An Aspect-based Multi-Agent System

Romain Robbes

Université de Caen - GREYC - CNRS - Romain.Robbes@info.unicaen.fr

Noury Bouraqadi

École des Mines de Douai - Département GIP - bouraqadi@ensm-douai.fr

Serge Stinckwich

Université de Caen - GREYC - CNRS - Serge.Stinckwich@info.unicaen.fr

Abstract

We present how we used Aspect-Oriented Programming (AOP) to ease the development of Multi-Agent Systems (MAS). Our study focuses on the Aalaadin MAS model. We make use of AOP at both the conceptual level of Aalaadin and the implementation level. On the conceptual level, we introduced in Aalaadin AOP concepts. It results in unifying the *group* concept of Aalaadin with the *aspect* concept. This unification relies on reflection to allow the definition of groups with *intrusive* processing. On the implementation level, we used AOP to ease the implementation of a MAS infrastructure. It's worth noting that the aspects we identified are not specific to the Aalaadin model. Indeed, aspects such as *agent message building*, *messaging strategy* or *agent lookup* are applicable in a variety of MAS.

Key words: Aspect Oriented Programming, Multi-Agent Systems, Reflection, Organizational Model, Group, Role.

1 Introduction

Although Object-Oriented Programming (OOP) has been a major progress in software engineering, it has some limitations that led in 1997 to the introduction of Aspect-Oriented Programming (AOP) [1]. OOP suffers particularly from code *cross-cutting* and *tangling*. Indeed, modern software does not only include functional code, but also code describing non-functional properties, such as persistency, remote communications or database management. Each of these properties cross-cuts classes describing entities of the application domain. Moreover, code tangling arises since implementation of such transverse properties are melted in *the base code* (*i.e.,* application "core" code).

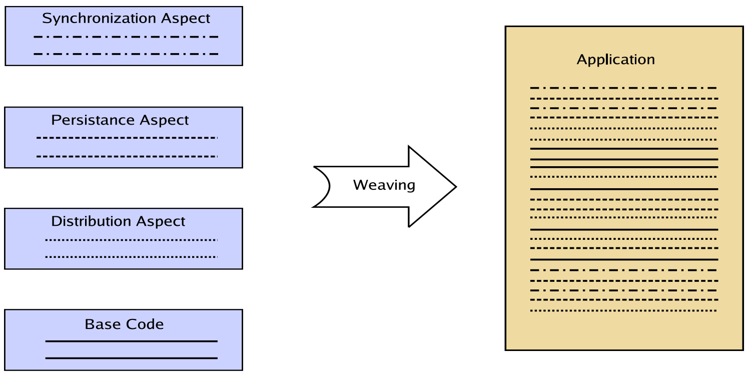

Figure 1. Building an application using AOP

AOP deals with the above mentionned limitations by introducing a new dimension to structure applications [2][3]. It strengthens modularity by allowing developers to isolate the definitions of transverse properties. As shown in Figure 1, each property is defined totally and exclusively in a single module, called *aspect*. Development is therefore simplified, and becomes less error-prone, as developers can focus on one concern at a time. Once the various aspects have been defined, they are assembled with base code to build the application. This integration process called *weaving* is essentially automatic. It consists of linking the aspects with the base code in particular points, called *join points*. These points are either in the base code structure (definitions, ...), or control flow (message sends, field read/write, ...).

In the field of Multi-Agent Systems (MAS), separation of concerns as promoted by AOP is not clearly supported. This is the case of the numerous existing platforms, such as MadKit, Brainstorm/J, JAF, Mobydic, ZEUS, ... [4][5][6][7]. No modularity is provided when it comes to defining non-functional properties even if they are bound to the MAS infrastructure (distribution, security, real-time behaviour) or to the agent's intrinsic nature (autonomy, bounded rationality, organizational aspects). In this paper, we address this issue by exploring two areas where MAS can make use of AOP:

(1) Introducing AOP concepts in MAS models and designs. Indeed, AOP concepts can be used at the conceptual level of a MAS. This facet is the one we will mainly talk about in this article. To our knowledge, it hasn't been studied yet.
(2) Using AOP to implement a MAS infrastructure. A MAS being a complex application, with numerous interrelating, and even sometimes conflicting functionalities, one can easily imagine the advantages of using AOP at this level. So far, some works have been done on this facet [5], [8]. They have identified several generic aspects of MASs (*e.g.,* exception handling, redundancy, persistence) and also specific aspects of agents (*e.g.,* autonomy, collaboration, mobility, learning). But in these works, the agent concept is not first class, as it is build upon a domain object woven with agent aspects.

The paper is organized as follow. Section 2 sums up what has to be known about agents, particularly the Aalaadin [9] model, as our study is based on it. This will bring us in section 3 to discuss the common points between AOP and Aalaadin. Then, we exploit the result of this comparison, to suggest some extensions to the Aalaadin model meant to unify groups and aspects. Section 4 describes and hints at how to implement some infrastructure aspects needed for a MAS. Finally, future works in section 5, and conclusion in section 6 closes this article. A comprehensive example including code from our prototype is shown in an appendix.

2 Background: Agents and the Alaadin Model

2.1 Agents

As H. Van Dyke Parunak said [10], agents are entities able to say “go and no”, *i.e.,* they can decide and act autonomously. An object deals with *how* to perform an action, whereas an *active object* deals with *how and when* to perform it. An agent, on the other hand reflects on *how, when and why* he should perform this action. Hence an agent has a notion of finality which lacks from objects, thus Castelfranchi argues [11] that the agent concept is denied if there is no intention notion.

An agent is an entity which commonly has the following properties:

- Autonomy: an agent decides on its own, taking into account its available ressources: computing time, energy, space ... There is a conflict between this property and the agent conception, since conception explicitely supposes that a function is given to the agent, whereas autonomy offers the agent the

possibility to evolve freely.
- Adaptativity: This property can be seen as a consequence of the previous property, as autonomy makes it necessary to have an open conception that agents can modify.
- Situationness: the agents are continuously interacting with their environment. They sense events happening in it with *sensors*, and can act upon the environment using *actuators*. This environment reported by the sensors can be uncertain and noisy, while the effects of the actuators cannot be guaranteed.
- Agency: This property stipulates that agents organize themselves in groups, thus forming societies, and communicate using messages.

There have been traditionally two epistemological approaches to design agents able to interact in a physically complex environment:

- The symbolic approach [12] adopts a top-down conception and is mainly interested in the inferencing capabilities of an agent. Logic and symbolic programming are then used to implement this behaviour.
- The behaviour-based approach [13] adopts a bottom-up one where simple, reactive behaviours are used. This approach uses non-symbolic technologies.

Even if they are still difficult to conceive, both approaches tend to be merged uniformly, in a *hybrid* approach.

A multi-agent system (MAS) is composed of a set of agents living in a logical or physical environment, interacting with each other and with the environment. These agents can be competiting, or more often cooperating, to solve a given task in the environment. MAS are commonly used in:

- distributed problem solving,
- simulation of ecosystems, chemical or physical phenomenon,
- control of complex systems,
- man-machine interaction, using interface agents.

2.2 Description of the Aalaadin Model

Aalaadin's authors describe it as a meta-model based on organizational concepts such as roles, groups and organizational structures [9]. Its primary goal is to reduce the intrinsic complexity of MAS systems (agent heterogeneity, interaction between different MAS, security . . .). Aalaadin is a meta-model in the sense that it does not force the designer to use a specific agent or organizational model. It rather provides a uniform framework allowing the modeler to define his inter-agents coordination structures, such as hierarchic or market-like organizations.

Aalaadin's main concepts are:

- *Agents* are autonomous and communicating entities, taking *roles* in *groups*. No behavioural or implementation constraints are imposed to them.
- A *group* is a dynamic set of agents. Each agent is a member of one or several groups. New groups can be created by any agent, and an agent must request its admission in a group.
- A *role* is an abstract representation of the functionalities or services of an agent inside a group. Each agent can take the responsibility of several roles, each role being locally associated to a group. Similarly as the admission in a group, being in charge of a role in a group must be requested by the candidate, and may not be allowed. This role notion allows agents to follow several heterogenous dialogues simultaneously. A role is characterized by:
  its *cardinality*, which can be unique or multiple in a group, *i.e.,* only a single agent can wear the role or multiple agents can wear it,
  its *competences*, which are the prerequisites that the requesting agent must fulfill to bear this role in the group, and
  its *capacities*, which are extra competences acquired by the agent when he plays a given role.
- A *group structure* is the abstract description of a group, where the set of roles composing it are described, like the possible interactions between those roles. A group structure is then used to instantiate concrete *groups*.
- An *organizational structure* is built with a set of group structures

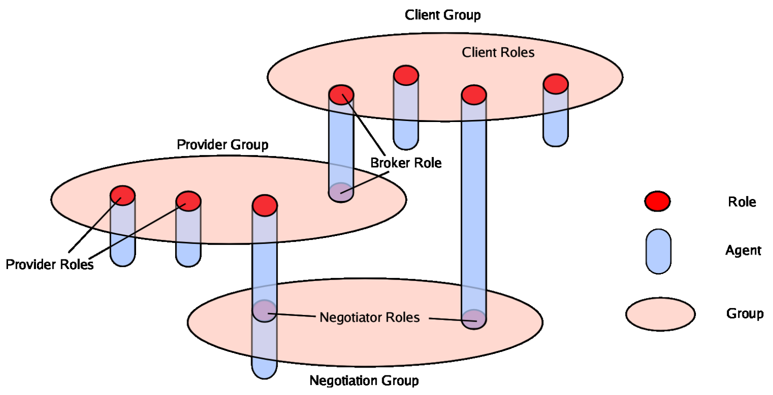

Figure 2. Sample application using the Aalaadin model: the market structure

Figure 2 shows a comprehensive example of the previous concepts. The provided sample application is a market organization structure that includes 3 groups. The provider group includes a provider role, worn by multiple agents, as well as the client role of the client group. Both the provider group and the client group have a broker role that should be worn by a single agent. The broker role consists of allowing clients needing some products to meet the right providers. Clients and providers that get matched this way join a negotiation group to discuss the price. Hence, they wear the negotiator role.

3 Aop at the Conceptual Level of the Aalaadin Model

3.1 Groups vs Aspects

The comparison between Aalaadin and AOP consists mainly in comparing groups and aspects. As we explain below, these two notions are indeed very close, especially as they both describe transverse properties. Because our goal is to import interesting AOP features in Aalaadin, we make the group/aspect comparison based on properties of the latter one. We thus list aspect characteristics and show whether Aalaadin's groups provide them or not.

3.1.1 Transversality

Groups and aspects share the property of being *transverse* to the application they are defined in. They both indeed describe facets of applications.

In object-oriented applications, aspects modify the control flow of computations performed by instances of *several* classes. For example, concurrent access synchronization is a facet of an e-commerce application. This facet, described in an aspect, is tranverse to several objects (clients, orders, products . . .) since computations performed by these objects all need synchronization.

Groups are tranverse to agents in the same manner aspects are transverse to objects. A group indeed describes roles which must be worn by several agents in a MAS. Each role features some capacities (such as access rights) given to the agent wearing the role. Thus the group "modifies" the different agents who join it by augmenting their capacities. For example, a "negotiation" group is defined by the "buyer" and "seller" roles, which must be worn by two different agents. This group is therefore transverse to these two agents.

We can also note that groups have a richer semantic than aspects, as they allow one to define the cardinality of the roles which they are constituted of. In addition, they are more dynamic than most AOP implementations as agents can join or leave groups at anytime.

3.1.2 Weaving and Join Points

The process of weaving an aspect into a given application consists of linking the processings defined in the aspect with those of the application. The corresponding operation in Aalaadin is performed when an agent wears a role to join a group in a MAS. However, these two processes differ on a few points:

- First, computations defined in an aspect are intrusive to the rest of the application: an aspect can modify the control flow of an application and the processings performed by its objects. For example, an authentification aspect can forbid the execution of a message if the given password is wrong. On the other hand computations performed inside a group are much more local. They dont influence directly processings performed in the other groups.
- An immediate consequence of the previous point is the absence of an AOP join point equivalent in Aalaadin. Groups are indeed not directly affecting the rest of the application contrary to aspects. The orthogonality of groups prevents this type of intrusions.
- Another difference is that weaving an aspect can provoke conflicts whereas joining a group does not. Conflicts happen when two aspects need to be woven on the same join point. A log aspect can for example produce a trace when an object receives a message, while an authentification aspect might prevent messages to be executed. To solve this conflict, the integrator must decide whether to always produce a trace, or to restrain it to the messages whose anthentication was successful. Such situations cannot happen in Aalaadin, as groups are orthogonal to each other, so the integration step can be totally automatized.

3.1.3 Modularity and Reusability

Aspects in AOP strengthen modularity of applications, as each aspect is a module describing totally and exclusively a unique facet of an application. It is hence possible to develop separately (in time, or by several persons) each aspect, since there is no coupling between them. This modularity also allows new reusability perspectives: generic application-independent aspects can be spread and reused in different contexts.

Reusability and modularity seems not to have interested Aalaadin's designers. If groups and roles are well identified conceptually, this is not the case for the implementation. This is shown in Aalaadin's reference implementation, MadKit, for which no support whatsoever is provided to modularly define groups and roles. Thus, most of the time, role implementation is hardwired in the agent code, preventing all reuse opportunities.

3.1.4 Functional and Non-Functional Properties

Most of the works in the AOP community are done on non-functional properties of applications. These works suppose that a monolithic base code exists. Aspects are then added to define infrastructure properties.

The situation is different in applications based on the Aalaadin model. Groups are mostly used to define functional facets of the application, as each group supports the interaction between agents trying to solve a domain problem. The application is then defined with groups, the "base code" being the code in the agents unrelated to roles (message handling, . . .). In MadKit, there is also a notion of "system groups" based on "hooks" allowing the definition of non-functional groups. Hooks allow agents to override the way the platform evaluates some primitive operations such as message sendings. Agents can register themselves to hooks. These non-functional properties are not totally satisfactory, as hooks are limited (only one agent can control a hook, composition is thus not taken into account) and seem an *a*d hoc solution (they seem to be implementation-specific and are not mentioned in the Aalaadin model itself). Another limitation is due to the fact that dependencies exist from functional to non-functional properties. Roles in functional groups that require some non-functional property *explicitely* refer to a role to wear in a non-functional group.

3.2 Importing Aspects Properties Into Groups

Our goal is to use AOP in order to benefit the design and implementation of MAS systems, mainly by improving modularity. In the following, we propose an extension to the concept of Aalaadin groups. Groups are already well-suited to the separation of functional concerns. We rely on this feature to support separation of non-functional concerns as supported by AOP.

Even if modularity was one of Aalaadin's main concerns, it is only fulfilled at the conceptual level. The first extension we propose allows developer to implement each group in an individual module, by adding to the transversality found in group the modularity which characterizes aspects. Development is thus simplified as it becomes possible to add or remove individual groups. It also becomes possible to split developement in time or to share it among several persons. And finally, groups can be reused in different MAS applications, much like generic aspects.

Another interesting characteristic of AOP is that aspects are intrusive with respect to the base code. Infrastructure properties can then be defined using these AOP properties. Multi-agent systems need to define such an infrastructure with properties varying among time or applications, so AOP seems to be the best suited technology for this task. Our second extension is the introduction of AOP concepts in Aalaadin. We were however reluctant to have the two concepts of groups and aspect coexisting in the model, given how close they are, so we unifyied them. Furthermore, implementing aspects by means of groups and roles follows the line defined by the reflexive nature of Aalaadin, which "agentifies" services by implementing them with groups of agents collaborating to perform the desired function [4].

3.2.1 Modularizing Groups Using Role Reification

The main goal of this extension is to allow groups to be defined in individual modules rather than dispersed in the body of several agents. This enforces modularity and maximize reusability. Since a group is a set of roles, isolating the definition of a group requires separating role definitions from the agent code.

One could think that since join points are missing from the definition of groups, we could use Smalltalk's packaging mechanism and class extensions to provide group modularity. This is not possible for several reasons :

- This mechanism is not present in most languages. One of the features of AspectJ is that it can add methods to existing classes, which is close to class extensions. Hence using class extensions could be seen as using a subset of AOP.
- Using only class extensions would populate the agent class with all the role code, which is not the desired effect since we want different agent to have different roles. Hence we want to have an entry point in the agent where we can dispatch methods correctly. Having all the roles as extensions to the agent class will also have a much higher probability of conflicts.
- We provide a way to have join points later on, relying on the role reification that is dealt with in the following paragraph.

This separation of roles from the agents bearing them is achieved by the *reification* of roles. They hence become full-fledged objects still existing and manipulable during execution. Wearing a role is then essentially, for the agent implementation, obtaining a reference to an object representing this role. An agent then becomes an "empty shell", able to wear roles necessary for it to perform its task. The agent is additionally responsible of low-level responsabilities, such as message dispatching, life cycle, managing and executing roles . . . .

Once roles are reified, they can be separated from the agent's implementation and be logically put with the group implementation they belong to. The group then becomes a module containing all the code necessary to its execution and can be shipped separately.

On the implementation side, each role is represented by a class. The *agent messages* [1] the role can respond to, are implemented by methods of this role. One responsibility of the agent (as a role manager) is to automate the message delivery by executing the right method on the right role, extracting arguments from the message in the meantime.

[1] These messages are not like object messages, as they are much more complex: they for example contain information about the sender, and group and role information for sender and receiver.

One must note that reified roles have an impact on modelling in addition to implementation. This refinement of the Aalaadin model allows the acquisition and the dynamic transfer of roles from an agent to another, and permits the implementation of meta-roles, which is explained in section 3.2.2.

One benefit of the use of a class per role (and one method per possible request), is that roles can reuse or inherit behaviour from each other by only using mechanisms provided by object-oriented programming. This greatly enhances role reusability.

A similar approach to our reification of roles can be found in Philippe Mathieu's (et al.) MAGIQUE MAS platform [14]. MAGIQUE introduces the notion of reified competences, which are rather similar, though finer grained, to roles. As our model, this allows for initially behaviourless agents to dynamically acquire competences and to specialize them. MAGIQUE does not however have a group notion, and thus does not deal with modularity.

3.2.2 Defining Intrusive Groups Using Meta-Roles

We introduce *meta-roles* in order to allow the definition of intrusive groups. Intrusive groups are groups able to modify the execution of other groups. Hence meta-roles in intrusive groups allows us to define an equivalent of AOP join points when such a behavior is needed [2] . The meta-role concept is directly spawned from the meta-object concept. We first remind below the definition of the latter, before explaining the former.

3.2.2.1 Meta-objects and aspects Meta-objects have been introduced in the fields of reflection and meta-programing in object-oriented languages [15][16]. A meta-object is an object which can act like a "virtual machine" for other objects called "base objects", thus defining the way they behave. For example, a meta-object managing concurrent access can modify the semantics of message sendings so as to control accesses to a base object with semaphores. Such modifications can be defined with a MOP, short for *Meta-Object Protocol*.

One of the uses of MOPs is the exploitation of meta-objects to support AOP [17]. Each aspect is defined with a set of meta-objects which are linked to the base objects where the aspect must intervene. A trace aspect for example is represented by meta-objects augmenting the semantics of message reception with a trace of the message to be recorded in a file. Weaving such an aspect consists of linking the meta-objects to the base objects which need tracing. Thus the meta-object associated to such an object adds a line to the log file each time a message is received.

[2] See section 3.2.2.3, page 11 for uses of this behavior

3.2.2.2 Meta-roles and aspects A meta-role is a role which controls the activity of other roles, and so can reason and act upon their execution. Meta-roles are quite close to meta-objects, as the roles are reified. A meta-role can therefore define the semantics of other roles, and so can act in an intrusive way on base roles which it is linked to via a *meta link*. This action consists of intercepting the control flow in the agent when the base role:

- receives an *agent message*,
- sends an *agent message*,
- requires that the agent enters in a group,
- requires that the agent exits from a group.

This intrusion of the meta-role is materialized for each of the intervention points by a message describing the semantics of this particular point. The set of signatures of these messages is the equivalent of a MOP.

This intrusive capacity allows us to use meta-roles to define aspects. An aspect is more precisely represented by a group which contains meta-roles. We name such groups "aspectual-groups". We thus reach our objective of representing aspects using groups. Facets belonging to the functional code of a MAS are described with groups made of base roles, and aspects representing infrastructure facets are defined with aspectual-groups containing meta-roles.

3.2.2.3 Example We use here the classic example of the market organization (Figure 3), adding the following aspects:

- A logging aspect: the broker traces the messages it sends and receives, to keep proofs in case of contestations for example.
- A cryptography aspect: the client/provider negotiation groups use secure communications.

We implement these aspects using the following aspectual-groups:

- The "Log" aspectual-group contains *1* "Logger" role, and *n* "Logged" roles. Logged roles send a message to the "Logger" role for each message their base role sends or receives. This aspectual-group is instantiated once, and each of the roles of the "Broker" agent are linked with a meta-role "Logged".
- Each "Secure" aspectual-group has two "Crypted" meta-roles. When two base roles establish a communication, their meta-roles exchange public keys in order to cypher their subsequent messages. This aspectual-group is linked to the "Negotiation" group structure. So, an instance of the aspectual-group will be woven with every instance of the Negotiation group. When an agent joins a Negotiation group, it will automatically join the Secure aspectual-group.

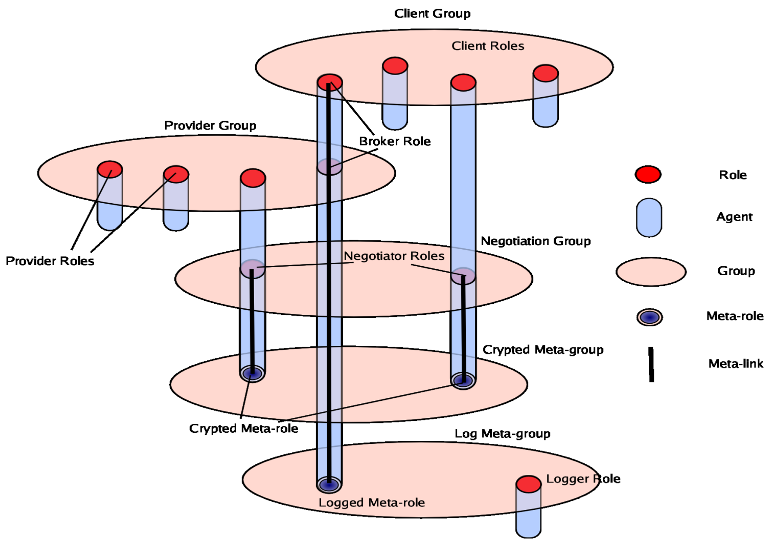

Figure 3. Meta-roles example

3.2.2.4 Weaving The weaving process consists in linking the "functional groups" to the aspectual-groups, by correctly setting meta-links between roles and meta-roles. The controls performed by a meta-role MR on a role R is materialized by messages from R to MR at each join point. To preserve the autonomous property of agents, an agent willing to wear R must prealably wear MR.

In order to keep the modularity and reusability properties defined previously, the meta link between a role and a meta-role is soft. There is no direct reference between roles and meta-roles. In the implementation phase, a role ignores the meta-role it will be linked with. Conversely, a meta-role knows the role it is linked with only at weave time. It is therefore possible to link different roles and meta-roles in different contexts at separate times. This low coupling needs a complementary mechanism allowing us to specify which meta-role will be woven to which role. This is done by group parametrization. The meta-roles to attach to a role are specified as prerequisites to wear the role, so that an agent willing to bear this role must also bear the associated meta-roles.

Please note that to simplify the discourse, we have until now essentially dealt with a *1-1* relationship between roles and meta-roles, but this relation can be possibly as wide as *n-n*. A meta-role in an aspectual-group can indeed be linked with several base roles, and conversely several meta-roles can be linked to a single base role. In the latter case, conflicts such as masking [3] may arise, as it is the case in classic AOP. A possible solution is to use the "chain of responsibility" design pattern, as described for meta-objects in [17].

[3] An example has been given earlier, in section 3.1.2, page 7.

4 Aspects for MAS Infrastructure

We list some infrastructure properties we have detected and which are suitable to be implement as aspects in a MAS platform. When possible, these aspects should be implemented using meta-roles in aspectual-groups rather than full-fledged aspects to follow Aalaadin's reflexive trend.

4.1 Messaging

The messaging strategy, albeit higher level, can also be isolated in a single aspect. This aspect comprises two parts:

(1) automatization of the message building process,
(2) parametrization of the message sending strategy.

4.1.1 Message Building

An object message sent to an agent is automatically promoted to an *agent message*. This avoids the burden of constructing *agent messages* manually, as most of the work can be automated.

An *agent message* have a much more complex structure than a regular object message. The one used here is a dictionary of keys and values allowing the receiver to reflect upon various parameters of the message, not only the request asked, but maybe who the sender is, when the message was sent or received, . . .

The aspect can use the sender role and its context to fill the `senderRole`, `receiverRole`, `group`, `date`, . . . , fields of the message. As for some other aspects mentioned, this aspect's join points are message sends on agent objects. On the receiving side, the agents can add extra parameters to the message before dispatching it to the role, such as the date when the message has been received.

To dispatch an *agent message* to a role, the agent performs two actions. On the one hand, it makes the receiver role refer to the *agent message*. And, on the other hand, it triggers the action to perform by sending an object message to the receiver role. So, when performing the action, the role can access to the *agent message* extra parameters such as the date if required.

4.1.2 Messaging Strategy

There are three messaging strategies so far:

- eager: This is the message sending strategy commonly used for objects. The message is sent to the other agent, and the sending role waits until the message is answered, the answer being received as the return value of the sent message, like in conventional object-oriented programming.
- asynchronous: This is the strategy often used by agents. The message is sent, and control returns immediately to the sender. The message has no answer, if one is needed, it will be given later by the receiver when it sends a message to the sender. The message will be processed normally by the agent, and the receiver may answer as many times as it wants to.
- lazy (using future objects): The message send returns immediately, as in the asynchronous case, but returns an answer like an eager message send. The answer is a "Future" (or Promise) object, of any type. When it it used, two cases can occur: the answer is either already computed, and evaluation proceeds immediately, or evaluation has not finished yet, and the calling role has to wait for it to end. In the current implementation, future objects are actually future roles, who wait for the response in place of the calling role.

The messaging policies are set group-wise to allow maximal flexibility. Messaging policies are high-level aspects, which are set by the application depending on its needs and its implementation, as asynchronous and synchronous messaging are fundamentally different. Lazy and eager policies are easier to swap on the other hand.

4.2 Agent Lookup

The join points of this aspect are the accesses to an instance variable of a role, as each role represents its acquaintances with some of its instance variables. If one of them represents a role, the lookup process is started. An instance variable represents a role if it contains a role, or if its name corresponds to a role available in the group the agent is in with the current role.

The procedure the aspect uses to find a suitable agent is as follows:

(1) The aspect first verifies if the variable really references an active role, and if so, if the role and the agent it represents are still in the system, and are still able to process requests.
(2) If both conditions hold, it proceeds and sends the message to the agent, which dispatches the message to the correct role as usual.
(3) If not, the searching process is started: the aspect searches an agent wear-

ing the correct role in the group the sender is in, sets its target to be the agent it just found, and finally sends the message to it.

A few exceptional handling strategies comes to mind, if no agent is found. They could be implemented using variants of this aspect:

- Wait until an agent holding the role appears in the group, checking from time to time if this event happens. The role sending the message is meanwhile blocked.
- Ask agents with the valid capacities if they can take over the role (including the message sender itself if it is eligible), then send the message to the selected one. One could imagine a variant of this where the search would consider all agents in the system if none is found in the group, asking agents to take the role and join the group.
- Fail, notifying the agent about the absence of such an agent, letting the sending role make a decision based on this information.

The current implementation uses the first scheme only. It proceeds by asking the agents responsible of the group (it wears a "group manager" role) if an agent fulfilling the given role is in the group, and waits on this event.

This aspect allows us to search for available agents more reliably and more naturally, as a reference to an agent instance variable will trigger the searching and verifying process as needed. This frees us from the burden of checking every time that the receiver is still available, hence the increased reliability as well as the increased focus on the functional code. It is of course possible to have various strategies in the different groups of the application or platform, depending on the needs of those groups.

4.3 Distributed Messaging

Distributed messaging must be dealt with transparently if we want to use an application indifferently on a single or on several machines. The use of a remote messaging framework such as SOAP, RST (Remote SmallTalk[4]), XML/RPC, CORBA ..., would provide us with a good basis to build agent distributed messaging on top.

This aspect can deviate control flow when the agent sends a message to another role. The aspect has ways to find where the agent really is and is able to send messages to him, using one of the previously mentioned remote messaging framework if he is outside. Moreover, this aspect could check the location where the agent is before sending the message, thus keeping track of mobile

[4] A framework that supports distribution of Squeak objects

agents, or use different remote messaging frameworks if they want to talk to an agent on a different platform, using a different communication medium.

The aspect must continue its processing on the receiving side, to properly decode the message and make it be received. One could implement this aspect using meta-roles previously mentioned, as this “remote” property could then be set on a by-group basis. Since this aspect was not considered a priority at the time of writing, some problems such as partial failures are not considered, though replication could be used.

4.4 Visualization and Debugging

As multi-agent systems are applications that are rather hard to develop, bugs can often happen. Hence the need to have good debugging tools is crucial. We believe a debuging aspect can be implemented to help the developer in bug tracking. We have implemented a first version of it in the form of a graphical visualisation tool of inter-agent communications. The structure of the application, in terms of groups, agents and roles is materialized onscreen, as are the messages sent between agents. Each agent is shown as an entity linked to the role it wears, and these roles are displayed in the group they are in. The messages exchanged are shown using arrows between the roles.

The screen capture provided in Figure 4 shows the visualization of a market organization with 4 agents (the red squares): two clients, a provider and a broker. Each agent wears several roles in various groups. Arrows link each agents and the roles it wears. Each role is contained in a group, represented by a rectangle.

Some work still remains in order to have a full-fledged debug aspect. Indeed, this aspect should also allow one to show and modify the state of the agents and roles, and to act upon the control flow of the application, as described in the next section.

4.5 Flow Control

This aspect should deal with the evaluation strategy of the agents: should they be all evaluating in parallel, or should they be synchronized? If they are synchronized, should they “step” in a repeatable order? This aspect should also govern whether time is equally shared between agents, or if each agent should have as much time as it needs and a limited number of actions.

This should be a pluggable policy at the group level, as the desirable strategies

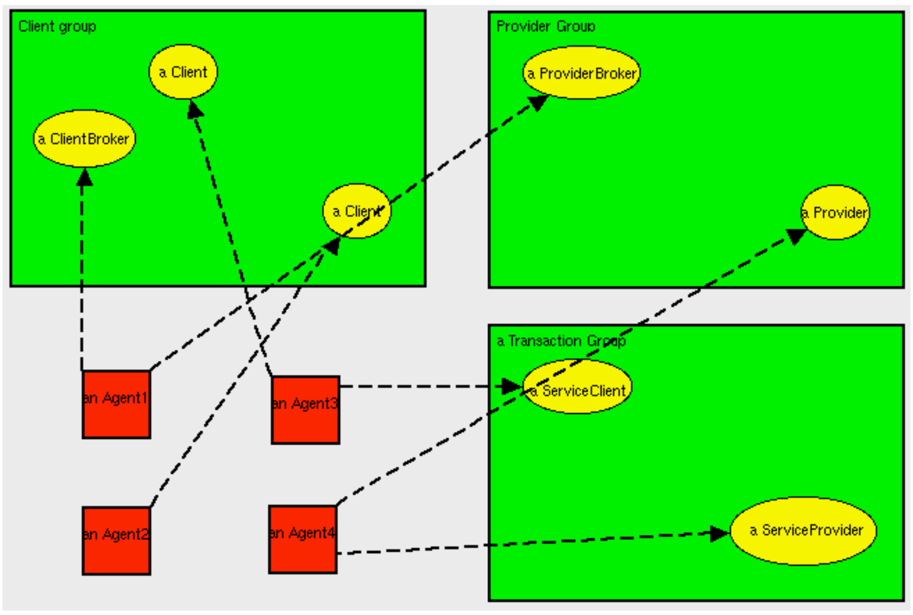

Figure 4. Broker example in Squeak, using the debug aspect

are greatly varying depending on the application designed. Different roles for the same agent could have different time-sharing strategies.

Another part of this aspect is more closely linked to the debug aspect described earlier: this aspect could deal with breakpoint and stepping support, so that one can define when the simulation should halt, and how its evaluation should continue after a halt. It can step at a pace indicated by the user, or restart normally, ... This debugging policy could be set at the group level as usual, but care must be given concerning synchronization-related issues.

5 Future Work

5.1 Improving Identified Aspects

Aspects we identified so far still require some improvements:

- While the messaging aspect is able to do automatic filling of *agent messages*, sometimes manual filling is needed. This could be done with a special syntax such as capitalizing the extra parameters. Suppose for example that we need to set the priority of a message to `urgent`. We could use the following syntax: **broker** provide: #apples atPrice: 200 **P**riority: #urgent (note the capitalized “P” to set a priority field, which is close to the one used in Seld (ref)) .
- Some aspects are designed but not yet implemented. This is mainly the case

of the Remote messaging aspect. Other aspects do not have all the “bells and whistles” mentionned, such as the agent lookup aspect.
- The visualisation aspect could be enhanced to be a debugging aspect if we add the notions of breakpoints, to halt all or parts of the platform on special conditions, and if we allowed groups, agents and roles to be easily inspected.

5.2 Other Aspects Envisioned

5.2.1 Bounded rationality

”Bounded rationality” as defined by Herbert Simon [18] is a compromise between the quality of a solution and the cost invested in computing and interaction. In order to embed such principles in a computer, several approaches have been proposed in the litterature: cost-calculus [19], anytime algorithms [20], approximate processing, ...

As the real-time aspects have been already studied in the litterature, the anytime properties of an algorithm could be considered an ”anytime aspect”. So a working direction would be to see if these algorithms could be extracted out of the base code, and implemented in a generic aspect.

5.2.2 Decision Making

The decision making process governs how the agent reasons and chooses tasks to do. There are indeed several ways for an agent to determine actions it should take next, the most common trends being:

- The reactive behaviour is the simplest. The agents responds a direct stimuli (here a message send) in a predictable way.
- A more complex behaviour can be built upon the latter one by combining several layers of reactive behaviour in a subsomption architecture. In this architecture, several simple reactive behaviours are layered and prioritized, so that basic behaviours are trigerred as needed (*e.g.,* avoiding obstacles), and other, more complex behaviours are trigerred when the situation is less urgent (*i.e.,* path planning).
- The deliberative behaviour: the agent evaluates a set of rules to determine the best action to take given the environment it observes.
- Finally, Markov Decision Processes can be used. In those, a policy is first computed for all possible states the agents can perceive, which associates each environmental state with the optimal action to take.

All these processes should ideally be replaceable, and would all fit in variants of a generic decision-making aspect. This aspect would require reifying the tasks that roles have to do. So, the agent can reason upon, prioritize them, in addition to simply performing them .... This reification effort have not yet been done, but could have a lot of benefits. With this, a role could just be a list of tasks to perform, and could be worn by several agents having various decision methods to perform a role. This would even more separate the content of the role from the evaluation strategy of it.

5.3 Introducing AOP concepts into other MAS Models

We focused in this paper on extending the Aalaadin model with AOP concepts. We have chosen Aalaadin, since it has a good basis for separating concerns with its concepts of groups and roles. Obviously, the way to introduce AOP concepts and the impact of this introduction depends on the extended MAS model. Thus, it should be interesting to conduct other studies with different MAS models.

6 Conclusion

Building Multi-Agent Systems is a hard task because of the multiplicity of simultaneous concerns that developers should take care of. We therefore use AOP to separate the multiple, transverse facets of a multi-agent system and its software platform. In this paper, we focuse on the Aalaadin MAS model.

In order to take benefit of aspect properties in Aalaadin, we unified the concept of group with the one of aspect. We thus made Aalaadin groups inherit the modularity and reutilisability of aspects. This gives groups the same qualities aspects have, and even more, since groups can also be used to decompose the functional code of an application, something AOP can not yet easily do. The unification is done by reifying roles and by introducing the concept of *meta-roles*. Role reification allows the separation of their definitions from the agents ones. Hence, the definitions of all roles constituting a given group can be packaged together within a single module. This enables us to achieve modularity and furthermore allows reuse of generic groups.

The concept of meta-roles permits the creation of *aspectual groups*. Since meta-roles can reason and act upon roles, they allow defining groups that alter the execution of roles belonging to other groups.

We experimented with our model by developping a MAS infrastructure on top of Squeak and MetaclassTalk[21][17]. We defined several aspects of this infrastructure using aspectual groups, acheiving hence separation of concerns.

Using the same broker example as above, the resulting implementation is by far more modular and understandable compared to Aalaadin's reference implementation, MadKit.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: M. Akşit, S. Matsuoka (Eds.), Proceedings of ECOOP'97, no. 1241 in LNCS, Springer-Verlag, Jyväskylä, Finland, 1997, pp. 220–242.

[2] T. Elrad, R. E. Filman, A. Bader, Aspect-oriented programming, Communications of the ACM 44 (10) (2001) 29–32.

[3] R. Filman, S. Clarke, M. Aksit, T. Elrad (Eds.), Aspect-Oriented Software Development, Addison-Wesley, 2004, (to appear).

[4] O. Gutknecht, J. Ferber, MadKit: Organizing heterogeneity with groups in a platform for multiple multi-agent systems, Tech. Rep. 97188, LIRMM, 161, rue Ada - Montpellier - France (Dec. 1997).
URL http://citeseer.nj.nec.com/gutknecht97madkit.html

[5] A. Zunino, A. Amandi, Brainstorm/J: a Java framework for intelligent agents, in: Proc. of the 2^{nd} Argentine Symposium on Artificial Intelligence (ASAI 2000 - XXIX JAIIO), SADIO, Tandil, Buenos Aires, Argentina, 2000.
URL http://www.exa.unicen.edu.ar/~azunino/asai2000.ps.gz

[6] R. Vincent, B. Horling, V. Lesser, An Agent Infrastructure to Build and Evaluate Multi-Agent Systems: The Java Agent Framework and Multi-Agent System Simulator, Lecture Notes in Artificial Intelligence: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems. 1887.
URL http://mas.cs.umass.edu/paper/200

[7] H. S. Nwana, D. T. Ndumu, L. C. Lee, J. C. Collis, ZEUS: a toolkit and approach for building distributed multi-agent systems, in: O. Etzioni, J. P. Müller, J. M. Bradshaw (Eds.), Proceedings of the Third International Conference on Autonomous Agents (Agents'99), ACM Press, Seattle, WA, USA, 1999, pp. 360–361.

[8] A. Garcia, C. Chavez, O. Silva, V. Silva, C. Lucena, Promoting advanced separation of concerns in intra-agent and inter-agent software engineering, in: Workshop on Advanced Separation of Concerns in Object-oriented Systems (ASoC) at OOPSLA'2001, 2001.
URL http://citeseer.nj.nec.com/460915.html

[9] J. Ferber, O. Gutknecht, A meta-model for the analysis and design of organizations in multi-agent systems, in: Third International Conference on Multi-Agent Systems (ICSMAS'98), 1998, pp. 128–135.

[10] H. V. D. Parunak, Go to the ant: Engineering principles from natural multi-agent systems, Annals of Operations Research (Special Issue on Artificial Intelligence and Management Science) (75) (1997) 69–101.

[11] Castelfranchi Cristiano, Guarantees for Autonomy in Cognitive Agent Architecture, in: Wooldridge Michael J, Jennings Nicholas R. (Eds.), ECAI-94 Workshop on Agent Theories, Architectures, and Languages, no. 890 in Lecture Notes in Artificial Intelligence, Springer Verlag, 1994, pp. 56–70.

[12] A. Newell, Physical symbol systems, Cognitive Science (4) (1980) 135–183.

[13] B. R. A., The whole iguana, in: SDF Benchmark Symposium, Robotics Science, MIT Press, 1989, pp. 432–456.

[14] J. Routier, P. Mathieu, Y. Secq, Dynamic skill learning: A support to agent evolution, in: Proceedings of the AISB'01 Symposium on Adaptive Agents and Multi-Agent Systems, 2001, pp. 25–32.

[15] P. Maes, Concepts and experiments in computational reflection, in: Proceedings of OOPSLA'87, ACM, Orlando, Florida, 1987, pp. 147–155.

[16] G. Kiczales, J. des Rivières, D. G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

[17] N. Bouraqadi, T. Ledoux, Aspect-Oriented Software Development, Addison-Wesley, 2004, Ch. 11 – Supporting AOP using Reflection, (to appear).

[18] H. Simon, A behavioral model of rational choice .

[19] E. Eberbach, \$-Calculus Bounded Rationality = Process Algebra + Anytime Algorithms, 2001.

[20] S. Zilberstein, Operational rationality through compilation of anytime algorithms, Ph.D. thesis (1993).

[21] N. Bouraqadi, Safe metaclass composition using mixin-based inheritance, Journal of Computer Languages and Structures 30 (1-2) (2004) 49–61, special issue: Smalltalk Language.