Frameworks, Architectures and Components: Revisiting the Development of Multi-Agent Systems

Victor Noël and Jean-Paul Arcangeli

Institut de Recherche en Informatique de Toulouse
Université de Toulouse
118, route de Narbonne, 31 062 Toulouse Cedex, France
{victor.noel,jean-paul.arcangeli}@irit.fr

Abstract. Motivated by the development of Multi-Agent Systems, we investigate in this paper the production of agent frameworks by using component-based software architectures. We introduce the development method SpEArAF (Species to Engineer Architectures for Agent Frameworks) used to answer this problem. The main contribution of this article is essentially in the description of the component model SpeAD (Species-based Architectural Design) that introduces *infrastructures*, *transverse* components and *agent factories*. Agents, implemented with components, are connected to the infrastructure, itself build with components, by the agent factory using transverse components.

Keywords: multi-agent systems engineering, components, component model, architectures, framework

1 Design of Multi-Agent Systems: New Development Paradigms

Modelling a problem and/or its solution with a Multi-Agent System (MAS) is a way to produce software. Such **systems** are composed of entities, the **agents**, that interact together inside an **environment**. The specificity of this kind of modelling lies on the fact that the design effort is focused on the definition of the agents' **individual behaviours**, in order for the system to exhibit a **global behaviour** answering the initial requirements. These behaviours are based on more or less complex interactions between the agents, mostly local, *i.e.* not exploiting global information of the system. This make MASs an approach particularly adapted to distributed, decentralised or acentric systems. MASs-based modelling is mainly used in the artificial intelligence field (problem solving, adaptive systems, etc.), in simulation (in particular social), in robotics (robot collectives) or in ambient intelligence (self-composition of services).[1]

[1] Being not the studied matter here, we will not justify the use of MASs in these contexts.

In this context, a lot of works studied the support of the **design** of these systems, and several **development methods** resulted from it (see [4]). These methods help to identify the agents of the realised MAS, the interaction means, direct or indirect, that are useful to them (message exchange, shared environment, social organisation, etc.) as well as the elements present in their environment. Then, based on existing **theoretical approaches**, these methods give guidelines to define the behaviours of the agents by exploiting these interaction means: all the difficulty in this step is to find the adequate individual behaviours that will enable the global system to have the wanted global behaviour.

From the results of this design phase, the **MAS developer** "just have to" **implement** the system, its agents, its interactions, all the necessary tooling (graphical interfaces, scheduler...) and, if needed, integrate it to existing software systems. Fortunately, most of the methods provide **tools** to ease this work using the artifacts produced during the design process. However, they are mostly limited to supporting the development of the process followed in the method, and do not take into account the **specificity** of the domain to which this process is applied.

Indeed, every method, every theory and actually every MAS has its **own types of agent** with their own interaction means, their own way of using them and all of this in a manner adapted to a specific domain. It is because of this particularity that **agent programming** differentiates itself from other, more classical, programming paradigm (*e.g.* object): every type of agent can be seen as an abstract machine with it inputs, its outputs, its semantics and its dynamic. Then, programming the behaviour of an agent is to program the abstract machine. Every type of agent brings its own **programming paradigm** with its own set of programming primitives. To program with objects, it is translating the manipulated concepts, and in particular interactions between entities, in terms of methods definitions and calls. Differently, MASs show a diversity of interaction means, for example message passing, group communication, indirect interaction by stigmergy[2], explicit organisation with its own dynamic, etc. Thus, to implement a MAS requires additional efforts to implement this new design and programming paradigm, then, at the same time, to use it to develop the built MAS. Some agent **platforms** try to answer this problem by providing more or less generic types of agent, but often by targeting a specific method, theory or application domain, and with a reduced number of interaction means. But this reduces the realm of possibility available to the MAS developer and can even push him to use unadapted abstractions to model its problem and solution.

In contrast, inspired from approaches using software architectures and software components, we dedicated ourselves to the study of means to produce agent platforms "a la carte" that would be adapted to a given domain[3] and to its developers. To achieve that, we propose to build **component-based architectures**

[2] Depositing and smelling "pheromones" in a situated environment. Often used in artificial intelligence.

[3] We will use indistinctly the terms "domain" and "application" when talking about dedicated, adapted or specialised development artifacts.

to realise dedicated agent **frameworks**. The latter would provide the wanted agent platforms accompanied by a way to program the agents executed on them. Defining such architectures is comparable to build the abstract machines able of executing the behaviour programmed in the paradigm corresponding to the realised type of agent. Our **contribution** to the problems is twofold: in terms of **method** (models and process) to help to tackle the design of dedicated agent frameworks, and in terms of **component model** to apply this approach and build the frameworks.

This work is part of more general study on the mutual contributions between software components and software agents, both concepts being commonly seen as two extensions of the objects.

In the following, we propose to present the component model that we defined by showing its specificity inferred from the context of MASs. To motivate this work, we will present in section 2 the method we proposed before, and the problems that brought us to create a new component model. Then we will describe the model in section 3. Finally we will briefly come back on our original objectives in section 4 before doing a quick state of the art and conclude.

2 SpEArAF to Design Agent Frameworks

A Multi-Agent System is constituted of **entities** that can be passive (objects, databases, blackboards and other resources used by the agents) or active (agents but also active objects and other entities that are active without taking part in the system as an agent). Agents interact inside an **environment** (plan, network nodes, networks, organisational structure...) using **action and perception mechanisms** (message passing, read and write on blackboard, access to a camera, to wheels, network mobility...). The environment can possibly be an interaction medium (organisation, blackboard, event manager...). An agent have a **dynamic** (cognitive, reactive...) parametrised by a **behaviour** (knowledge, objectives, parameters...) that characterises him as an individual. The dynamic and the behaviour use and rely on **operational mechanisms** (interaction protocols, computational skills, reasoning...), which possibly exploit the action and perception mechanisms.

As explained before, we proposed a method, named SpEArAF (Species to Engineer Architectures for Agent Frameworks) [8] that provides models and a process. It should be noted that SpEArAF does not compete with other existing MASs design methods that focus on the functionality of the systems. Indeed, SpEArAF has the objective of helping to realise frameworks dedicated to an application by relying on component-based architecture engineering.

Such frameworks provides what we call "**agent species**" and "**ecosystems**". A species is a set of agents with common structural characteristics. An ecosystem is the environment (runtime but also applicative) where the agents of one or several species can exist. We can note that the notion of species and ecosystem differ from the notions of agent and environment used in MASs: indeed, the former covers the functional and domain aspect of the MAS to produce (type of

agents, interactions, organisation...) as much as the operational aspect needed for the practical realisation of the MAS (rules engine, distribution, scheduling, visualisation...). These notions enrich those defined by the MAS development methods and give us a guide to build the frameworks. For example, we can define and implement a species of robots that interact by radio messages and move in a 2 dimensions virtual world, scheduled turn by turn and visualisable using a graphical interface. Or a species of mobile actors (in the spirit of Hewitt and Agha) whose members interact by messages on the distributed nodes of a network. The global idea here is to provide specific types of agents that fits the functional requirements of the system: the application developers can thus rely on the species to design and to implement the MAS **by expressing what the agents "do"** without worrying of operational concerns, *i.e.* **"how" the agents do** what they do. They can thus focus on the functional behaviour of the agents of their applications.

To define an agent species means: a) to identify the mechanisms the agents of the species need to interact with other agents and the ecosystem in which they exist; b) to define the dynamic of the agents of this species (*i.e.* when and how the perceptions are handled and the decisions taken) and the internal mechanisms needed to realise it; c) to define the dynamic of the ecosystem, in particular what allows the agents to interact but also to be created, as well as the internal mechanisms needed to realise it; and d) to define the language (*i.e.* the abstraction, the primitives...) that will allow to program the behaviour of the agents of the species. All of this is equivalent to define, "a la carte", a design and programming paradigm dedicated to the application (to the system) that have to be produced.

SpEArAF has a process in two steps corresponding respectively to the building of a framework by the **framework developer** then to the exploitation of this framework by the **framework user** (the MAS programmer). During the programming of the MAS, the *hotspots* of the framework are instantiated by the framework user to implement the behaviours of the agents. The idea is to build architectures for agents for which some of the components stay abstract (*i.e.* don't have implementation). Like this, the hotspots of the framework based on these architecture will be the components that will be implemented by the framework user. Conversely, the framework developers will provide the implementation for the other components that will be the *frozenspots* of the framework.

From the concept of species, the building of these architectures can be more or less automatised. After presenting our proposal in terms of architecture in the next section, we will give more details on this point in section 4.

3 SpeAD to Describe and Implement Agent Architectures

We presented in the previous section a method supporting the building of frameworks for MASs. In this section, we propose SpeAD (Species-based Architectural Design), a component model enabling the realisation of architectures for these

frameworks. We will focus here on the component model and the architectures that we can describe using it, and not on the produced frameworks.

The concepts defined in SpeAD are separated in 2 levels: infrastructures and components. Our objective is to enable the easy description, implementation, use and reuse of components. In particular, we were careful to have a flexible articulation between the description and the implementation of the component avoiding any repeat and manual verification of coherence between them. For that we exploit code generation and strong static typing of the target language (here Java or Scala).

We first present the components level, its model as well as the way we can implement the described artifacts. Then we will introduce the infrastructure level on top of the component by motivating its necessity.

The interest of using components is not to be demonstrated in the software engineering field, what motivates us here is before everything the reuse of recurring mechanisms in the MASs and the agents to ease the development of MASs and allow for a more broader use of such approaches. Moreover, using fine-grained components and their composition easiness enable us to answer our objective of building domain-specific frameworks. Finally, we exploited the notion of required services to define the abstractions available for the implementation of the hotspots of the produced frameworks.

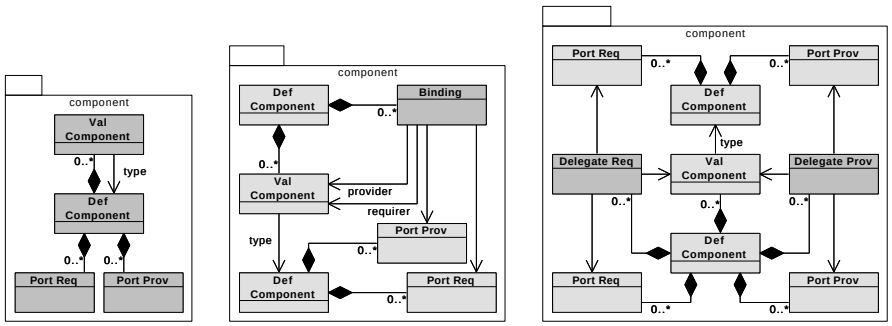
To motivate and illustrate our proposal, we use a common example of MAS: a population of artificial ants. The ant agents moves in a 2 dimensions world and interact by stigmergy: they deposit pheromones at their position and can smell it around them. Pheromones evaporate with time. This metaphor is often used to solve a problem that can be summarised as finding the shortest path in a space: indeed, with the adequate behaviour, the ants are able to find an optimal path from one point to another by exploiting the stigmergy. For us, this example have particularities in terms of MAS: for example, the agents live in an environment and interact with it (moving, depositing and smelling of pheromones), the pheromones have their own evaporation dynamic and the ants have a behaviour defined in terms of perception and action. But it also have operational particularities: for example, the system has to be scheduled in a fair manner for every agent, but also for the pheromones to evaporate coherently. Moreover we would like to visualise the evolution of the system and possibly controls the speed of execution and other such parameters. Finally, in terms of design and programming abstraction, we would like to describe the agents' behaviours in a perception-decision-action cycle by using the high-level mechanisms available to the ant.

3.1 Components

The idea here is to build software architectures that represent the internal part of our agents. For the ant for example, this would be its dynamic and its behaviour: we can have a very simple architecture made of a behavioural component and a lifecycle component. The former contains the behaviour of the ant (in SpEArAF, would be a hotspot of our framework providing the abstraction to program

the agents). It will implement a perception phase (will receive as inputs the perceptions at the current time, *i.e.* the pheromones around), a decision phase and an action phase (will use the interaction mechanisms). The three of these will be available as "services" to the rest of the architecture. The lifecycle will use these services and is responsible of executing the behaviour in the precise order perception-decision-action by giving it the right information at the right time (in SpEArAF, would be a frozenspot of our framework implementing the dynamic of the agents). This architecture will provide a "service" available from the outside to be scheduled as well as require services that will be provided by the infrastructure (presented in the next section).

We will recognise (Fig. 1a) common concepts used in this kind of model: the components have a **definition** with provided and required ports typed by an interface (Java-like). The model is hierarchical and the components are **composite** or **primitive** (*i.e.* with or without sub-components, the `Val Component`). In the composites are defined bindings and delegations to connect the ports of the sub-components together and towards the outside (figures 1b and 1c). Our proposal does not include the definition of new connectors nor parametric typing of components and interfaces but we think that all of this can be introduced in the model without modifying the specificity and advantages of our proposition.



(a) Components. (b) Component: bindings. (c) Component: delegations.

Fig. 1: Definitions of the component model in UML2.

The definition of a component corresponds to a type that can have one or several **implementations**. If the component is composite, its definition also corresponds to an implementation of its type in the form of an architecture connecting its sub-components.

These component definitions are transformed into Java classes that verify the same coherence of the definition by exploiting the strong static typing of the language. Thus, implementing a primitive component is to extend an abstract class and implement the provided ports (*i.e.* the methods of each one of the interfaces). The access to the required ports is done through members of the abstract class that implement the connection of the component to its future ar-

chitecture. The composite component implementations correspond to the several choices of implementations for its sub-components (a choice that can be delayed til the composite instantiation).

3.2 Runtime Infrastructure

The definition and implementation of the runtime infrastructure of the agents is the heart of our proposal. The infrastructure enables the agents to be created, to be executed and to be able to interact together and with the environment. The specificity of our proposition is *a priori* linked to the realisation of MASs, and we couldn't find any equivalent proposition in the literature. Some of the concepts presented here will remind us about the notion of connector: we will attempt to point up the differences between our proposition and connectors.

Motivation. An infrastructure for ant agents must contains an internal architecture[4] that will maintain the internal state of the environment (managing pheromones and the evaporation, movements of the agents...), but also control the scheduling and allow to visualise the system. We want to describe this infrastructure using components for its internal part, but also for the part between the agents and the infrastructure. Indeed, in our example, when we add a new ant to the system we need to connect the internal architecture of the ant (that realises its dynamic and behaviour) to the infrastructure architecture.

At a specific time, from the point of view of the complete executed system containing some agents, there exists a connection between every agent and the infrastructure, and this for each of the interaction mechanisms. This connection is not realised at the system deployment but at runtime since we want agents to be created and destroyed during the system life. Here for example, every ant agent architecture must be connected to be scheduled, to move and to deposit/smell pheromones. Moreover these mechanisms must sometimes be deployed both on the infrastructure side and the agent side: for example for stigmergy, evaporation is done in the infrastructure, deposit and perception are executed in the infrastructure from the point of the view of a particular agent that is in a particular position, and the pheromones stock is proper to each agent. We will note in particular that the agent and the infrastructure are two elements of the system of different nature: the problems here are not to connect two agents together like we would connect two components with a connector, but to connect an agent to the infrastructure that will enable him to be executed (to "live"), and possibly to interact with other agents.

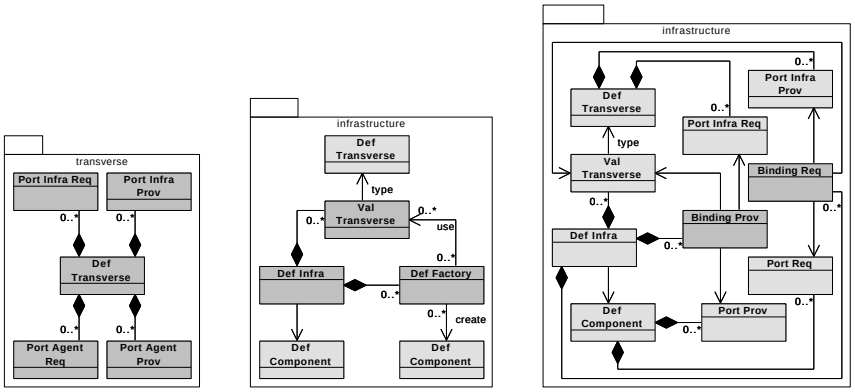
We want to be able to describe and implement the mechanisms that are between the agent and the infrastructure, and at the same time describe and implement the way to connect these mechanisms to the agent. Moreover we have to handle this agent-infrastructure link, but also the links between the several

[4] We will use indistinctly the terms "architecture" and "composite component" in the rest.

interaction mechanisms of one agent: for example, the position of an agent is needed to implement the deposit of pheromones of this particular agent.

Thus, we propose to provide a model to define such infrastructures, but also to implement them so that the created mechanisms and components are reusable and so that it is easy for the developer to build new infrastructures. We identified two orthogonal challenges to allow the realisation of such infrastructures: the definition and implementation of the agent-infrastructure interaction means, and their composition with the internal architecture of the agents when he is instantiated.

To answer the first problem, we propose the *transverse* components. To answer the second problem, we propose the infrastructures and their *agent factories*.



(a) Transverses. (b) Infrastructure. (c) Infrastructure: bindings.

Fig. 2: Definitions of the infrastructure and transverse model in UML2.

Proposed Solution. The transverse component (Fig. 2a) allows to define a type of link between an agent and an infrastructure in a reusable manner. Its objective is to define the mechanisms enabling the interaction between the agents and their environment, including other agents. For this, the model first allows to describe what a transverse provides and requires from the infrastructure, but also what it provides and requires from each of the agents that will be connected to the infrastructure using it.

Then, after the definitions are transformed into Java classes, implementing a transverse is to define:

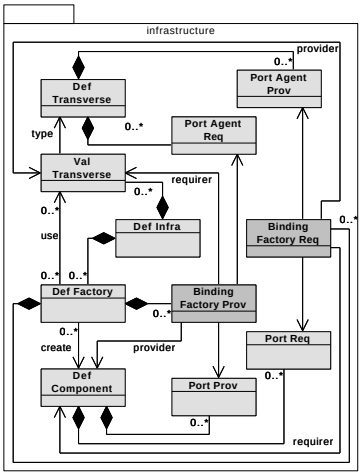
– The part of the link on the infrastructure side, for which there will be one instance for each transverse instance in each infrastructure. It implements the infrastructure side provided ports and use the infrastructure-side required ports.

– The part of the link on the agent side, for which there will be one instance per agent for each transverse in each infrastructure. It implements the agent-side provided ports and use the agent-side required ports.
– The factory that will be used to instantiate the link on the agent side every time an agent is created.

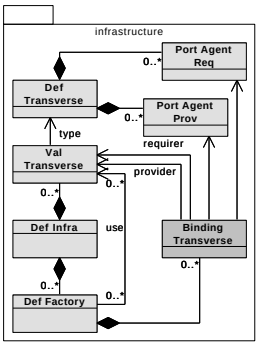
The connection between the two parts of the transverse component are encapsulated inside the implementation of the component.

Infrastructures (Fig. 2b) are aggregates of transverses components possibly connected to a composite component representing the internal architecture of the infrastructure. To describe an infrastructure means to express to define the sub-component for the internal architectures and the set of transverse components. Then, it is to describe the bindings between the infrastructure part of the transverses and the internal composite (Fig. 2c).

In an infrastructure (*i.e.* given a set of transverse components), we can define as much factories as we have ways of connecting an internal agent architecture to the infrastructure. Describing a factory is to express first what are the transverse that will be used by the agent (Fig. 2b). Then it is to express the bindings between provided and required ports of the composite realising the internal architecture of the agent, and the required and provided ports of the agent side of the transverse that the agent use (Fig. 3a). But it is also to describe the binding between the provided and required ports of the agent side of the transverses between them (Fig. 3b). To enable to create agents (possibly by other agents), the factories are available through ports provided by the infrastructure.



(a) Factory: bindings between agents and transverses.



(b) Factory: bindings between transverses.

Fig. 3: Definitions of the factory in the infrastructure model in UML2.

These definitions are also transformed into Java classes. Thus, implementing an infrastructure is to choose the implementations of its composite and its transverses. To implement a factory is to define how is instantiated the architecture of the agent as well as the agent side of the transverses. In a way, a factory is the composition of the factories of the transverses with the architecture of an agent.

3.3 SpeADL to Describe Architectures

To allow the description of these architectures practically, we defined an architecture description language named SpeADL (Species-based Architectural Design Language). An example of its use for the ants MAS is given Fig. 4. This description is not exactly identical to the example developed in this article: in particular we added a transverse component to represent the notion of situated agent.

```
component Lifecycle {
 required perceive: Perceive
 required decide: Decide
 required act: Act
}
component Behaviour {
 provided perceive: Perceive
 provided decide: Decide
 provided act: Act
 required move: Move
 required deposit: Deposit
 required sniff: Sniff
}
component CAnt {
 required move = c.move
 required deposit = c.deposit
 required sniff = c.sniff
 val c: Behaviour
 val s: Lifecycle {
  bind perceive to c.perceive
  bind decide to c.decide
  bind act to c.act
 }
}
component 2DMatrix {
 provided getset: GetSet
}
transverse Situated {
 agent provided position: Position
 agent provided move: SetPosition
 infra required getset: GetSet
}
transverse Movement {
 agent provided move: Move
 agent required position: Position
 agent required setPos: SetPosition
 infra required getset: GetSet
}
transverse Pheromones {
 agent provided deposit: Deposit
 agent provided sniff: Sniff
 agent required position: Position
 infra required getset: GetSet
}
infrastructure IAnt {
 val map: 2DMatrix
 val d: Movement {
  bind getset to map.getset
 }
 val s: Situated {
  bind getset to map.getset
 }
 val p: Pheromone {
  bind getset to map.getset
 }
 factory FAnt creates CAnt {
  bind move to d.move
  bind deposit to p.deposit
  bind sniff to p.sniff
  use d {
   bind position to s.position
   bind setPos to s.move
  }
  use s
  use p {
   bind position to s.position
  }
 }
}
```

Fig. 4: Example of the use of SpeADL.

4 Exploitation

Back to SpEArAF. To answer our initial problems, here is sketch of the way to connect the concepts used in SpEArAF to the concepts of SpeAD:

- the identification of the abstractions needed by the framework user as well as the definition of the dynamic of the agents of the species will guide the design of the species architecture, of the factories and of the choice of the abstract and implemented components (hotspots and frozenspots of the framework);
- the identification of what compose the ecosystem, be it MAS (functional), runtime or deployment (operational), or user environment, will guide the design of the architecture of the infrastructure; and
- the identification of what enable the architecture of the species to interact with the infrastructure architecture will guide the design of the transverse components.

Implementation. A first version of the SpeADL language was implemented in the shape of a textual editor and a code generator, but without the infrastructure

aspect, which limit its interest.[5] Prototypes were made to verify the feasibility of the complete approach, and a usable version is currently being realised.

Application. This approach is applied in several research projects, in particular in the field of distributed robotics to design the architectures of robots meant to be used in a simulator and in real robots, see [5] for more information. Other projects include social simulation and ambient intelligence.

5 State of the Art

We very briefly study here some works that can be compared to our and show in what they differ.

Multi-Agent Systems. Several works exist in the MAS field exploiting components to build agents. We can take as specimens Generic Agent Model (GAM) [1], MALEVA [2] and Magique [7]. These works use components to build agent architectures. GAM is a generic agent model made of components and specific interaction mechanisms. MALEVA is focused on applicative aspect of the agent and allows to design the behaviour of the agent using components. Magique has the same objective but adding runtime dynamism to it. But all of these works are focused on the behavioural aspect of the agent, they proposes a fixed specific type of agent (a species of agents) and do not tackle the infrastructure needed to support the system.

Composants. As said before, we can see some similarities between transverses components and connectors. For example, Medium [3] proposes a way to implement connectors in a distributed environment and easing the reuse. We can find there the definition of a type of component acting as an interaction mean, corresponding in our approach by the transverse component supported by the infrastructure. However, we are focusing on the integration of a component, the agent, in an infrastructure, which differentiate conceptually Medium from our work. Even if less mature on the transverse component, our approach, motivated by MASs, proposes to define a mean to connect everything at runtime with the factories. This enable us to describe the composition of several transverse component with one agent architecture.

[6] proposes a mean to define domain-specific concepts that will guide the generation and implementation of a dedicated component-based framework. This work differentiates from ours by the fact that it defines a component model that is used as a basis for the generation of a generic framework, while we define directly a framework that we implement using an architecture defined using a component model.

[5] See `http://www.irit.fr/MAY`.

6 Conclusion

In this article, motivated by the development of Multi-Agent Systems (MASs), we presented a method to support the building of domain-specific agent frameworks. To realise them, we propose a component model that differs from existing work by the two available description levels: component and infrastructure. In particular, a specific type of component is described, the transverse component, that allows to connect components to an infrastructure using an agent factory.

This work allowed us to corroborate our interest in the presented abstractions. The next step will focus on the problems linked to the implementation of the transverse components and the factories. Moreover, this work is done in the context of a broader research on the link between components and agents. Among other things, we are interested in the question of knowing if the definition of a species of agent can be considered as the definition of a component model, of an architectural style or even of a component container. Another relevant axis is the use of our approach to support software product lines for MASs as well as the study of its integration with the existing MAS development methods.

References

1. Brazier, F.M.T., Jonker, C.M., Treur, J.: Compositional Design and Reuse of a Generic Agent Model. Applied Artificial Intelligence Journal 14, 491–538 (1999)
2. Briot, J.P., Meurisse, T., Peschanski, F.: Architectural Design of Component-Based Agents: A Behavior-Based Approach. In: Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.) ProMAS 2006. LNCS (LNAI), vol. 4411, pp. 71–90. Springer (2007)
3. Cariou, E., Beugnard, A., Jézéquel, J.M.: An architecture and a process for implementing distributed collaborations. In: EDOC. pp. 132–143 (2002)
4. Henderson-Sellers, B., Giorgini, P.: Agent-Oriented Methodologies. IGI Global (2005)
5. Lacouture, J., Noel, V., Arcangeli, J.P., Gleizes, M.P.: Engineering Agent Frameworks: An Application in Multi-Robot Systems. In: Demazeau, Y., Pechoucek, M., Corchado, J.M., Bajo, J. (eds.) International Conference on Practical Applications of Agents and Multiagent Systems, Salamanca (Spain), 06/04/2011-08/04/2011. Advances in Intelligent and Soft Computing, Springer (2011)
6. Loiret, F., Plsek, A., Merle, P., Seinturier, L., Malohlava, M.: Constructing domain-specific component frameworks through architecture refinement. In: EUROMICRO-SEAA. pp. 375–382 (2009)
7. Mathieu, P., Routier, J.C., Secq, Y.: Dynamic Skills Learning: A Support to Agent Evolution. In: Adaptive and Emergent Behaviour and Complex Systems Convention, Symposium on Adaptive Agents and Multi-agent Systems (2001)
8. Noël, V., Arcangeli, J.P., Gleizes, M.P.: Between Design and Implementation of Multi-Agent Systems: A Component-Based Two-Step Process. In: Moraitis, P., Miles, S. (eds.) European Workshop on Multi-Agent Systems (EUMAS), Paris (France), 16/12/2010-17/12/2010 (2010)