Component-Based Agent Architectures to Build Dedicated Agent Frameworks

Victor NOËL, Jean-Paul ARCANGELI and Marie-Pierre GLEIZES
Université de Toulouse, Institut de Recherche en Informatique de Toulouse,
118, route de Narbonne, 31 062 Toulouse Cedex, France
{victor.noel,jean-paul.arcangeli,marie-pierre.gleizes}@irit.fr

Abstract

In order to fill the gap between design and implementation of multi-agent systems, we introduce an intermediary phase in the development process that consists in implementing application-specific "models of agents". Their realisation using multi-level software architectures relies on software components for separation of concerns and complies with agent-oriented requirements. Then, a dedicated agent framework is derived from the architecture, which can be more or less refined depending on the application requirements and the ability and skills of the targeted multi-agent developer. Finally, the agents of the multi-agent system are implemented using the framework by programming behaviours that make use of their respective model. We present MAY (MAKE AGENTS YOURSELF), a tool integrated into Eclipse that supports the description of agent architectures and their transformation into executable agents implementable in JAVA. An example illustrates the development process by following our approach and using MAY.

1 Introduction

Multi-agent systems (MAS) are complex systems composed of interacting agents that are possibly heterogeneous. Their development requires appropriate tools (methodologies, development environments, programming languages...) in order to cope with this complexity. The more the development tools fit with the considered application, the more developers can concentrate on "what" their system does instead of "how" it works: development is facilitated and the produced software is easier to maintain.

Many platforms and programming frameworks have been proposed (Jade, Jadex, AgentScape, Jason, MadKit, CArtAgO, NetLogo, Repast...), some of them specifically for agent-oriented methodologies (Tropos, Prometheus, Ingenias, Passi, SODA, IODA...), or theoretical models (MASQ, A&A...). When implementing MAS, developers must fill the gap between the framework they chose and the agents of their application (most often resulting from an agent-oriented design methodology). Indeed, the main problem is how the agent concepts ("agent", "interaction", "communication", "adaptation"...), which are useful for the application, are defined and provided by the chosen platform and its programming model. This often results in the programming framework being tightly coupled to the used design methodology, or worse, in agents that are designed so that they can target an existing agent programming framework. But in reality, depending on the application, different "models of agents" are needed, whether by the way they perceive, act, communicate or by the way they are internally structured: an agent controlling a mobile robot tracking targets using a camera does not need the same capabilities and interaction mechanisms with its environment as a grid agent that solves constraint-based problems in a distributed computing context and moves between nodes of the network.

For example, programming the behaviours of like-ant agents that use coordination by stigmergy demands high-level mechanisms such as moving, depositing or scenting pheromones in a virtual space. If the development relies on a platform like Jade [Bellifemine *et al.*, 1999], stigmergy must be implemented through message passing, which is inadequate in this case. If the chosen platform is NetLogo [Wilensky, 1999], implementation of stigmergy may be facilitated (through the use of the "patches") but no advanced mechanism for direct communication is available. In both case, the implementation is complicated, the volume of the code is increased, reuse is made difficult and the gap between the "model of agent" proposed by the platform and those of the applications only burdens the developer with different concerns mixed together.

Bridging the gap between analysis and implementation is a key challenge for the MAS community [Bordini *et al.*, 2005]. To cross it, instead of proposing another more or less generic agent programming framework, we propose to produce a dedicated framework implementing the application-specific models of agents that will next support the implementation of the agents. More concretely, the resulting framework is a programming toolbox (for example in our case JAVA classes) that final developers can use to program the agents using high-level mechanisms instead of (re-)programming them with low-level ones. This approach clearly separates the level of the expression of "how" agents do (by developing the dedicated framework implementing the models of agents) from "what" they do (by developing the MAS using the framework).

Component-based software engineering aims at building software by composing independently developed and

reusable pieces of software, with well-specified interfaces and dependencies, called software components [Szyperski *et al.*, 2002]. It promotes separation of concerns and definition of clear, composable, and reusable abstractions. In this work, we aim at assisting developers in the use of component-based technologies for MAS development while taking into account agent-oriented concerns such as autonomy, interactions or adaptation. We thus propose to make possible the realisation of models of agents by designing software architectures (of agents) composed of software components. Then, the architecture is transformed into usable and executable code. Finally, by implementing some of the components of the architecture, a more or less refined and specialised framework can be generated. So, the level of abstraction and expressiveness of the framework can be adjusted depending on the expertise of the targeted user of the framework (*i.e.* the developer of the MAS).

This work has been used for biological simulation [Bonjean *et al.*, 2009], dynamic ontology construction [Sellami *et al.*, 2009] and naval surveillance [Georgé *et al.*, 2009], but also integrated into a methodology [Rougemaille *et al.*, 2009]. It is currently applied in French national projects on distributed robotics for crisis management (ROSACE[1]) and multi-agent simulation for environmental norms impact assessment (MAELIA[2]).

Currently, only the agent level is addressed from an architectural point of view, but our next step is to consider environments and mechanisms for interaction as first class entities (components) in the spirit of [Weyns *et al.*, 2006].

The paper is organised as follows. In Sect. 2, our solution to describe component-based agent architectures as a mean to realise models of agents dedicated to an application is presented. We show the μADL description language and detail how agent-oriented concerns are taken into account with our solution. Then Sect. 3 presents the translation from μADL to a class-based object-oriented programming language to make the implementation of the architectures possible. It is followed in Sect. 4 by a presentation of MAY, a set of model-based tools with adequate editors and generators for JAVA. In Sect. 5, some examples are presented and one is developed to show the advantages of the solution and illustrate a possible development process. Finally, related works are discussed in Sect. 6, then conclusions and some interesting perspectives are presented in Sect. 7.

2 Realising Models of Agents with Component-Based Architectures

What is the simplest way to implement models of agents to be as expressive and safe as possible, by reusing existing mechanisms, and at the same time complying with the requirements of MAS (complexity, autonomy, interactions in a system, self-adaptation, diversity of mechanisms...)? To answer this question, in line with previous works on the Agent$^\varphi$ approach [Leriche and Arcangeli, 2008], we propose to realise models of agents by: 1) programming, composing and reusing existing components aggregated in an architecture; 2) using constructs adapted to MAS (at component and architecture level).

We thus propose the μADL description language to describe components and architectures. In μADL, agent architectures are described using two orthogonal constructs to separate concerns:

1. their separation in components, at description and implementation;

2. their separation in two levels, operational and applicative, corresponding to the separation between what will be part of the dedicated agent framework and what will need to be implemented to program the agents.

They are orthogonal in the sense that the first enables separation of concerns between the different developers of the different components, while the second enables separation of concerns between the creator of the agent architecture and its user.

In this section, the μADL language is presented, then we show how it answers software engineering requirement such as re-usability, evolution and safety and agent-oriented software engineering requirements such as autonomy, interaction and self-adaptation.

2.1 μADL: Components and Architectures

Components. Software components are a way to provide different mechanisms in a reusable and composable way. For agents, examples of components are those for the control loop, such as lifecycle; for interaction with the environment, such as communication (message passing...), sensors and actuators (stigmergy, wheels, vision...) or scheduling and distribution; and of course more logical components such as behaviour or knowledge management.

Our proposition introduces a simple component model where components are specified using a description (what is achieved by the component) and can have several implementations (how it is achieved). Furthermore, they are kept decoupled from any architecture and are the units of adaptation.

More technically, a component has a name and is living in a flat global namespace (hierarchical naming based on the Internet domain names as in JAVA). A component description provides operations, which are identified by a name, a return type and typed parameters. It also has requirements over the architecture into which it will be used: currently a component can require that some operations must be present or that it must be able to change at runtime the implementation of another component of the architecture (see Sect. 2.3). Finally, it can specify a persistent (typed) state that must be kept when its implementation is dynamically changed.

In all these descriptions, to stay simple because this is not the objective here, we consider that components and operations names as well as types denotes the specification of their semantics (like with interface and methods names and types in JAVA).

The description of one of the example components from Sect. 5 is shown Fig. 1.

Architectures. An agent architecture realises a model of agent by connecting components. It is separated in two levels: operational and applicative. Typically, the operational level contains mechanisms such as lifecycle, message pass-

[1] http://www.irit.fr/Rosace,737

[2] http://www.iaai-maelia.eu/

```
component Stigmergy {
 package components.ants

 provided deposit(quantity: Int)
 provided scent(): Int

 required myPosition(): Position
}
```

Figure 1: Component Description in μADL

```
model SimpleAgent {
 package my.archs.simple

 operational {
  Message
  LifeCycle
 }
 application {
  changeable Behaviour
 }
 visibility {
  external receive(m: Msg)
  application send(a: Agent, m: Msg)
  application me(): Agent
  application suicide()
 }
}
```

Figure 2: Architecture Description in μADL

ing, adaptation, sensors and actuators; and the applicative level assembles more logical concerns such as behaviour, knowledge representation or interpretation of perceptions.

In each of the two levels, used components are referenced by their name. It must be noted that contrary to most of component models, components operations are not connected together by hand but implicitly by the architecture (conflicts are not currently handled, aliasing of operations is a possible solution). In the architecture, it is also specified if components are replaceable (with the specific construct `changeable`) at runtime and if some of the operations of the components are available from outside of the agent (with the specific construct `external`).

The description of an architecture derived from the example from Sect. 5 is shown Fig. 2. This example is here to illustrate the possibilities of the language, obviously the interest of our proposition is to realise more complex and dedicated model of agents.

2.2 General Software Engineering Concerns

Benefiting from the advantages of component-based architectures, we are able to check that every required operation is provided by exactly one component in the architecture, i.e. that the architecture is valid. So, when implementing components, the required operations are always provided.

Moreover, components enable adaptation at development time (software evolution) by choosing different component implementations for different needs. The simplest example is to use the same agent architecture for different behaviours, but more interesting application of static adaptation is for prototyping (simplified implementation of components), simulation (before deployment on real hardware: only the operational components implementations change), debugging (components with and without tracing)...

2.3 Agent-Oriented Software Engineering Concerns

We now show briefly how our proposition complies to agent-specific concerns. It is very important to understand that we do not propose a generic solution for the following concerns: we aim at simplifying the implementation of dedicated solutions and their composition with other concerns. Future work will be focused on exploiting and improving this part of our proposition.

Self-Adaptation. At this point of our work, self-adaptation relies on the replacement of component implementations at runtime (dynamic adaptation). A simple case is when several possible behaviours for an agent are managed by the agent itself; but it can also consist in the self-replacement of the reasoning process depending on the context, or in the upgrade of a component with a bug.

Because architectures are valid, self-adaptation is safe and the architecture coherence can't be disturbed as long as component implementations respect their description (which is what they can only do to be compilable).

Interactions. An important point in MAS is that agents are entities interacting together. Thus, it is necessary to build (reusable) components to support the interactions. External operations are the way (even if simple for now) of enabling the implementation of dedicated and adequate interaction mechanisms in a reusable manner. The example developed in Sect. 5 illustrates this aspect.

This part of our proposition is currently being worked on to make it safer and more composable by considering interactions as first-class entities at the development level.

Autonomy. Two aspects of the autonomy of agents are addressed by our approach. First, because the architectures encapsulate the components, agents keep the control of their inside, which is made accessible only through external operations. Besides, autonomy of execution is answered by the fact that the scheduling of the agents is implemented by components: it allows to choose, depending on the model to be realised, how agents are executed and thus what are their lifecycles. For example, a thread could run inside the agent, but on the opposite, scheduling could also be done by an external engine (e.g. for simulation purpose).

3 Generating Dedicated Agent Frameworks

At runtime, an agent is an executable instance of a μADL architecture where all components have been implemented. Programming the agent relies on a dedicated agent framework, which is a set of classes corresponding to the implementation of the architecture and the components of the operational level to form a specialised architecture. Using the framework consists in implementing the remaining components (application level) and plug them in the provided specialised architecture.

Translation. Our approach relies on the following translation from μADL to a class-based object-oriented programming language that can be automatised. This exploits

the type system of the language and well-known design patterns [Gamma *et al.*, 1995].

The basic principle of this translation is that an architecture description is translated to a class that links the implementations of its components without preventing their independence from any architecture.

Technically, each component description is translated to an abstract class with abstract public methods corresponding to the provided operations. This class will be extended to write an implementation of the component description. Each class also has an attribute that gives it access to the required operations, without connecting them directly (Bridge Pattern). To handle persistent state, a class representing a common data structure for the state is generated and abstract methods to get and set the state is added to the component description.

Each agent architecture is translated to a class that contains an attribute for each of its component: it connects them and is responsible of applying the dynamic adaptation (Mediator Pattern). In particular, dynamic replacement of component implementation is automatically done by getting the persistent state of the old component and setting it in the new one. Finally, a class, which represents the agent in the MAS, provides external methods using public methods (Facade Pattern).

To create an agent, the framework provides a factory for to the specialised architecture (Factory Pattern) with some component implementations and holes for the others.

Validity Preservation. Always with safety in mind, the properties and advantages of agent architectures descriptions are preserved at the code level, in particular for adaptation and safe calls to required operations. Our solution does not use error-prone solutions such as XML or string interpretation at runtime to define architectures but relies on the type system of the host language. It insures that, at every step of development, validity of the architecture is preserved (see Sect. 2.2). The agent architecture is set at compile time and the only possible actions to modify it at runtime are architecturally safe.

4 MAKE AGENT YOURSELF

To validate these propositions and experimentally apply them, we developed and released MAY[3] (MAKE AGENT YOURSELF). It provides textual and graphical editors for the μADL language to define components and architectures, a tool to generate the corresponding classes needed to implement agents in JAVA and a factory creator to automatise the creation of dedicated framework. The textual editor also features error highlighting and completion, while the graphical focus on drag-and-drop architecture building. From the user point of view, all these pieces are integrated in the Eclipse IDE around a new file type recognised by Eclipse and its facilities to program JAVA classes.

Technically, descriptions and their transformation to other representations are typical applications of model driven engineering. Thus, we used a meta-model to define the way components and agent architectures are described, model editors to instantiate it and model transformation to generate the code. All of this is relying on the Eclipse modelling ecosystem. The textual editor was realised using TMF Xtext, while the graphical is based on GMF. Constraints to check the validity of the descriptions are developed with TMF Xcheck and code generation with TMF Xpand.

[3] `http://www.irit.fr/MAY`

5 Examples and Development Process

We show now a simple example of communicating agent then complicate it by adding movement and stigmergy in a virtual space. The detail of its implementation can be found on the website of MAY.

Disclaimer. This example is given to briefly illustrate our proposition with known concepts. We build common agent components from scratch to show how our proposition handles MAS requirements. In practice, the idea is to reuse such components when needed. More complex models of agents can be produced with specific mechanisms composed with the presented ones, like: self-organising term agents creating ontological relations [Sellami *et al.*, 2009]; cells that divide, mutate, die and communicate by exchange of molecules [Bonjean *et al.*, 2009]; agents representing real boats for naval surveillance [Georgé *et al.*, 2009]; robot agent executable indistinctly in a simulator or in a real robot (ROSACE); agents for social simulation (MAELIA).

Example. We want to build agents complying to the following model of agent: capable of sending and receiving messages, processing them one at a time to react. The behaviour should be implementable by defining a method that takes a message in input and that can use the send primitive.

First we write a component description for the behaviour: it provides `step(m: Msg)` and requires `send(a: Agent, m: Msg)`; and a component for the lifecycle: it requires `getNextMsg(): Msg` and `step(m: Msg)`. Then, we write an architecture description stating that it must have a component behaviour in its application level and a component lifecycle in its operational level. Using the editors provided by MAY, we are informed that the dependencies `getNextMsg(): Msg` and `send(a: Agent, m: Msg)` are not present in the architecture. We thus write a component description for messages providing both. We add it in the operational level of the architecture: the architecture is now valid.

We generate the corresponding JAVA classes (component description and agent architecture) using MAY. First we implement the lifecycle, see Fig. 3, implementing the definition of the dynamic of the model of agent: in a loop, take a message in the mailbox (blocks if there is no message) and treat it with the behaviour. Then, for the messages component, we first need to write the class `Agent`: its purpose is to encapsulate a reference to an agent architecture to keep it hidden from the user of the framework. To pass a message to an agent from the outside, it needs an entry point: we add `receive(m: Msg)` to the message component description and specify it as external in the architecture. We regenerate the corresponding classes and implement the method `send(Agent ag, Msg m)` with `ag.receive(m)`.

Finally, we create a factory that specialises the agent architecture with the implementation of these two compo-

```
public class LifeCycle
  extends QuasiComponentLifeCycle {
 private boolean alive = true;
  public void start() {
   new Thread(new Runnable() {
    public void run() {
     while (alive) {
      Msg m = architecture().getNextMsg();
      architecture().step(m);
     }}}
   ).start();
 }
 public void suicide() {
  alive = false;
 }
}
```

Figure 3: An implementation of LifeCycle

nents. We can export this set of classes to make a deliverable framework for this model of agent. To program this model of agent, one needs to create a class implementing the behaviour component using the required operations, and create an agent using the factory.

Evolution. Now we want to add movement and stigmergy in a virtual 2D space. This requires to add a component for movement (in 4 directions) and a component for stigmergy (deposit and scenting of pheromones) to the architecture and required operations to the behaviour.

The movement component provides `move(d: Direction)` and `myPosition(): Position`. The stigmergy component provides `deposit(qty: Int)` and `scent(): Int` and requires `myPosition(): Position`. In the architecture only `move(d: Direction)`, `deposit(qty: Int)` and `scent(): Int` are made available to the application level while the behaviour component now requires these 3 operations.

The implementation of the movement component will be given an object shared by agents representing the 2D space, while the stigmergy component will have its own.

Notes. The message and movement components can be used in any architecture, the lifecycle component can be used in any architecture with any components providing `getNextMsg(): Msg` and `step(m: Msg)` and the stigmergy component only needs `myPosition(): Position` to be provided.

At description and development time, the only software dependency between the movement and stigmergy components is the class `Position`, and between message and lifecycle the class `Msg`.

6 Related Works

In this section, we focus on works at the same level than us, *i.e.* the gap between design and implementation, and not on methodologies, specific models of agents or behavioural models.

Modular and implementation-independent design at the agent level has been proposed with the generic agent model (GAM) [Brazier *et al.*, 1999]. The GAM is an high-level, abstract, and component-oriented pattern of agent that defines the essential generic parts of an agent: six interaction and internal components. The GAM can also be reused by specialisation and refinement, but entirely at the charge of the designer. In some sense, the authors propose a very generic agent model (without separation of level) while we advocate for the description of a very specialised one that could be delivered after generation.

Several research works concern implementation of agents by means of software components. MALEVA [Briot *et al.*, 2007] is a model of software components for the building of complex behaviours of agents by composing elementary ones. MALEVA targets the applicative level while we focus mainly on the operational one. Magique [Mathieu *et al.*, 2001] is a platform which permits the construction of agents by gathering reusable units of code representing skills. The set of skills of an agent can change dynamically but the availability of skills at runtime is not guaranteed. In our current solution, it is not possible to change a component that doesn't exist: we preferred to constrain dynamic adaptation in exchange for safety.

Closer to our work, authors of [Garcia and Lucena, 2008] proposes to use aspects-based engineering as a means to define cross-cutting concerns and provide them through components to easily integrates different concerns without modifying the others. Cross-cutting interfaces inverses the dependencies of the component and enables components to be more flexibly composed. Our work focus more on building architectures by integrating different mechanisms that would be used by the developer of the behaviour than building a behaviour as the result of the composition of the mechanisms. More globally, we want to build frameworks by separating its development from its use by non-experts.

For this last concerns, [Weyns and Holvoet, 2006] proposes a complete framework to build situated MAS where holes, called hot-spots, are left to be implemented by the users by relying on provided parts, called frozen spots. In a way this solution is close to ours by simplifying the development of MAS at the programming level, but it only provides a more or less generic model of agent. In fact, this framework could exactly be a dedicated framework produced with our proposition: frozen spots would be operational components and hot-spots applicative components. Using our solution would have added the possibility of reuse of existing components and generation of specialised versions of the framework depending on the application.

7 Conclusion and Perspectives

The global objective of our work is to facilitate the development of MAS. Here, our proposal relies on the use of software components for their advantages in separation of concerns, reuse, composability and safety. In order to fill the gap between design and implementation of multi-agent systems, we have introduced an intermediary phase in the development process based on the realisation of models of agents by component-based multi-level software architectures that are specifically designed for an application. Architectures are coherent (an operation required by a component is provided by another one). Operational components represent sensors, effectors, lifecycle, and other basic mechanisms of the agent while applicative components represent quite the logic of the agent.

The development process is split in two steps which can be carried out by different engineers with different ability

and skills. Dedicated agent frameworks that fit requirements of applications can be generated from agent architectures and delivered to MAS developers, which can benefit from the adequacy and clean abstraction that the framework provides. Connecting components in an architecture is pretty fast and easy, and can be done each time a specific kind of agent is needed. Additionally, the validity of the architectures is preserved at every step of the development process: when generating the framework, when implementing agents, and at runtime when an agent adapts itself by replacing one component by one another.

A framework is a set of classes in an object-oriented language, currently JAVA. In our opinion, the use of JAVA and well-tried technology such as components can foster a smooth integration of the agent-oriented paradigm in the software industry because few new programming concepts need to be learnt by the developers. Coupled with a repository of common components, the development experience of MAS can be improved and the development effort can focus on the problems it has to tackle, that is on "what" agents do rather than on "how" they do it. We also hope this approach can enable reuse and share of works in the MAS community.

To experiment, we have presented tools that make possible to describe and implement agents in an integrated environment based on Eclipse. As said before, experiments are currently done in the context of several projects in different domains.

For the future, we see different research directions to follow. First, to realise model of agents and address agent-oriented concerns, our next step will be to make possible to define and implement environments and mechanisms for interaction as components instead of hiding it behind internal components. Then, we aim at perfecting the component and architecture model we use to allow for better composition and adaptation. Finally, more work is needed to be in line with a complete development process, either by being usable by different methodologies or by exploiting facilities of programming languages. Of course, in parallel to these goals, we are willing to improve the tools as well as to propose a mature library of components for common agent mechanisms.

References

[Bellifemine *et al.*, 1999] F. Bellifemine, A. Poggi, and G. Rimassa. JADE - A FIPA-Compliant Agent Framework. In *Proceedings of International Conference on the Practical Applications of Intelligent Agents*, pages 97–108, 1999.

[Bonjean *et al.*, 2009] N. Bonjean, C. Bernon, and P. Glize. Engineering Development of Agents using the Cooperative Behaviour of their Components. In G. Fortino, M. Cossentino, M.-P. Gleizes, and J. Pavon, editors, *MAS&S @ MALLOW'09, Turin*, volume 494. CEUR Workshop Proceedings, 2009.

[Bordini *et al.*, 2005] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer, 2005.

[Brazier *et al.*, 1999] Frances M. T. Brazier, Catholijn M. Jonker, and Jan Treur. Compositional Design and Reuse of a Generic Agent Model. *Applied Artificial Intelligence Journal*, 14:491–538, 1999.

[Briot *et al.*, 2007] J.-P. Briot, T. Meurisse, and F. Peschanski. Architectural Design of Component-Based Agents: A Behavior-Based Approach. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems (ProMAS 2006)*, volume 4411 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2007.

[Gamma *et al.*, 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Professional Computing Series. A. Wesley, 1995.

[Garcia and Lucena, 2008] A. Garcia and C. Lucena. Taming Heterogeneous Agent Architectures with Aspects. *Communications of the ACM*, 51(5):75–81, 2008.

[Georgé *et al.*, 2009] J.-P. Georgé, J.-P. Mano, M.-P. Gleizes, M. Morel, A. Bonnot, and D. Carreras. Emergent Maritime Multi-Sensor Surveillance Using an Adaptive Multi-Agent System. In *Cognitive systems with Interactive Sensors (COGIS), Paris*. SEE/URISCA, november 2009.

[Leriche and Arcangeli, 2008] S. Leriche and J.-P. Arcangeli. Agent$^{\varphi}$: A Tool for Modeling Composite Self-Adaptive Agents. *International Transactions on Systems Science and Applications*, 4(2):130–138, 2008.

[Mathieu *et al.*, 2001] P. Mathieu, J.-C. Routier, and Y. Secq. Dynamic Skills Learning: A Support to Agent Evolution. In *AISB'O1, Symposium on Adaptive Agents and Multi-agent Systems*, 2001.

[Rougemaille *et al.*, 2009] S. Rougemaille, J.-P. Arcangeli, M.-P. Gleizes, and F. Migeon. ADELFE Design, AMAS-ML in Action. In *Post-Proceedings of the International Workshop on Engineering Societies in the Agents World (ESAW 2008)*, volume 5485 of *LNAI*, pages 97–112. Springer-Verlag, 2009.

[Sellami *et al.*, 2009] Z. Sellami, M.-P. Gleizes, N. Aussenac-Gilles, and Sylvain Rougemaille. Dynamic ontology co-construction based on adaptive multi-agent technology. In *International Conference on Knowledge Engineering and Ontology Development, Madeira, Portugal*. Springer, 2009.

[Szyperski *et al.*, 2002] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. A. Wesley / ACM Press, 2002.

[Weyns and Holvoet, 2006] D. Weyns and T. Holvoet. A Framework for Situated Multiagent Systems. In *SELMAS*, pages 204–231, 2006.

[Weyns *et al.*, 2006] D. Weyns, A. Omicini, and J. Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5, 2006.

[Wilensky, 1999] U. Wilensky. Netlogo, 1999. `http://ccl.northwestern.edu/netlogo/`. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.