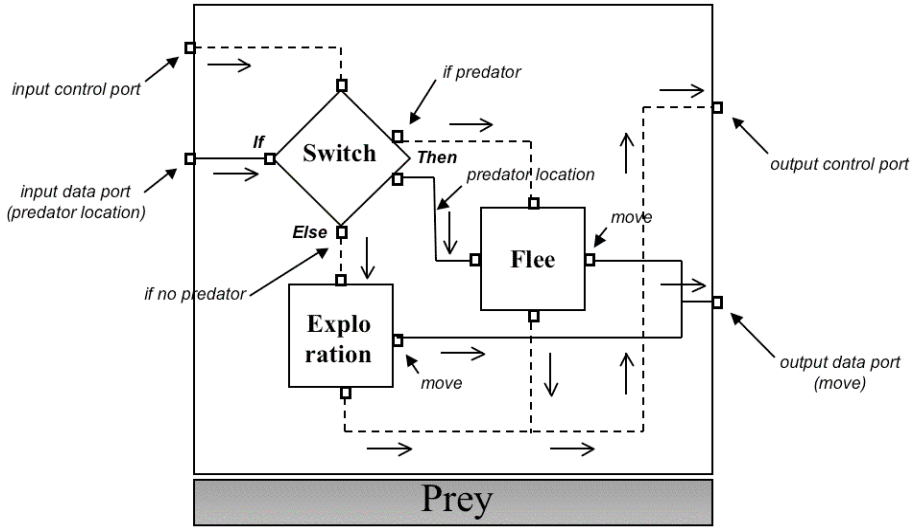

Fig. 9. `Prey` behavior

tion has been received on the input data port),[8] `Switch` transfers 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 is finally transfered 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 sensed (no data received), `Switch` transfers control through its `Else` output control port, which activates `Exploration` behavior. Note that `Exploration` does not need a data input to produce a move data.

4.4 Predator Behavior

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 predator behavior may be defined as a prey behavior (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 `Predator` behavior component as a new composite behavior embedding *as it is* the existing `Prey` behavior component (see the result in Figure 10). Note that in our current design, hunger (predation) has priority over fear (fleeing), as `Prey` is activated by `Predator`. Other combinations could be possible.

[8] 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 8.

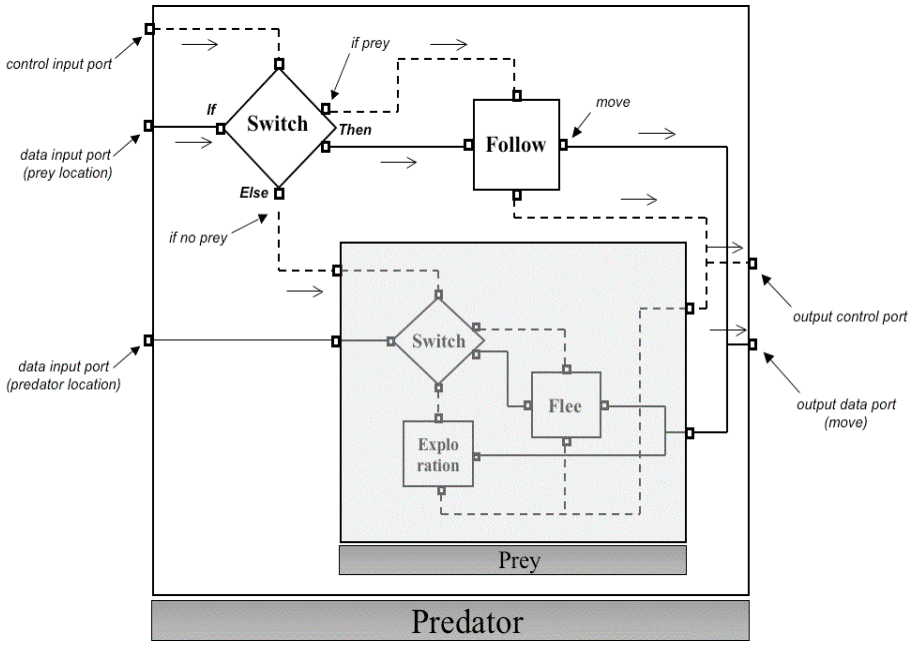

Fig. 10. `Predator` behavior (with `Prey` as a sub-component)

5 A Second 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 [15] of the modeling and simulation of ant colonies for the study of their sociogenesis (Alexis Drogoul's MANTA framework [10]). Various types of ant agents are considered: eggs, larvae, worker ants, queens.[9] In this paper, we focus on the top down design of the behavior of an ant worker.

5.1 The Living Pattern

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 11. It includes 4 sub-behaviors/components: behavior `CheckAgeLimit` (it includes a variable `age`, incremented for each activation step and compared with the agent age limit); behavior `Die`; abstract behavior `Behavior`; 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

[9] The metamorphosis process - from egg to larva and then to ant or queen - leads to the issue of behavior evolution and architectural dynamicity, see Section 8.

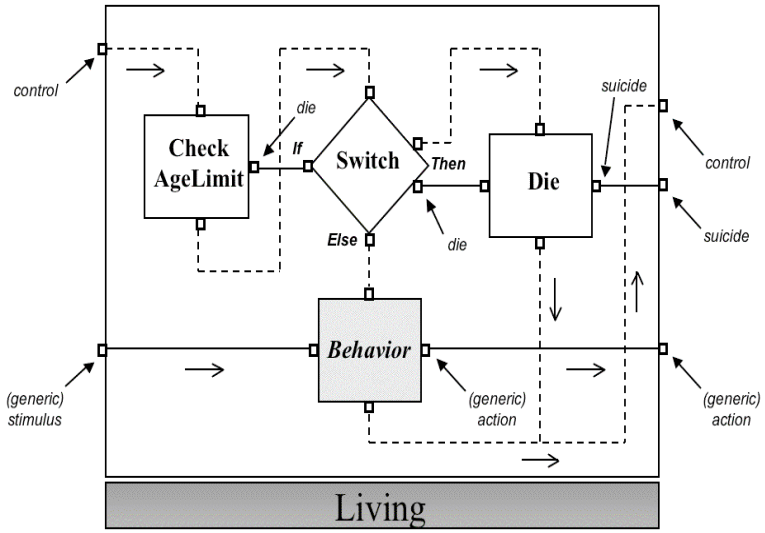

Fig. 11. `Living` abstract behavior

(in practice, it may e.g., remove the agent from the environment). Otherwise, `Behavior` is activated by `Switch`.

This design is a simplified form of a *design pattern* [11].[10] `Living` implements that pattern as some *"mini black-box framework"*, where the *hot spot* is the abstract component `Behavior`. To construct a specific agent behavior, we replace (instanciate) the abstract component `Behavior` with a concrete behavior, e.g., specific to an ant, egg, larva or queen [15].

5.2 The Behavior of an Ant

A worker ant has a relatively complex behavior because its various types of activities: move, pheromone following, egg carrying, egg caring. It is simplified in this paper. First, we instanciate `Living` into a concrete behavior specific to ants, named `Ant`. In practice, `Behavior` is replaced by a concrete behavior, named `AntActivity`.[11] Result is shown at Figure 12.

We now define the specific behavior of the ant, named `AntActivity`, shown at Figure 13. The ant explores its surroundings through a random movement (`Exploration` behavior), unless it perceives some stimulus (`ManageStimulus`

[10] We have identified others, e.g., *"exploration unless perception"*, used by the `Prey` behavior, in Section 4, and which will be reused for the `AntActivity` ant internal behavior, in Section 5.2. A further discussion about MALEVA design patterns may be found in [15].

[11] 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).

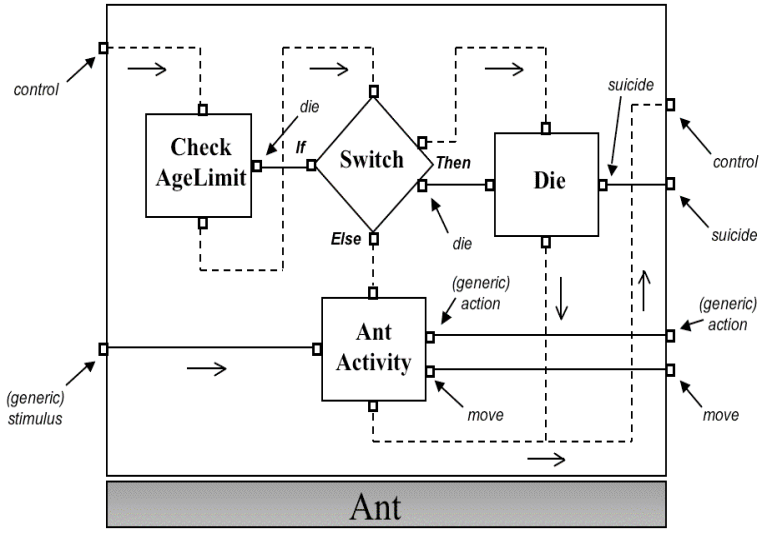

Fig. 12. `Ant` behavior

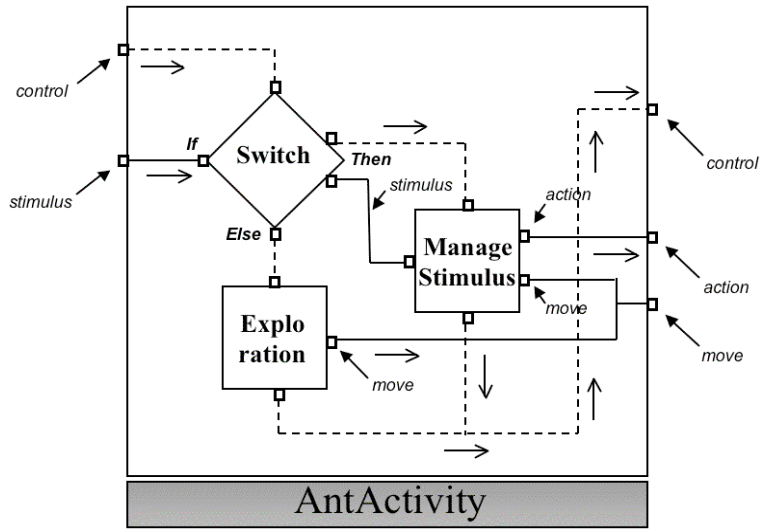

Fig. 13. `AntActivity` behavior: first level of decomposition of `Ant` behavior

behavior). Thus, `AntActivity` reuses the *"exploration unless perception"* pattern, already used for `Prey` and `Predator` behaviors (see Figures 9 and 10).

Because of space limitation, we do not detail the design of `ManageStimulus` behavior (follow gradient and take action when reaching a local maximum).

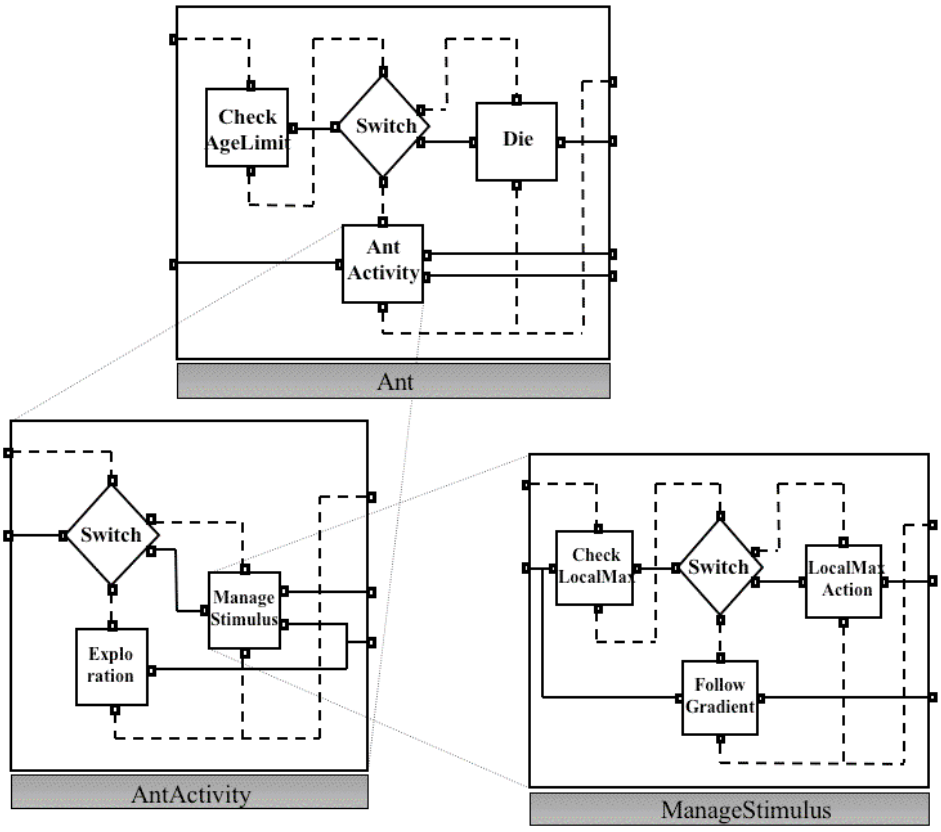

Fig. 14. `Ant` behavior: complete decomposition

The complete architecture of the ant behavior (including 3 levels and 14 components) is shown at Figure 14.

6 Implementation

6.1 Evolution of Implementation

The MALEVA component model and its associated prototype CASE tool have been implemented successively in three versions and languages: Delphi, Java [15], and C++ [16].

The Java-based reimplementation 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, 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.

6.2 Modes of Activation and Scheduling

At the level of the general scheduler, two alternative modes (or approaches) 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 behaviours 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., [14] and [16] for more discussion).

6.3 From Methods to Components

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 agents.

An interesting feature of CGraphGen [16] is the importation of actual Java code and its reification into MALEVA components. The granularity considered is a Java method. After specifying the class and 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 both data-flow and control-flow between components, and the creation of composite components.

7 Related Work

In addition to the agent architectures already discussed in Section 2, we now quickly refer to a few additional related works, still focusing on the agent architectures offering some modular or compositional support at the level of one agent. See also, e.g., [2], for a recent more general survey of languages, architectures and platforms for multi-agent systems.

Like MALEVA, JAF (Java Agent Framework [13]), also based on JavaBeans, uses components to decompose behaviours of agents. JAF does not explicitly separate control flow from data flow. But it proposes some interesting matchmaking 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 behaviour decomposition. JAF decomposition appears at a relatively high level, whereas MALEVA promotes a fine grain behaviour decomposition, and its management through explicit control.

The MaSE methodology [9] includes a modular representation of agent behaviours as sets of concurrent tasks. Each task is described as a finite state automaton and is implemented as an object with a separate thread. A task can communicate with other tasks, inside the same agent, or with a task of another agent, through event communication. A first difference with MALEVA is that the implementation of MaSE concurrent tasks does not use components with explicit input and output ports. Another difference is that MALEVA provides more explicit control of activation, whereas MaSE concurrent tasks rely partly on some implicit control (inter-tasks implicit concurrency and synchronous message reception discipline). That said, as the MaSE methodology is actually very general, we could imagine using several of the MaSE steps to produce MALEVA components.

The DESIRE methodology and component model [3] 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/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.

8 Further Issues and Future Directions

A first issue, for our current component architecture, is that simulation designers must design activation models (control flow) from a relatively low-level perspective, with explicit manipulation of connexions. Abstract components and their related design patterns help at capitalizing and reusing experiences. While several patterns and reusable abstracts components have been tested and documented in various experiments (see, e.g., in Section 5 and also in [15]), we are still far from a complete library of such reusable activation patterns. We wish to provide the designers with several types of libraries: behavior components (e.g., `Exploration`) ; abstract components, e.g., `Living` ; control components, e.g., `Switch` ; and "system" components, e.g., for perception (sensors), action (effectors), inter-agent communication, migration. In addition, 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 analyse existing designs and to create new ones.

A second issue is that the experience with the specification of control through connexions shows 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 the recursive design of an ant behavior in Section 5). 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) [17] could allow the concise representation of complex activation patterns. The idea is somehow analog to coordination languages, but for very fine grained components. The starting point is to model data used for control (e.g., presence of prey, of predator, of pheromone...) as channels and synchronize activity of behaviors on them. The result is a compact term to express a control graph analog to the example of prey and predator (in Section 4):

$$isPrey.Follow \;||\; isPredator.Flee \;||\; (isNoPrey.Exploration + isNoPredator.Exploration)$$

where *isPrey*, *isPredator*, *isNoPrey* and *isNoPredator* are channels, connected to the sensors of the agent; and *Follow*, *Flee*, and *Exploration* are processes representing behaviors. Such formal characterization would also allow the semantic analysis of such specifications, for example through model checking.

A third issue is the dynamicity of behaviors. An example of modeling is the metamorphosis process of ants (egg, larva, ant), introduced in Section 5. Current implementation strategy relies on a specific meta-component to manage the reconfiguration and reassemblage of behaviors. We are currently considering using a higher level mechanism, based on concepts of configurations, roles and policies, such as [12]. Last, to allow the dynamicity of formalisms for activation patterns, we are considering models of process algebras supporting dynamicity and channel name passing, such as the Pi-calculus [18].

9 Conclusion

In this paper, we presented some experience in using a component model to design and implement agents. This model is relatively original in the explicit management of activation control through control ports and connexions, by applying the concept of component to the specification of control. Several experiments illustrate how MALEVA can support various forms of potential reuse through: structural composition of behaviors, abstract components and design patterns, and specialization of intra-agent scheduling policies (that latter issue is discussed in [6]).

Considering rationales for agent architectures, we believe that there is no ultimate *best* agent architecture, as it depends on the application domain and requirements. General purpose (also named hybrid) architectures, like InteRRaP, which attempt at reconciling both cognitive and reactive architectures, turn out to be powerful, but also complex. On the contrary, our architecture focuses on a lower-level agent component model with a fine-grained control. It was initially

more targeted at reactive agent models for multi-agent simulation, but we believe that the MALEVA component model is more general, the issue being more in providing sufficiently rich libraries of components and abstract architectures, supporting the types of architectures and applications targeted (e.g., interaction protocols for e-commerce, reasoning components for rational/cognitive agents, etc.). More generally speaking, we believe that some features of our architecture 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.

Acknowledgement

We would like to thank Marc Lhuillier, Alexandre Guillemet and Grégory Haïk, for their contribution to the MALEVA project.

References

1. F. Bellifemine, A. Poggi, G. Rimassa, Developing Multi-Agent Systems with a FIPA-compliant Agent Framework, *Software Practice and Experience*, (31): 103–128, 2001.
2. R. Bordini, L. Braubach, M. Dastani, A. El Fallah Seghrouchni, J.J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, A. Ricci, A Survey of Programming Languages and Platforms for Multi-Agent Systems, *Informatica*, 30:33–44, 2006.
3. F. Brazier, B. Dunin-Keplicz, N.Jennings, J. Treur, Formal Specification of Multi-Agent Systems : a Real-World Case, *1st International Conference on Multi-Agent Systems (ICMAS'95)*, San Francisco, CA, USA, MIT Press, 1995, pp. 25–32.
4. F. Brazier, C. Jonker, J. Treur, N. Wijngaards, Compositional Design of a Generic Design Agent, *Design Studies Journal*, (22):439–471, 2001.
5. J.-P. Briot, Composants logiciels et systèmes multi-agents, *Technologies SMA et leur utilisation dans l'industrie*, A. El Fallah-Seghrouchni (ed.), Collection IC2, Hermès/Lavoisier, France, to appear in 2007.
6. J.-P. Briot, T. Meurisse, A Component-based Model of Agent Behaviors for Multi-Agent-based Simulations, *7th International Workshop on Multi-Agent-Based Simulation (MABS'06)*, AAMAS'2006, Japan, May 2006, pp. 183–190.
7. R.A. Brooks, A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.
8. E. Bruneton, T. Coupaye, M. Leclerc, V. Quema, J.-B. Stefani, An Open Component Model and its Support in Java, *7th International Symposium on Component-Based Software Engineering*, No 3054, LNCS, Springer-Verlag, May 2004, pp. 7–22.
9. 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.
10. A. Drogoul, B. Corbara, D. Fresneau, MANTA: Experimental Results on the Emergence of (Artificial) Ant Societies, in *Artificial Societies: the Computer Simulation of Social Life*, N. Gilbert and R. Conte (eds), UCL Press, U.K., 1995.
11. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.

12. G. Grondin, N. Bouraqadi, L. Vercouter, MaDcAr: an Abstract Model for Dynamic and Automatic (Re-)Assembling of Component-Based Applications, *9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'2006)*, No 4063, LNCS, Springer-Verlag, 2006, pp. 360–367.
13. B. Horling, A Reusable Component Architecture for Agent Construction, *Technical Report* No 1998-49, Computer Science Dept., UMASS, MA, USA, October 1998.
14. B. G. Lawson, S. Park, Asynchronous Time Evolution in an Artificial Society Mode, *Journal of Artificial Societies and Social Simulation*, 3(1), 2000.
15. T. Meurisse, J.-P. Briot, Une approche à base de composants pour la conception d'agents, *Journal Technique et Science Informatiques (TSI)*, 20(4):583–602, Hermès/Lavoisier, France, April 2001.
16. T. Meurisse, Simulation multi-agent : du modèle à l'opérationnalisation, *Thèse de doctorat (PhD thesis)*, Université Paris 6, Paris, France, July 2004.
17. R. Milner, *A Calculus for Communicating Systems*, Springer-Verlag, 1982.
18. R. Milner, *Communicating and Mobile Systems: the Pi-Calculus*, Cambridge University Press, 1999.
19. J.P. Müller, M. Pischel, The Agent Architecture InteRRaP: Concept and Application. *Technical Report* RR-93-26, DFKI, Saarbrucken, Germany, 1993.
20. J.P. Müller, Control Architectures for Autonomous and Interacting Agents: A Survey, In *Intelligent Agent Systems: Theoretical and Practical Issues*, No 1209, LNAI, Springer-Verlag, 1997, pp. 1–26.
21. P.-G. Ricordel, Y. Demazeau, Volcano, a Vowels-Oriented Multi-Agent Platform, *2nd International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS'01)*, No 2296, LNCS, Springer-Verlag, 2001, pp. 253–262.
22. S. Moss, P. Davidsson (eds), Multi-Agent-Based Simulation, *2nd International Workshop on Multi-Agent Based Simulation (MABS'2000) - Revised and Additional Papers*, No 1979, LNCS, Springer-Verlag, 2001.
23. M. Shaw, D. Garlan, *Software Architectures: Perspective on an Emerging Discipline*, Prentice Hall, 1996.
24. Y. Shoham, Agent Oriented Programming, *Artificial Intelligence*, 60(1):51–92, 1993.