Between Design and Implementation of Multi-Agent Systems: A Component-Based Two-Step Process

Victor Noël, Jean-Paul Arcangeli, Marie-Pierre Gleizes

Between Design and Implementation of Multi-Agent Systems: A Component-Based Two-Step Process

Victor Noël, Jean-Paul Arcangeli, and Marie-Pierre Gleizes

Institut de Recherche en Informatique de Toulouse
Université 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 ease the development of Multi-Agent Systems (mas), we propose a two-step process named SpEArAF (Species to Engineer Architectures for Agent Frameworks) that aims to facilitate the transition between design and implementation and to promote reuse. The idea is to build, in the first step, a specialised and reusable framework (a programming library) that fits the application requirements, particularly concerning the agents and their interaction abilities. In order to do that, application-specific "species" of agents are defined, then realised as architectures in a component-based manner, and finaly implemented by either programming new components or reusing existing code. This step defines "how" agents work and is done in order to provide adequate programming abstractions depending on the ability and skills of the end-user developer. In the second step, end-users implement the mas by defining application-level behaviours of agents via the dedicated programming mechanisms at the level of abstraction provided by the framework. This step defines "what" agents do to realise the functionality of the mas. In practice, SpEArAF relies on Make Agents Yourself (may), a tool integrated into Eclipse, supporting the description of species of agents and their transformation into executable agents implemented in Java.

Keywords: MAS engineering, process, detailed design, implementation, framework, reuse, separation of concerns, components

1 Introduction

MAS are complex systems that are composed of interacting and possibly heterogeneous agents. 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 application and with the developers' programming abilities, the more developers can concentrate on "what" their system does instead of "how" it works: in which case development is facilitated and the produced software is easier to maintain. In complement, reuse of pieces of code and facilities for code composition are often suitable.

Motivation. When implementing MAS, developers must fill the gap between the agent platform they chose and the agents of their application: many platforms (often realised as programming frameworks) have been proposed (Jade, Jadex, AgentScape, Jason, MadKit, CArtAgO, NetLogo, Repast, PDT...), some of them specifically for agent-oriented methodologies (Tropos, Prometheus, Ingenias, Passi, SODA, IODA...), or theoretical models (MASQ, A&A...). The main problem is how the agent concepts ("agent", "interaction", "communication", "adaptation"...), necessary for the application and used to describe the agents, are defined and provided by the chosen platform and its programming model. Most of the time, developing a MAS results in the chosen platform being tightly coupled to the used design methodology, or worse, in agents designed to target an existing agent platform. But in reality, depending on the application, different "species" of agents are needed, whether by the way they perceive, act, communicate or by the way they are "anatomically" organised to behave, decide and react to external stimuli: an agent controlling a mobile robot tracking targets by camera does not need the same capabilities and interaction mechanisms with its environment than a grid mobile agent solving constraint-based problems in a distributed computing context.

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 [1], stigmergy must be implemented through message passing, which can be considered as too low-level in this case. If the chosen platform is NetLogo [17], implementation of stigmergy is 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 "species of agent" proposed by the platform and those of the application only burdens the developer with different concerns mixed together. The same problems appear in the "anatomy" of the agent: (i) a reactive agent like an ant will need different "organs" than a proactive agent with a BDI architecture, but also (ii) managing the execution of the tasks of a goal-based agent behaviour is different than expressing the goals, beliefs and reactions to stimuli of the agent.

Existing design methodologies provide guidance to develop MAS by identifying agents and environment, as well as designing their behaviours and interactions: their objective is mainly on the functionality of the MAS through the definition of agents. They often focus on some specific species of agents and/or agent platforms (e.g. Passi, Ingenias for FIPA agents, Tropos for Jadex, etc.). They can also provide tools and models to implement the MAS, but focus on the implementation of the behavioural part of the agents (goal-based, BDI, subsumption, etc.) that have to be (or is) included in an existing agent platform. When the resulting implementation is put in a specific software environment (the final application), more work has to be done by the developer to integrate it with the domain artifacts such as GUIs (per agent or per MAS), sensors, databases, schedulers but also domain-specific MAS environment.

We are missing a way to build, with reusability in mind, these platforms for the MAS and the agents in order to be able to implement such application-specific species of agents used in methodologies. Moreover, to be able to separate the code which uses the high-level (domain-specific) mechanisms from the code which realises them seems to be a desired advantage in terms of development. Thus, the following proposition focuses on this orthogonal concerns of building a platform for specific species of agents that can be completed with behaviours coming from design methodologies.

Proposition. Bridging the gap between analysis and implementation is a key challenge for the MAS community [3]. To complete methodologies that focus on designing the functionality of a MAS and instead of proposing another more or less generic agent platform, we propose SpEArAF, a **development process** that promotes the engineering of application-specific frameworks to realise agent platforms for the development of multi-agent applications. Such frameworks provide what we call "species of agents": **species define sets of agents with common structural characteristics**. By defining species, the idea is to provide specific types of agents that fit functional requirements: developers can rely on species both when designing and implementing the MAS (the expression of "what" agents do), they don't need to deal with operational concerns ("how" agents do) and can focus on the agent's functional behaviours.

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 [14]. It promotes separation of concerns and definition of clear, composable, and reusable abstractions. This work is part of a larger study on the possible links between component and agent technologies. Here, we aim at assisting framework developers in the use of component-based technologies for MAS development while taking into account general and agent-oriented software engineering concerns such as flexibility, autonomy or adaptation. We thus propose a development process that enables the realisation of species of agents by designing software architectures ("anatomy") composed of software components ("organs") by identifying a species of agent corresponding to the requirements for the agents of the application and assembling components (implementation or reuse) in architectures for agents to create a framework. In this way, the level of abstraction and expressiveness of the produced framework can be adjusted depending on the expertise of the targeted user of the framework (*i.e.* the developer of the application).

This work has been used for biological simulation [2] (cells that divide, mutate, die and communicate by exchange of molecules), dynamic ontology construction [13] (self-organising term agents creating ontological relations) and naval surveillance [8] (agents representing real boats computing a threat level). 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]). Early results of integrating this approach with the ADELFE design methodology already exist [12].

[1] `http://www.irit.fr/Rosace,737`
[2] `http://www.iaai-maelia.eu/`

The interested reader can find in the present proceedings an illustration of SpEArAF for the development of a framework for adaptation of workflow services composition using an agent-oriented organisation model [9].

Outline of the paper. In Sect. 2, we present the two-step development process SpEArAF that aims to describe component-based agent architectures as a mean to realise species of agents dedicated to an application. Then Sect. 3 presents the μADL description language, the translation from μADL to a class-based object-oriented programming language to make the implementation of the architectures possible, and MAY, a set of model-based tools with adequate editors and generators for Java. In Sect. 4, one simple and one real world examples are presented. Finally, related work is discussed in Sect. 5, then conclusions and some interesting perspectives are presented in Sect. 6.

2 SpEArAF: Architectures and Frameworks for Species of Agents

We now present SpEArAF (Species to Engineer Architectures for Agent Frameworks) the development process that, as said before, has not for objective of helping the design of a whole MAS, but precisely to help the realisation of a framework used to implement the MAS designed.

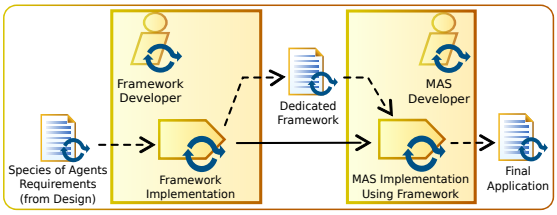

Fig. 1. A Simple View of SpEArAF

Figure 1 shows the proposed development process where we distinguish the following steps: 1) creation of a framework by the **framework developer** and 2) use of this framework to develop the MAS by the **framework user**. Indeed, when programming the MAS, *hotspots* in a produced frameworks can be instantiated (possibly with sub-architectures) by the framework user to define the behaviour of the agents. For that, he uses a set of agent-oriented and application-specific programming primitives defined by the framework developer.

In order to do that, the main concern is to build a framework answering the requirements of the stakeholders to describe the agents of the application. Here stakeholders are the framework users, *i.e.* the MAS designers (define the species of agents and design the system as a whole) and final developers (implement the agent behaviours), as well as the different persons involved with the application itself as in classical architectural approaches. Such requirements (the species of agents) will be expressed as:

1. the description of the MAS infrastructure, environment and interaction means;
2. the description of the dynamics of the agent, such as the way external stimuli are handled internally and how to interpret, process, decide and act;

3. the set of programming primitives — or dedicated language — that will be used to express the behaviour, with respect to the dynamics of the agent.

Thus, such requirements could possibly come from the analysis and design phase of an agent-oriented software engineering methodology where the environment, the agents and their behaviours would have been identified.

2.1 Building a Dedicated Agent Framework

As said in the introduction and in line with previous works on the Agent$^{\varphi}$ approach [10], in order to enable easy reuse and composition of the developed artifact, SpEArAF relies on component-oriented architectures and code generation. Here, software components are a way to provide different mechanisms, the organs of our agents, to framework developers in a reusable and composable way. Practical realisation of these components and architectures is detailed Sect. 3.

Based on the targeted species of agents, the production of a framework can be divided in the following steps:

1. identifying action-perception means (communication included) used by the agent to interact with its environment and other agents
   → a set of *action-perception components* that immerse agents into the MAS
2. identifying the programming primitives that the framework user will use to program its agents
   → a set of *behaviour components* with high-level and dedicated required and provided interfaces that will hide the implementation of the services
3. building the anatomy realising the species and enabling the execution of the previously identified *behaviour components*
   → a set of *organ components* implementing the dynamics of the agent
   → an architecture with *organ*, *action-perception* and *behaviour components*
4. building an *infrastructure* for executing the agents and connecting them to their (runtime and MAS) environment

Typically, *organ components* implement lifecycles, adaptation, GUI, capabilities, knowledge management etc. *Action-perception components* implement sensors, actuators, messages passing and other interaction means. *Infrastructure* implement environment such as 2D plan, extra-agent organisation dynamics, but also scheduling, distribution, visualisation GUI, etc. Finally, *behaviour components* are left to be implemented to express behaviours, interpretation of perception...

2.2 Exploiting a Dedicated Agent Framework

At the end of the previous phase, we end up with new reusable components and an application-specific framework composed of reused and produced components. The framework user is responsible of using the produced framework and components to implement the behaviours of the agents based of the design of the MAS. The idea being that this phase can only focus on defining behaviours without being bothered with low-level technical mechanisms.

Moreover, an important point of this approach is developing **for** reuse and not only by reuse. In particular, an objective is to be able to produce frameworks (*i.e.* species or architectures) that can be reused completely or partly to produce different applications relying on the same species of agents. For example reusable components could be produced and reused by the designers and developers of a specific methodology.

3 Applying SpEArAF: Languages and Tools

In this section, we present the μADL description language to describe components and architectures, the translation from μADL to Java and tools such as an editor for this language and a generator implementing the translation. Integrating all of this results in the complete development process for detailed design and implementation of MAS shown Fig. 2.

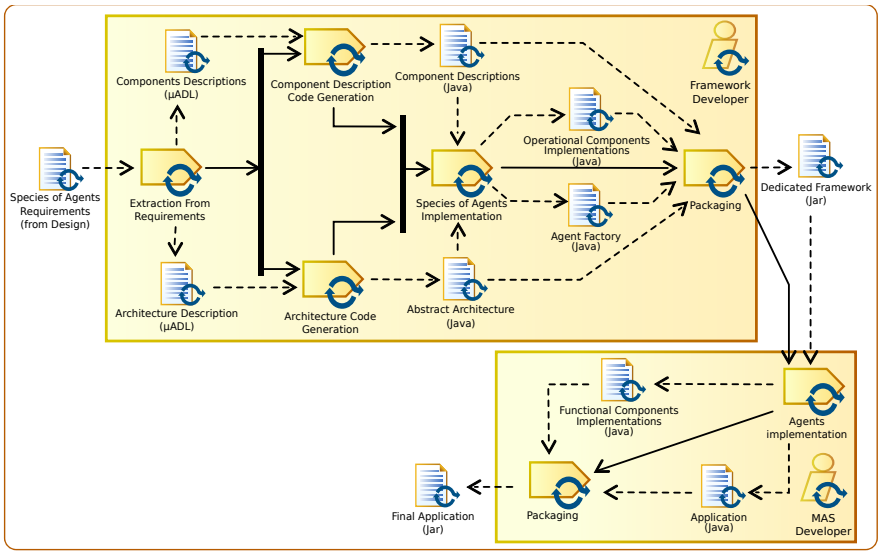

Fig. 2. A Complete View of SpEArAF

3.1 μADL: Components and Architectures

In μADL, agent architectures are described using two constructs using:

- components, at description and implementation, for separation of concerns;
- two levels, operational and applicative, respectively what will be part of the dedicated agent framework (mainly *organ* and *action-perception components*) and what will need to be implemented to program the agents (*behaviour components*).

In this section, the μADL language is presented, then we show how it answers software engineering requirement such as re-usability, evolution, safety but also agent concerns such as autonomy or self-adaptation.

Components. 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 (in terms of implementation) 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. 3.2). Finally, it can specify a persistent (typed) state that must be kept when its implementation is dynamically changed.

In all these descriptions, 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).

Some descriptions for the example components from Sect. 4 are shown Figs. 3(b), 3(c) and 3(d).

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

 operational {
  Message
  LifeCycle
  Executor
 }

 application {
  changeable Behaviour
 }

 visibility {
  external receive(m: Msg)
  application send(a: Agent, m: Msg)
  application me(): Agent
  application suicide()
 }
}
```

(a) Architecture

```
component Behaviour {
 package my.comps.simple

 provided step(m: Msg)
 required send(a: Agent, m: Msg)
}
```

(b) Behaviour Component

```
component LifeCycle {
 package my.comps.simple

 required step(m: Msg)
 required getNextMsg(): Msg
 required execute(r: Runnable)
}
```

(c) LifeCycle Component

```
component Stigmergy {
 package my.comps.ants

 provided deposit(quantity: Int)
 provided scent(): Int
 required myPosition(): Position
}
```

(d) Stigmergy Component

Fig. 3. Descriptions in μADL

Architectures. An agent architecture realises a species of agents by connecting components. It is separated in two levels: operational and applicative.

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 autowired by the architecture (conflicts are not currently handled, aliasing of operations is a possible solution). In the architecture, it is specified if components are replaceable at runtime (with the construct `changeable`), if some operations are available at the `application` level, and if some operations are available from outside of the agent (with the construct `external`).

Description for the architecture of the species of agents of the example in Sect. 4 is shown Fig. 3(a).

Infrastructure. The infrastructure where the agents are executed results from the *action-perception components* of the agents and the use of the `external`

operations of their architectures. This special construct is used to access to an agent from the infrastructure and can be considered as an interface provided by the agent. Thus, the infrastructure itself (*infrastructure* and *environment components*) as well as the glue between the agents and the infrastructure has to be implemented in an ad-hoc way, *i.e.* hidden in the implementation of the action-perception components.

So, 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 [16].

3.2 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 consistent. So, when implementing components, the required operations are always provided. More generally, to use architectures allows to reason on the built system, which is not our concern here.

Flexibility. 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)...

Self-Adaptation. Furthermore, providing self-adaptation mechanisms to the agents can be accomplished by using 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 consist in the self-replacement of the reasoning process depending on the context, or the upgrade of a component with bugs.

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

Autonomy. Two aspects of the autonomy of agents are addressed by this 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 species 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).

Usability. Programming the agent relies on dedicated agent framework, which is a set of classes corresponding to the implementation of the architecture and the components of the operational level forming together a specialised architecture (*frozenspots*). Using a framework consists in implementing the remaining

components (*hotspots*) at the convenient level of abstraction (application level) and plug them in the provided specialised architecture. Finally, at runtime, an agent is an executable instance of a μADL architecture where all components have been implemented.

3.3 Generating Dedicated Agent Frameworks

We detail here how architectures and components are translated to JAVA to answer the previously presented soft-engineering concerns.

Translation. Our approach relies on the following automatisable translation from μADL to a class-based object-oriented programming language. This exploits the type system of the language and well-known design patterns [6].

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 are added to the component description.

Each agent architecture is translated to a class that contains an attribute for each of its components: it connects them and is responsible for 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, a framework provides a factory for the specialised architecture (Factory Pattern) with some fixed component implementations and holes (*hotspots*) for the others.

Consistency 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, consistency of the architecture is preserved (see also Sect. 3.2). The agent architecture is set at compile time and the only possible runtime modification are architecturally safe.

3.4 MAKE AGENTS YOURSELF

To validate these propositions and experimentally apply them, we developed and released MAKE AGENTS YOURSELF (MAY)[3]. It provides textual and graphical editors for the μADL language to define components and architectures, a tool to

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

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 tool users 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 metamodel 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 Modeling ecosystem. The textual editor was realised using TMF Xtext, while the graphical is based on GMF. Constraints to check the consistency of the descriptions are developed with TMF Xcheck and code generation with TMF Xpand.

4 Examples

We show here two examples. The first one is simple and has for objective to show how the μADL language is used andhow components are developed to produce a dedicated agent framework. The second one is taken from a real research project and is focused on illustrating the process itself. The website of MAY provides other complete examples.

4.1 Simple Stigmergic Agent

We show now a simple example of communicating agent then complicate it by adding movement and stigmergy in a virtual space. We build common agent components from scratch to show how our proposition handles MAS concerns. We want to build agents complying to the following species 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 as input and that can use the send primitive.

First we write a component description for the behaviour (Fig. 3(b)): it provides `step(m: Msg)` and requires `send(a: Agent, m: Msg)`; and a component for lifecycle (Fig. 3(c)): it requires `getNextMsg(): Msg`, `step(m: Msg)` and `execute(r: Runnable)`. Then, we write an architecture description stating that it must have a component behaviour in its application level as well as a component lifecycle in its operational level. Using the editors provided by MAY, an error informs us that the dependencies `getNextMsg(): Msg`, `send(a: Agent, m: Msg)` and `execute(r: Runnable)` are not present in the architecture. We thus write a description for a messages component providing both first and an executor component providing the latter. We add it in the operational level of the architecture: the architecture is now consistent. Lifecycle and executor are *organ components* and messages is *action-perception component*.

We generate the corresponding Java classes (component description and agent architecture) using MAY. First we implement the lifecycle, see Fig. 4, implementing the definition of the dynamics of the species of agent: in a loop executed by the executor, take a message in the mailbox (which blocks if there

is no message) and treat it with the behaviour. The executor component implements the `execute(r: Runnable)` by running a task in a thread. 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 components. We can export this set of classes to make a deliverable framework for this species of agent. To program this species of agent, one needs to create a class implementing the behaviour component using the required operations, and create an agent using the factory.

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

Fig. 4. Implementation for LifeCycle

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 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. Stigmergy and movement are *action-perception component* and encapsulates the infrastructure that will be made of the 2D plan and pheromones dynamics.

Implication for Reuse. The executor, 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`. Of course, the produced framework itself is usable in different applications that use this species of agents.

4.2 Real World Example

We show now an example of the application of SPEARAF to answer the requirements expressed in the Rosace project. The objective of one of the subgroup of

the project is to evaluate different multi-agent strategies for dynamic task allocation between a group of autonomous robot. The chosen solution relies on the use of the Morse OpenRobots simulator[4] where robots are controlled by agents. After dialogue with the partners, the expressed requirements were the following:

1. build a framework to allow agents to control simulated robots;
2. build different anatomies of agents depending on the strategies;
3. deliver theses to the different research groups for them to experiment their strategies with different parameters and behaviours.

We will cover here the first requirement. From this point of view, the dedicated language (with its dynamics) to program the agent contains the following:

- primitives to consult the reachable robots by radio, send them messages as well as consult the received messages;
- primitives for consulting the objects visible by the robot with its camera, its GPS position, its orientation;
- primitives to move the robot following a path as well as events to be notified of the arrival at the different point of the path (using callback);
- primitives to execute concurrent tasks;
- a queue to store the received messages by radio.

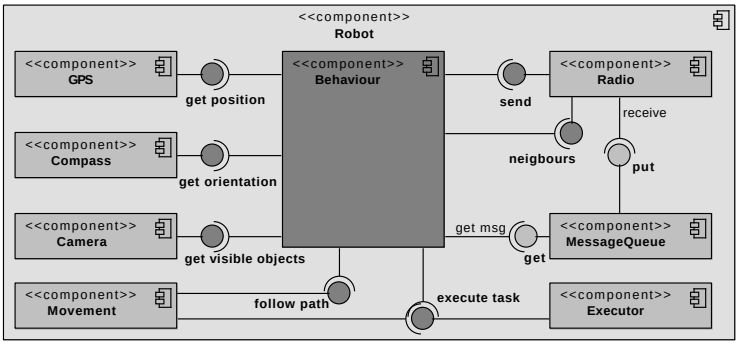

Fig. 5. Architecture of the Species of Robot

This species of agents requirements corresponds to the definition of a set of interfaces as well as the definition of the anatomy of agent depicted Fig. 5. The *action-perception components* are `GPS`, `Compass`, `Camera`, `Movement` and `Radio`. They implements the bindings between the agents and the simulator. The internal anatomy (*organ components*) is made of the `MessageQueue` and `Executor`. Finally `Behaviour` is a *behaviour component* that will contains the behaviour of the robot exploiting the presented primitives.

This architecture can then be used by the different research groups to implement their specific species with their anatomy by replacing `Behaviour`.

5 Related Work

In this section, we focus on works at the same level than ours, *i.e.* the gap between design and implementation, not on methodologies, specific species of

[4] `http://morse.openrobots.org`

agents or behavioural models: indeed, as said before, design methodologies and their associated tools help the developer to design and implement the behaviours of its agents in one continuous way by relying on an (eventually) existing agent framework. Inversely, our approach focuses on building an agent framework itself by explicitly separating this step from the behaviour definition in order to produce a framework adapted to the problem and the developer. In this sense, our approach is orthogonal to the focus of current agent-oriented methodologies.

Modular and implementation-independent design at the agent level has been proposed with the Generic Agent Model (GAM) [4]. 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 is delivered after generation.

Several research works concern implementation of agents by means of software components. MALEVA [5] 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 [11] 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: dynamic adaptation is constrained in exchange for safety.

Closer to our work, authors of [7] propose to use Aspect-Oriented Software Development 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 framework user 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, [15] proposes a complete framework to build situated MAS where holes, called hot-spots, are left to be implemented by the framework 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 species 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 this framework depending on the application.

6 Conclusion and Perspectives

The global objective of our work is to facilitate the development of MAS. Our proposal relies on the use of software components for their advantages in separa-

tion of concerns, reuse, composability and safety. In order to fill the gap between design and implementation of MAS, we have introduced a development process named SpEArAF based on the realisation of species of agents by component-based software architectures that are specifically designed for an application. Architectures are consistent (an operation required by a component is provided by another one). Organ and action-perception components realise sensors, effectors, lifecycle, and other basic mechanisms of the agent while behaviour components realise 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. The μADL language as well as adapted tools are provided to help them to apply the process. Dedicated agent frameworks that fit requirements of applications can be generated from agent architectures and delivered to MAS final 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 species of agent is needed. Additionally, the consistency of the architectures is preserved at every step of the development process: when generating a 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 thanks to the small amount of new programming concepts needed to be learnt by the final 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 apply the process, we presented tools allowing for the description and implementation of agents in an integrated environment based on Eclipse. 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 species 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, in particular current developments focus on providing a more advanced description language with hierarchical components and interface bindings. Finally, more work is needed to be in line with a complete development process, either by integrating our proposition with 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

1. Bellifemine, F., Poggi, A., Rimassa, G.: JADE - A FIPA-Compliant Agent Framework. In: Proceedings of International Conference on the Practical Applications of Intelligent Agents. pp. 97–108 (1999)
2. Bonjean, N., Bernon, C., Glize, P.: Engineering Development of Agents using the Cooperative Behaviour of their Components. In: Fortino, G., Cossentino, M., Gleizes, M.P., Pavon, J. (eds.) MAS&S @ MALLOW'09, Turin. vol. 494. CEUR Workshop Proceedings (2009)
3. Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.): Multi-Agent Programming: Languages, Platforms and Applications, Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15. Springer (2005)
4. 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)
5. 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)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Professional Computing Series, A. Wesley (1995)
7. Garcia, A., Lucena, C.: Taming Heterogeneous Agent Architectures with Aspects. Communications of the ACM 51(5), 75–81 (2008)
8. Georgé, J.P., Mano, J.P., Gleizes, M.P., Morel, M., Bonnot, A., Carreras, D.: Emergent Maritime Multi-Sensor Surveillance Using an Adaptive Multi-Agent System. In: Cognitive systems with Interactive Sensors (COGIS), Paris. SEE/URISCA (November 2009)
9. Henrique Cruz Torres, M., Noël, V., Holvoet, T., Arcangeli, J.P.: MAS Organisation at your Composite Service. In: EUMAS'10 (December 2010)
10. Leriche, S., Arcangeli, J.P.: Flexible Architectures of Adaptive Agents: the Agent-ϕ approach. Tech. Rep. RR-2008-11-FR, IRIT (April 2008)
11. Mathieu, P., Routier, J.C., Secq, Y.: Dynamic Skills Learning: A Support to Agent Evolution. In: AISB'O1, Symposium on Adaptive Agents and Multi-agent Systems (2001)
12. Rougemaille, S., Arcangeli, J.P., Gleizes, M.P., Migeon, F.: ADELFE Design, AMAS-ML in Action: A Case Study. In: Post-Proceedings of the International Workshop on Engineering Societies in the Agents World (ESAW 2008). LNAI, vol. 5485, pp. 97–112. Springer-Verlag (2009)
13. Sellami, Z., Gleizes, M.P., Aussenac-Gilles, N., Rougemaille, S.: Dynamic ontology co-construction based on adaptive multi-agent technology. In: International Conference on Knowledge Engineering and Ontology Development, Madeira, Portugal. Springer (2009)
14. Szyperski, C., Gruntz, D., Murer, S.: Component Software - Beyond Object-Oriented Programming. A. Wesley / ACM Press (2002)
15. Weyns, D., Holvoet, T.: A Framework for Situated Multiagent Systems. In: SELMAS. pp. 204–231 (2006)
16. Weyns, D., Omicini, A., Odell, J.: Environment as a first class abstraction in multiagent systems. Autonomous Agents and Multi-Agent Systems 14(1), 5–30 (2006)
17. Wilensky, U.: Netlogo itself (1999), `http://ccl.northwestern.edu/netlogo/`. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.