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

Jean-Pierre Briot[1,3], Thomas Meurisse[2], and Frédéric Peschanski[1]

[1] Laboratoire d'Informatique de Paris 6 (LIP6)
8 rue du Capitaine Scott, 75015 Paris, France
Jean-Pierre.Briot@lip6.fr, Frederic.Peschanski@lip6.fr
[2] Sinovia, 93 rue Henri Rochefort, 91000 Evry, France
Thomas.Meurisse@sinovia.com
[3] 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 component-based model of agents, named MALEVA. In this model, components encapsulate various units of agent behaviors (e.g., follow gradient, flee, reproduce...). Among its specificities, it provides an explicit notion of control flow between 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 case studies in the domain of multi-agent based simulation are presented in this paper. They illustrate the ability of the model to support both a bottom-up and a top-down approach for agent architecture design.

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. 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,
- 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...) and their specific requirements definitely influenced our design decisions.

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. As opposed to most of modular or component-based agent architectures, this component model does not impose a specific architectural style. Moreover, it also applies the principles of components and software composition to the specification of control, through the notions of control ports and control components. From a methodological point of view, the model and its associated tools and implementations have been used in both top-down and bottom-up approaches to agent design. Most experiments have been conducted in the field of multi-agent based simulation.

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 agent behavior. Except for simple reactive behaviors, 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 [18], we tentatively propose a classification for agent architectures from the perspective of *architectural styles*. There is of course no unique typology for agent architectures, and we may find other classifications in the literature (e.g., in [15] or [20]), such as horizontal/vertical or reactive/cognitive/hybrid. 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 [18], a complex architecture (e.g., InteRRaP, see Section 2.3) may juxtapose and combine several architectural/decomposition styles.

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 [19] (see Figure 1).

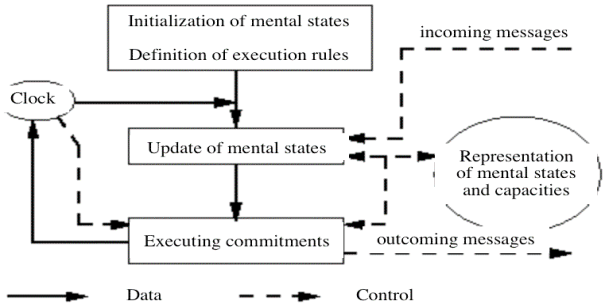

Fig. 1. AOP Architecture

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 (perception, communication, coordination...). An example is the VOLCANO architecture [16], which follows the decomposition guidelines of the Voyelles methodology [7] 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 (see left side of Figure 2). Note that the designer needs also to implement inter-bricks adaptors, respectively AE, AI, AO, EI, EO and IO.

Another example is the generic model of agent architecture (Generic Agent Model: GAM), designed on top of the DESIRE agent component model [2]. It includes a set of components (interaction management, information maintenance..., see right side of Figure 2), each dedicated to a specific type of processing. The GAM generic model (also a framework) has been instantiated to model (retro-engineer) various agent architectures, such as BDI, and ARCHON.

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 [14] (see Figure 3).[4]

[4] Note that the internal architecture of each level follows the same model, based on planning. Also note that, as for software architectures [18], a complex architecture (such as InteRRaP) may juxtapose and combine several architectural styles.

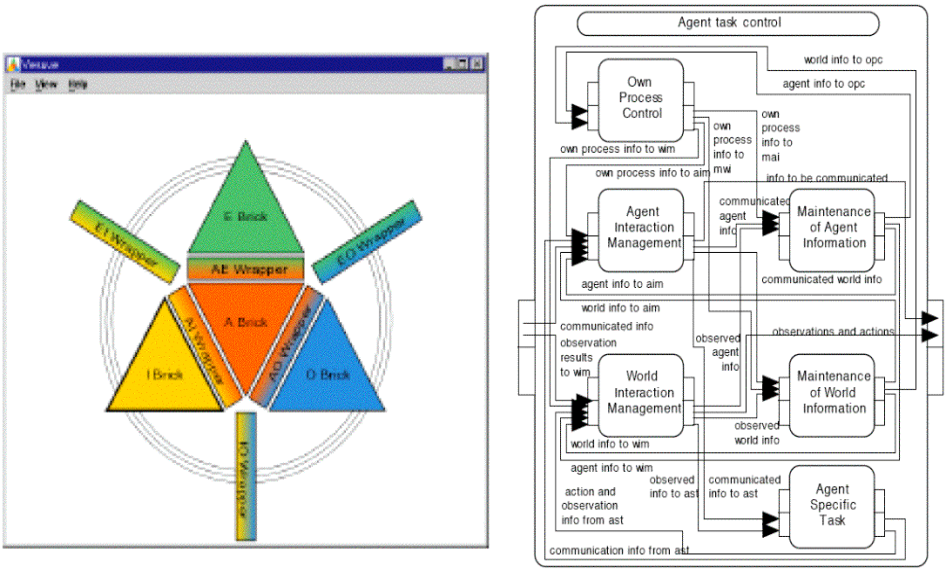

Fig. 2. VOLCANO (left) and DESIRE GAM (right) architectures

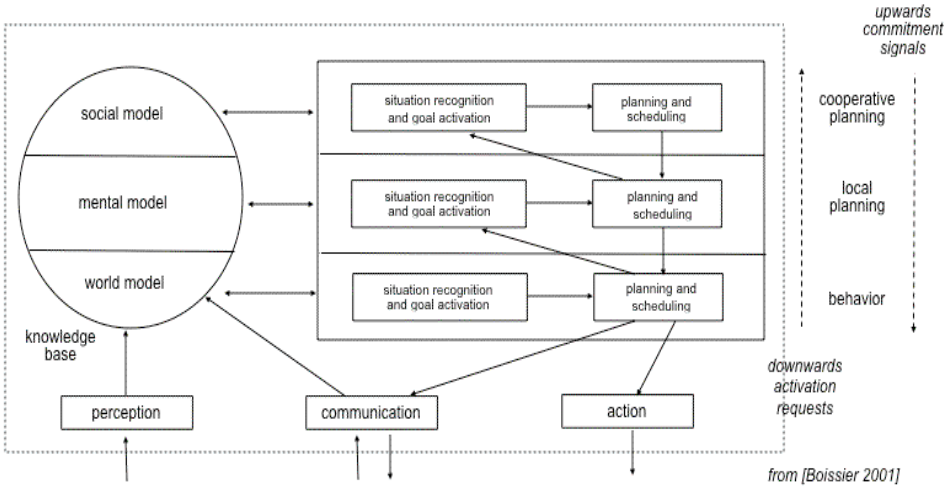

Fig. 3. InteRRaP architecture

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 [5], 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 4). 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).

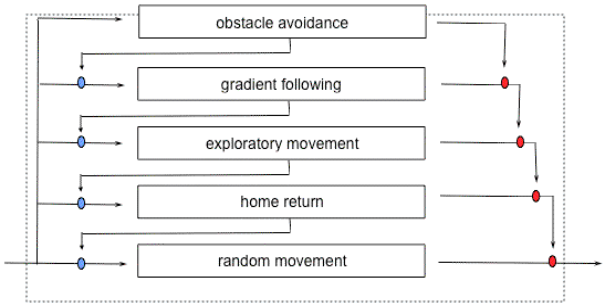

Fig. 4. Subsumption architecture

2.5 Discussion

The architectures that we surveyed are generally rather stable and generic, although usually more tailored at a specific model of agent (e.g., cognitive collaborative agent for the InteRRaP architecture, situated agent or robot for subsomption architecture). But these architectures are often not very flexible. In practice 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...),[5] neither classical component models (e.g., JavaBeans). The VOLCANO architecture clearly separates the bricks, but in order to replace one brick by another we are forced to re-implement the corresponding adaptors. To make coupling more flexible, JAF (Java Agent Framework)

[5] 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.

[9] 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.

More generally speaking, the existence of an architecture constrains the possible combinations of components, which is actually the very role of an architecture. An architecture such as VOLCANO is a framework with abstract components (named *hot spots* in the framework terminology) to be instantiated, but also with adaptors to implement. The subsumption architecture actually represents an abstract model of architecture, instantiated for a specific robot and objective (e.g., see Figure 4). It is concise but also hard 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 very 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. We then must replace the fixed hierarchical control architecture of the subsumption architecture by something more open.

In the model we propose, control is made explicit. And, by keeping in line with a component model, control is specified through control ports and connexions (see Section 3.1), in order to represent arbitrary patterns of control flow.

3 The MALEVA Model

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, we make a distinction 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 a control input port. When the execution of the behavior is completed, the activation signal is passed to the control output 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.

	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

3.2 An Introductory Example

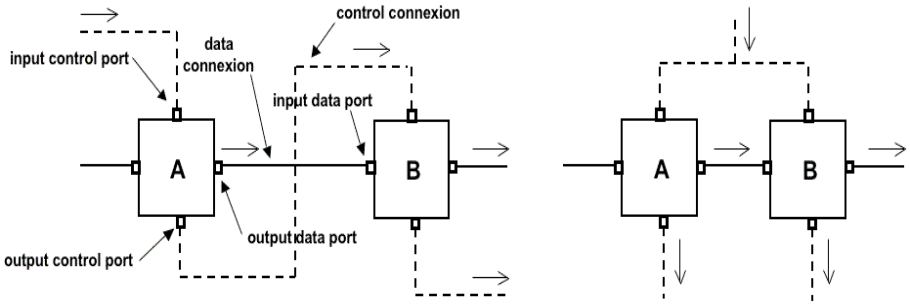

Fig. 5. Sequential and concurrent activation of two components

Left side of Figure 5 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 all figures, data flow connexions are shown in solid lines, and control flow connexions in dotted lines.

The right side of the figure 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* [18] 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.[6] 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 [3].[7]

3.3 Designing Agent Behaviors

Designing agents for an application consists 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*,[8] as the encapsulation of a composition (assembly) of components.

4 A First Case Study: Design of Prey and Predator

The MALEVA model has been particularly targeted and used for multi-agent-based simulation (MABS) applications [17] (in various domains such as ecology, ethology, economy...). In multi-agent-based simulations, various elements of the phenomena modeled and their interactions are explicitly modeled and studied.

Our first case study will define behaviors of situated agents within an ecosystem. First step is thus to define a general architecture for situated agents.[9]

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* → *behavior* → *effectors* and is shown at Figure 6.

[6] An additional dimension, the model of activation of components, which could be asynchronous or synchronous, will be briefly addressed in Section 6.1.

[7] For instance, [10] shows that results of simulations can be found biased in cases where the scheduling of the actions within an agent remains deterministic.

[8] We believe that the notion of composite component is important and useful. It corresponds to a notion of *structural composition* as opposed to, or rather *in addition to*, *functional composition* (simple assembly). Encapsulation and hierarchization offered by the notion of composite component are useful to help at managing complexity. Another example supporting the notion of composite components is the Fractal general-purpose component model [6].

[9] Note that for other applications, e.g., micro-simulation [3], 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 modules.

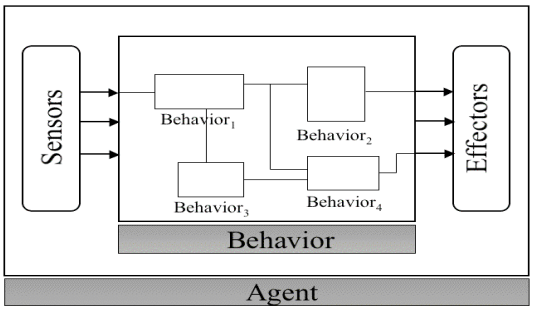

Fig. 6. General architecture of a situated agent

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, representing the basic behaviors of preys and predators, that we name: `Flee` (fleeing a predator), `Follow` (following a prey), and `RandomMove` (random move for exploration, 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 moves randomly (basic exploratory behavior). Thus, we construct the `Prey` behavior as the composition of the following three components: `Flee`, `RandomMove`, 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.[10] 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 has been 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)
end if

---

[10] 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). They will not be described in this paper.

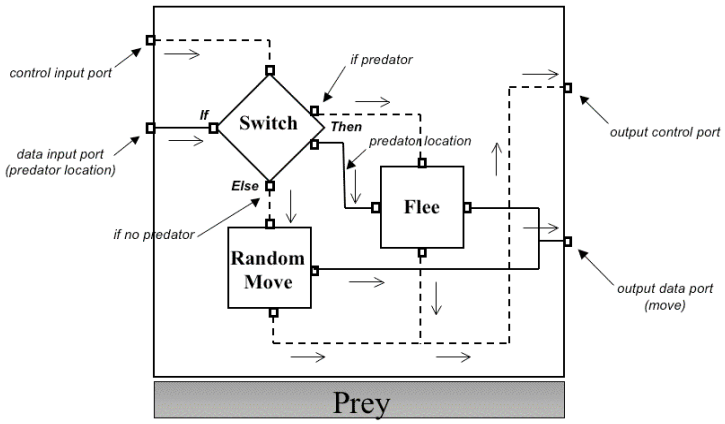

Fig. 7. Architecture of a prey

The architecture of the prey behavior follows this pattern and is shown at Figure 7. If a predator has been detected (some data representing the predator location has been received on the input data port),[11] `Switch` transfers control through its `Then` control output 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 `RandomMove` behavior. Note that `RandomMove` does not need a data input to be able 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 moves randomly if he does not sense any other agent. The predator behavior may be defined as a prey behavior (it flees other predators and otherwise carries out a random movement), to which is added a behavior of predation (it follows the preys). 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 8). Note that in that our current design, hunger (predation) has priority over fear (fleeing), as `Prey` is activated by `Predator`. Other combinations could be possible.

[11] 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, as shown at Figure 6.

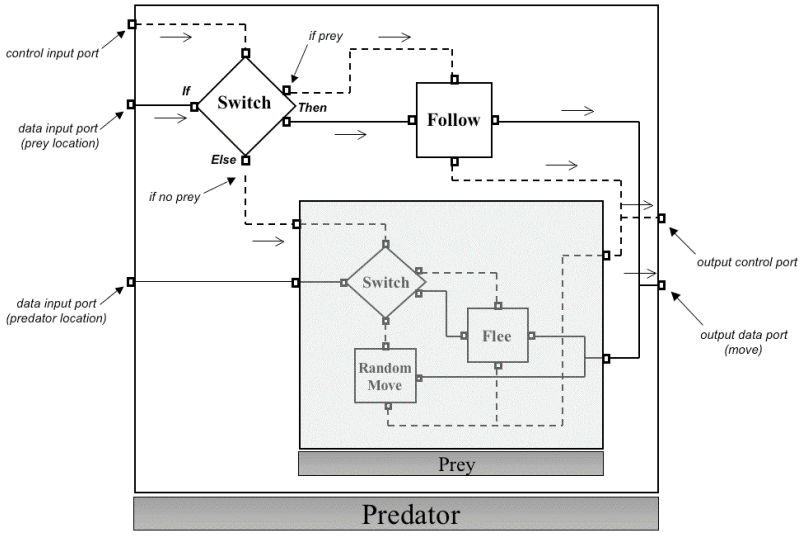

Fig. 8. Architecture of a predator (with prey as a sub-component)

5 A Second Case Study: Top-down Design of Ants

The second case study will provide a flavor of the reengineering in MALEVA of a seminal work on sociogenesis of ant colonies (Alexis Drogoul's MANTA framework [8]). Because of space limitation, we do not detail the different types of agents involved (ant worker, egg, larva, queen), neither the different steps and components of the top down design of the behavior of an ant worker.

The behavior of an ant worker (named `Worker Agent`) is based on a pattern of an *aging* agent (an agent dies when having reached its limit age). The sub-component `Maturing` manages the incrementation of the age and the age limit test. The first level of decomposition, named `Behavior1`, states that random move (exploration) is the default action unless a stimulus (food, pheromone...) is sensed. The second level of decomposition, named `Behavior2`, states that the ant worker will follow the gradient of the stimulus until it reaches the local maximum and then produces an action, depending on the type of stimulus (food, pheromone...). The architecture is summarized at Figure 9.

Note that this case study followed a top down approach, as opposed to previous one (Section 4). The case study is further detailed in [11], which also includes identification of some design patterns. Last, ant metamorphosis (from an egg, to a larva, to a worker ant) was modeled as a dynamic change of behaviors [4].

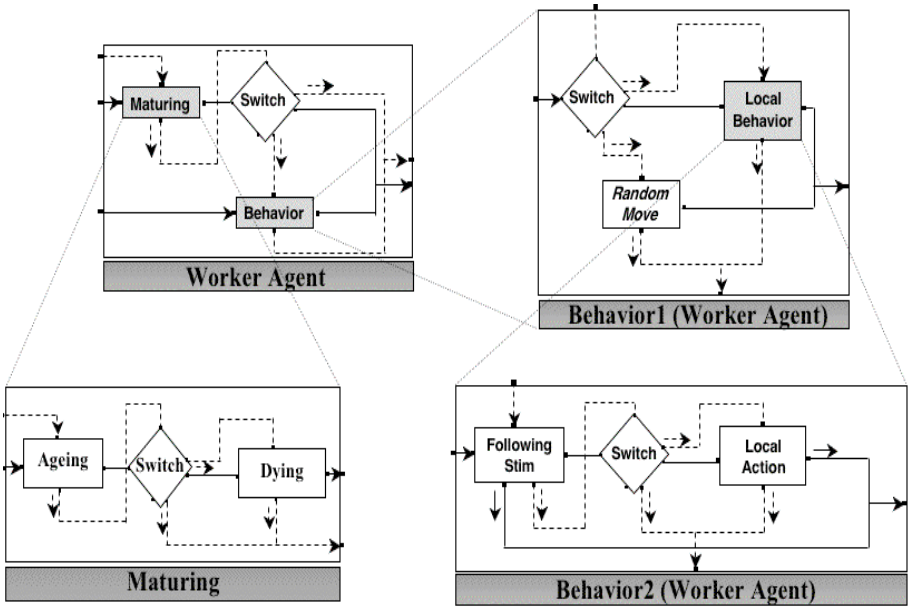

Fig. 9. Behavior of an ant worker, involving 4 components

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 [11], and C++ [12].

The Java-based implementation improved the possibility of architectural dynamic evolution,[12] which turned out to be useful to model evolving behaviors, e.g., ant metamorphosis (as mentioned in Section 5). Typing the ports was also added and turned out to be useful for verifying some compatibility between components. Then, sub-typing helps at defining more generic components. Java also supports inspecting various information about a component, thanks to its introspection facilities (API and tools). Thus, the designer of agents can easily query a component to obtain its internal information.

The Java implementation, based on JavaBeans, also gave opportunity to compare our MALEVA prototype component model with an industrial component model such as JavaBeans (both being light-weight). Note that the JavaBeans model conforms to a publish/subscribe communication model, *but* the implementation still relies on standard method call. In our Java-based implementation of MALEVA, a mailbox (FIFO queue of messages) is associated to each input data

[12] The Delphi implementation imposed recompilation, because of the absence of a dynamic code loader.

port and to the input control port, in order to decouple data transfer and actual activation.

At the level of the general scheduler, two alternative models of activation have been implemented: an *asynchronous* model and a *synchronous* model. The asynchronous mode is more efficient and the expression of control is more precise. Within each agent, the control connexions ensure an intra-agent synchronization at wish. Meanwhile, unless the designer also uses explicit control connexions between agents, the different agents may not be synchronized (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 forces a global synchronization. The choice between the two models depends on the requirements for the application (see e.g., [10] and [11] for more discussions).

6.2 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 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 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 Further Issues and Future Directions

A first issue, for our current component architecture is that simulation designers must design activation models from a relatively low-level perspective, with explicit manipulation of connexions. Abstract components and their related design patterns (see e.g., [11]) help at capitalizing and reusing experiences.

A second issue, is that experience with 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 [13] 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):

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

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

8 Conclusion

In this paper, we presented some experience in using a component model to design and implement agents. This model is relatively original in that it aims at decomposing the inner behavior of agents, and not just its general architecture. It is also original in the explicit management of activation control through control ports and connexions, by applying the concept of component also to the specification of control. As opposed to most of modular or component-based agent architectures, our component model does not impose a specific architectural style nor a bottom up or top down approach. Experiments showed that its characteristics help in improving genericity of components, and in rationalizing the control of intra-agent behavior scheduling, an important issue for simulation applications.

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 is simple, with a fine-grained control, and more targeted at reactive agent models for multi-agent simulation. Thus, it may not be optimal for all applications, for instance to construct e-business applications based on negotiation protocols. But 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.

Acknowledgments: 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, and G. Rimassa, Developing Multi-Agent Systems with a FIPA-compliant Agent Framework, *Software Practice and Experience*, (31):103–128, 2001.
2. F. Brazier, C. Jonker, J. Treur, and N. Wijngaards, Compositional Design of a Generic Design Agent, *Design Studies Journal*, (22):439–471, 2001.
3. J.-P. Briot and T. Meurisse, A Component-based Model of Agent Behaviors for Multi-Agent-based Simulations, *Pre-Proceedings of the 7th International Workshop on Multi-Agent-Based Simulation (MABS'06)*, AAMAS'2006, Japan, May 2006.
4. J.-P. Briot, T. Meurisse, and F. Peschanski, Une expérience de conception et de composition de comportements d'agents à l'aide de composants, To appear in Numéro spécial on Composants et Systèmes Multi-Agents, O. Boissier (ed.), *Journal L'Objet*, Hermes/Lavoisier, Paris, 2006.
5. R.A. Brooks, A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.
6. E. Bruneton, T. Coupaye, M. Leclerc, V. Quema, and 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.
7. Y. Demazeau, From Interactions to Collective Behaviour in Agent-Based Systems, *1st European Conference on Cognitive Science*, St-Malo, France, 1995, pp. 117–132.
8. A. Drogoul, B. Corbara, and 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.
9. B. C. Horling, A Reusable Component Architecture for Agent Construction, *Technical Report* No 1998-49, Computer Science Dept., UMASS, MA, USA, October 1998.
10. B. G. Lawson and S. Park, Asynchronous Time Evolution in an Artificial Society Mode, *Journal of Artificial Societies and Social Simulation*, 3(1), 2000.
11. T. Meurisse and J.-P. Briot, Une approche à base de composants pour la conception d'agents. *Journal Technique et Science Informatiques (TSI)*, 20(4):583–602, Hermes/Lavoisier, Paris, April 2001.
12. T. Meurisse, Simulation multi-agent : du modèle à l'opérationnalisation, *Thèse de doctorat (PhD thesis)*, Université Paris 6, France, July 2004.
13. R. Milner, *A Calculus for Communicating Systems*, Springer-Verlag, 1982.
14. J.P. Müller and M. Pischel, The Agent Architecture InteRRaP: Concept and Application. *Technical Report* RR-93-26, DFKI, Saarbrucken, Germany, 1993.
15. 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.
16. P.-G. Ricordel and Y. Demazeau, Volcano, a Vowels-Oriented Multi-Agent Platform, *Revised Papers from the 2nd International Workshop of Central and Eastern Europe on Multi-Agent Systems*, No 2296, LNCS, Springer-Verlag, 2001, pp. 253–262.
17. S. Moss and 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.
18. M. Shaw and D. Garlan, *Software Architectures: Perspective on an Emerging Discipline*, Prentice Hall, 1996.
19. Y. Shoham, *Agent Oriented Programming*, Artificial Intelligence, 60(1):51–92, 1993.
20. M. Wooldridge and N.R. Jennings, Agent Theories, Architectures, and Languages: a Survey, *Workshop on Agent Theories, Architectures, and Languages on Intelligent Agents (ATAL'94/ECAI'94)*, Springer-Verlag, 1995, pp. 1–39.