Towards an Aspect-Oriented Approach of Multi-Agent Programming

Matthieu Amiguet, Adina Nagy, José-Antonio Baez-Barranco

Towards an Aspect-Oriented approach of Multi-Agent Programming

M. Amiguet*, A. Nagy†, J. Baez‡.

Abstract

Programming large systems requires to cut them in smaller pieces (modules, objects, components, agents, ...). In all but trivial cases, choices must be made about what will be the pieces, and some aspects are inevitably spread out over several parts. In Object-Oriented Programming, these have been identified as "Crosscutting Concerns", and Aspects-Oriented Programming is about a re-unification of these transversal aspects of the system. In multi-agent programming, one of these crosscutting concerns is interaction.

In this paper, we present a multi-agent model that allows to design agents and interaction patterns independently, as first-order objects, and combine them at runtime. The result is a two-dimensional, orthogonal approach of multi-agent systems.

This model features a very flexible agent architecture due to its componential nature. It is targeted towards independance of agents and of interaction patterns, hence increasing their reusability. The model is formalized in an Object-Z/Statechart mixed formalism, and fully implemented in Java.

1 Introduction

Programming large systems requires to cut them in smaller pieces (modules, objects, components, agents, ...). The aims of such a decomposition are, now, obvious. However, in all but trivial cases, choices must be made about what will be the pieces, and some aspects are inevitably spread out over several parts.

In Object-Oriented Programming (OOP), these have been identified as "Crosscutting Concerns", and Aspect-Oriented Programming (AsOP) is about a re-unification of these transversal aspects of the system to facilitate its build, maintenance, customizability and evolution. While the tendency in OOP is to find similarity among classes and push it up in the inheritance tree, AsOP attempts to realize scattered concerns as first-class elements, eject them horizontally from the object structure and then offer a way to compose them.

In Agent-Oriented Programming (AgOP), the primary structuration is evidently the separation into agents. In this case, important crosscutting concerns include interactions. An "aspect-oriented multi-agent programming" would therefore include the possibility to design interaction patterns separately from the agents and combine them afterwards (at design or run time).

Although much work has been done about communication or interaction patterns in Multi-Agent Systems (MAS), few approaches really allow the programmer to do this. In

*École d'ingénieurs Arc - University of applied sciences, Le Locle, Switzerland, matthieu.amiguet@bluewin.ch

†Institut fur Informatik, Humboldt Universität zu Berlin, Germany, adina_nagy@yahoo.fr

‡LIRMM, UMR 5506, Université Montpellier II, France, baez@lirmm.fr

this paper, we will present an approach called MOCA[1] which is probably the closest existing approach to an "aspect-oriented multi-agent programming" paradigm.

The rest of this article is organized as follows. Next section discusses the state of the art of AsOP. Section 3 exposes the main concepts of our architecture and how they are related. Section 5 introduces related works. And finally section 6 discusses the strength and weakness of our approach, together with research perspectives.

2 Aspect Oriented Programming

2.1 General Presentation

In software engineering, architecture and design, the *separation of concerns* became a long-established principle (many AsOP authors refer to the founding principles expressed by Parnas and Djikstra) and many techniques as role and pattern-oriented programming, subject oriented programming, feature-oriented development, co-exist. Among the separation of concerns directions, the increased interest in Aspect-Oriented Programming techniques appeared from the remark that important concerns of modern applications are not easily expressed in a modular way, but often split all-over the code. Aspect-oriented software development was proposed in order to improve the separation of concerns all along the life-cycle of software development.

According to IEEE, a concern is "a set of interests which pertain to the system's development, its operations and any other aspects that are critical or otherwise important to any of its stackholders". Generally, a concern can be any property or matter of interest in a software system. A *space of concerns* is understood by (Sutton Jr. and Rouvellou, 2002) as an organised set of concerns with their respective relationships. Such a space of concerns has the following properties:

- it is multi-dimensional, in the sense that multiple concerns of multiple types may apply to one software unit at a given time. Multi-dimensional separation of concerns (MDSOC (Ossher and Tarr, 2000)) allows multiple and distinct decompositions of a system to co-exist as modules, which overlap and carve up the system in various ways (Elrad et al., 2001a).

- it is highly structured: concerns may be organised by multiple (in)dependent relationships and they commonly have a hierarchical or lattice-like organisation.

The present section gives a glimpse of the main issues and challenges in AsOP and sets the basic definitions. Notions as *aspects*, *cross-cutting concerns*, *decomposition*, *modularization*, *join points* and *weaving* are recalled.

Aspect oriented programming deals with a particular type of concerns, the *aspects*. Aspects are properties which cannot be encapsulated in a general software unit or - as defined by (Kiczales et al., 1997) - an aspect is "a modular unit of crosscutting implementation." Actually, it is the focus on cross-cutting concerns that really distinguishes AsOP from other separation of concerns methods. Two concerns cross-cut if the methods related to those concerns - that is methods which contribute to their design, description or implementation - intersect. One of the difficult issues about cross-cutting concerns is to understand what cuts across what.

Identifying, separately expressing various aspects of a system and describing their relationships allow developers to design a system out of orthogonal concerns. AsOP techniques

[1]Note that the name clash with the name of the present workshop is accidental!

rely both on this decomposition and on particular mechanism to compose or weave concerns together in an over-all system.

Orthogonal decompositions are represented as distinct modules, at design time. Aspect-oriented techniques aim to preserve the design modularity, while representing a given design structure into a program structure. A software preserves modularity (i.e. independence of key parameters belonging to distinct modules) if independent parameters in the design are represented by independent constructs in the program. As indicated in (Sullivan et al., 2002) one special case where modularity is not preserved "occurs when each of a set of apparently independent design parameters is represented by a corresponding program construct, but where the representation of some other design parameter is distributed (cuts) across and is merged into the previous program constructs."

(Sullivan et al., 2002) distinguishes between two situations: when a *program structure* can be said to be an aspect, and when a *design parameter* can be said to be an aspect. (In this paper we shall see that both these types of aspects coexist in MOCA.)

Other classifications of cross-cutting concerns are

- related to the level implied: high-level notions as the security or the optimization, or low level notions as caching or buffering;
- functional concerns vs. non-functional (systemic) concerns,
- behaviour concerns vs. structural concerns, etc.

We have seen that the purpose of AsOP is to provide tools for separating components (objects) and aspects (cross-cutting concerns) from each other.

It implies that AsOP deals with two types of first-order elements: components of the structural realization of the system (objects, classes, interfaces, etc), on the one hand, and scattered concerns, whose identification allows them being extracted from the program structure, on the other hand. Behaviour that would have been spread over the final code is finally congealed in a single structure, by aggregation of aspects into cross-cutting concerns.

The separation of objects and aspects requires a mechanism to recombine (weave) them later, into an over-all system. A compiler, called *weaver*, intertwines the objects (describing functions requested for system) and aspects (that describe cross-cutting concerns). The central notion in the weaving process is the one of join points. A *join point* is generally an element of the program's semantics with which aspect elements coordinate; that is a point where a hook is placed to combine objects and aspects. According to the way concerns and objects are woven, we distinguish two types of composition: static, when the composition of concerns is realized at compile time, and dynamic, that is done along the system's execution. MOCA relies on a dynamic composition of concerns.

Among of the main problems to develop an aspect-oriented system (Elrad et al., 2001b) and (Elrad et al., 2001a) mention the following which are relevant for MOCA:

- *modularization*: the way the AsOP system specifies aspects (aspect description), defines the join points and the behaviour at join points, etc.
- *composition* mechanisms (weaving), deciding whether a dominant decomposition exists or whether the system is based on multidimensional separation of concerns, where concerns belonging to various dimensions are treated as equals;
- visibility of aspects and *conflict mechanisms among aspects*
- *implementation mechanisms*, including - among other features - the decision whether compositions are determined statically or dynamically

2.2 Some Important Approaches

2.2.1 Hyper/J

Hyper/J (IBM, 2004) is based on the notion of *multi-dimensional separation of concerns* (MDSOC) (Ossher and Tarr, 2000). Its objective is to permit the encapsulation of arbitrary kind of concerns and the integration of separate ones. This approach allows developers to decompose their software so it encapsulates all relevant kind (dimensions) of concerns simultaneously, without more than those necessary and without one dominating the others. MDSOC also includes composition ability to allow developers to integrate these separate pieces. A concrete realization of MDSOC for Java is the tool called Hyper/J.

Hyper/J allows a developer to compose a collection of separate models, called hyperslices, each encapsulating a concern, by defining and implementing a (partial) class hierarchy appropriate for that concern. The models typically overlap, and might or might not cut one another. Each model can be understood independently. Hyper/J doesn't require a dominating class hierarchy and doesn't make any distinction between *classes* and *aspects*, allowing any hyperslice to extend, adapt, and be integrated with another one. Hyperslices are loosely coupled, since they never refer directly to one another. This point is achieved through abstract methods: an abstract method in an hyperslice has to be concreted in another one. The developer can integrate separate hyperslices in an unique Java class. To do so Hyper/J operates on class files using hypermodule declaration indicating which hyperslices are to be composed to obtain the final program.

2.2.2 Composition Filter

The filters in the composition filters (CF) (Aksit et al., 1992) model can express crosscutting concerns by modular and orthogonal enhancements to objects. Modularity, since filters have well-defined interfaces and are conceptually independent of the implementation of the object. Orthogonality, since filter specifications do not refer to other filters. Since the observable behavior of an object is determined by the messages it receives and sends, filters can express category of concerns such as inheritance and delegation, synchronization, real-time constraints, and interobject protocols. Filters are attached to objects as enhancement. A CF class aggregates zero or more internal classes and composes their behavior using one or more filters, divided in two sets: input and output filters. An internal class may be again a CF class or it may be a standard one.

Whenever a message arrives at the filter, a matching is initiated. The matching process primarily involves the targeted (named object) and the selector of the message. If the matching succeeds, the message is said to be accepted by the filter (and it is sent to the target), if not, the message is said to be rejected. Some filter types are:

- *Dispatch* which sends the message to its target if accepted, if not continues with the next filter;
- *Error* which sends the message to its target if accepted, if not an exception is raised;
- *Meta* which sends the message as parameter of a new one to a new target if accepted, if not continues with the next filter.

2.2.3 AspectJ

In AspectJ (Kiczales et al., 2001), a join point is the point in the execution flow of a program when a method is called, and when method returns. Each method call itself is one join

point. Pointcut designators are used to identify a subset of all the join points in the program flow. Programmers can define named pointcut designators, and pointcut designators can identify join points from many different classes, AspectJ also allows specification of a pointcut in terms of properties of methods rather than their exact name. In AspectJ, advice declarations are used to define additional code that runs at join points. *Before advices* run before the method begins running. *After advices* run just after the method has run. *Around advices* run when the join point is reached and has explicit control over whether the method itself is allowed to run at all. Pointcut designators can expose certain values in the execution context at join points and those values can be used in advice declarations. Finally, AspectJ is easy to use but the fact that aspects often cannot be understood without references to their model reduces their reusability.

2.2.4 DJ Library

DJ Library's aim is to allow Java programmers to experiment with AsOP without having to learn an extension of Java or to preprocess source files. This library is based on the idea of adaptive methods (known in (Lieberherr, 1996) as propagation pattern). Such a method encapsulates the behavior of an operation into one place, avoiding to disseminate an operation across several classes, but also abstract over the class structure, avoiding to tangle to much information about the structure of classes. To do this, the behavior is expressed as a high-level description of how to reach the classes participating in an operation (a traversal strategy) and what to do when the participants have been reached (adaptive visitor). The class structure (to traverse) is directly the one used by Java at run-time: the DJ Library exploits the reflection ability of Java to know it. The join points are those defined by the traversal of the structure. Then the class structures, the traversal strategy and the visitors can easily evolve independently. This library is easy to use but only a limited class of behavioral concerns can be expressed as adaptive methods.

2.3 Overview of Recent Works

Although recent works on aspects are often Java-based, many other languages benefit from aspectual extensions: C++, C# and the .NET framework, Smalltalk, Python, Perl and Ruby among others (see (The AOSD Steering Committee, 2004) for details). Works are also done towards an aspect-oriented extension of UML (Clarke and Walker, 2002).

Whatever language is used, one key issue is when and how classes and aspects are "woven". Many approaches do this at compile time; however, other possibilities exist:

- Load time weaving. This can be very useful e.g. in situations where one wants to modify or adapt a third-party library of which one doesn't have the sources (Kniesel et al., 2004).
- Run time weaving (also called *dynamic weaving*). This kind of weaving allows for dynamic program adaptation to runtime circumstances (Schult and Polze, 2003; Popovici et al., 2002)

As a general rule, a later weaving is more flexible, but also entails more overhead and a greater difficulty to ensure correct execution.

Modularity and *reusability* are two keywords of AsOP. As they also are center concepts in component-based approaches, it is no wonder to find several works about aspect-oriented components. (Grundy, 2000) proposes a decomposition in "vertical slices" (components) and "horizontal slices" (aspects) to improve readability and reusability of code. The Dynamic Aspect-Oriented Platform (Pinto et al., 2002) proposes a similar approach including

runtime weaving of aspects. QCSS (Sassen et al., 2002) proposes a methodology based on aspects, components and contracts to guarantee functional behaviour, structural properties and synchronization.

Although AsOP is a relatively new area of research, several applications have been proposed in very various fields like prefetching in OS development (Coady et al., 2001), bug tracking (Bodkin, 2003) and Quality of Service (Schantz et al., 2002).

3 MOCA: An Aspect Oriented Multi-Agent Approach

In the context of this workshop, we suppose the reader familiar with general Agent-Oriented programming concepts and we will directly proceed to the presentation of the MOCA model. For the sake of clarity, we will provide a presentation in a traditional multi-agent style and delay the AsOP discussion until the end of the section.

From an Agent-Oriented point of view, MOCA has the following characteristics:

1. It is a dynamic behaviorist organizational model. This means that agents can take and leave roles at runtime, and that taking a role entails to follow a given behavior.
2. Agents have a componential architecture.
3. Agents can take several roles simultaneously, and a role conflicts management mechanism is provided.
4. The model is fully implemented in Java above the MadKit multi-agent platform.

The next section will briefly present MadKit and its underlying model Aalaadin, as well as their relationships to MOCA. Section 3.2, will then present the main concepts of the model; sections 3.3 and 3.4 will go into deeper detail. Sections 3.5 and 3.6 will give an overview of organizational dynamics and role conflicts management. Finally, section 3.7 will discuss links between MOCA and AsOP.

3.1 Aalaadin, MadKit and MOCA

MOCA is based on the Aalaadin methodology (Ferber and Gutknecht, 1998), which includes two conceptual levels: the *concrete* level, which includes the notions of *agent*, *group*, and *role* (only as names for agents in a group), implementing the actual multi-agent structure; the *abstract* level describes valid interactions and structures of groups (which we call organizations). However, the abstract level is methodological: although the concrete level is fully formalized (Ferber and Gutknecht, 1999), the abstract level is never formalized nor implemented. In (Hübner et al., 2002), both levels are formalized but only from a specification perspective.

MadKit (Ferber et al., 2004) is a multi-agent platform based on the Agent-Group-Role model (Ferber and Gutknecht, 1998), on which the Aalaadin methodology is targeted (Ferber and Gutknecht, 1998). The services provided by this platform are of two kinds: (i) The agent services (Management of messages, Agent life cycle and acquaintances); (ii) The group services (group creation, entering and leaving, information).

Note that a group is explicitly not a recursive notion in MadKit in the sense that a group cannot be an agent.

Even if methodologically the groups are thought as instantiations of organizations and roles are seen as instances of behaviour patterns, this is not implemented in the platform. In fact, roles and groups are just names for structuring the MAS architecture. In order to

overcome this limitation, we want to introduce explicitly *organization* and *role descriptions* as first order objects of which groups and roles are instantiations. Therefore, our contribution with MOCA is to propose formalizations for representing the concepts of Aalaadin and to enrich MadKit with a reification of the *abstract* level concepts, thus providing a complete, operational formalism to build multi-agent systems in an organizational-centered approach (Lemaître and Excelente, 1998). MOCA is implemented in such a way that any multi-agent platform providing communicating agents and group structures could be used in place of MadKit.

Additionally, to ensure orthogonality and hence improve reusability of organizations and role descriptions, we limit the possible interactions to those that take place inside a given group. The only opportunity for two different groups to exchange and cooperate is when some agents participate in both of them at the same time. This last option is mentioned in the Aalaadin methodology but not currently enforced by MadKit.

3.2 Overview of MOCA Concepts

To describe the organizational structure on the one hand and the multi-agent system on the other, our model is composed of two different levels: the descriptive or organizational level and the executive or multi-agent level. Additionally, we distinguish the external or behaviorist point of view on the system from the internal point of view of the agent. In the external point of view we observe interacting (through influence) agents achieving acquaintance structures or groups whose recurrence can be abstracted away as relational structures forming organizations. In the internal point of view, we describe the agent architectures classified into agent types (Reactive, deliberative, BDI, etc.) and providing competences. Role descriptions and their instantiations articulate both by providing at the same time the recurrent pattern of interactions from an external point of view and the way it is achieved from an internal point of view, using and relying on agent competences. Most probably, separating these two aspects would allow further flexibility but have not been taken into account in MOCA. Figure 1 represents the important concepts for each level and from each point of view and the correspondence between them. Figure 2 describes the links between the internal concepts.

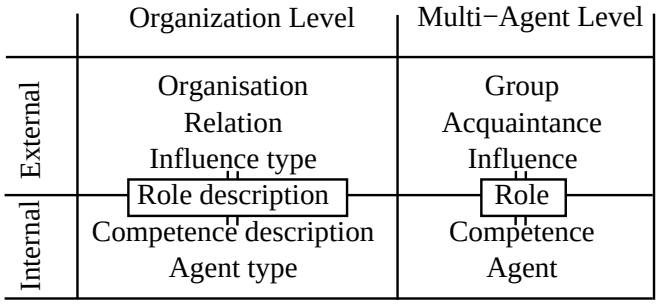

Figure 1: The concepts of MOCA

Most of the notions of the MAS level exist in MadKit, except for the notions of *influence* and *competence* which will be described further in this section. Our major contribution from an AgOP point of view is hence the explicit representation of the Organization level.

Agents in MOCA must have a very flexible architecture because of their possibility to take and leave roles at runtime. To achieve this flexibility, we chose a componential approach. A full description of the componential aspects of MOCA is out of the scope of

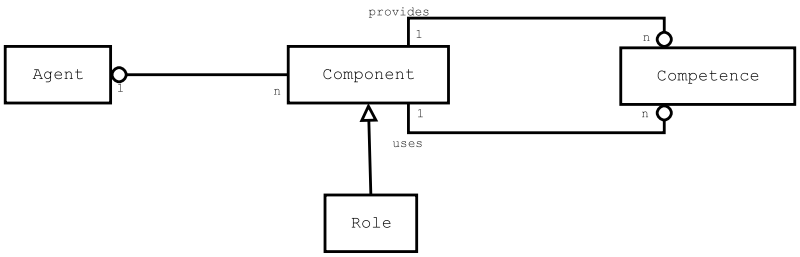

Figure 2: The internal concepts of MOCA

this paper, but this presentation will introduce the minimal concepts needed for a good understanding of the model.

In the next subsections, we will describe in detail the notions of figures 1 and 2.

3.3 The Organization Level

3.3.1 Organizations and Relationships

An organization is defined as a recurrent pattern of interactions from a given point of view (financial exchanges, goods exchanges, ants collective behavior, etc.) and very often attributed with a rationale (ensure equilibrium, feed the ant colony, etc.). In our case, it is formalized as a set of role descriptions and relationships between them. Cardinalities can be specified on both role descriptions and relationships, to describe how many times they can be instantiated as roles and acquaintances, respectively. Figure 3 pictures an example of an organization for the FIPA Contract Net protocol (Collective, 2003). There are two role descriptions in this organization: the initiator and the participant; in a given instantiation of the organization, there can be only one initiator, but of course several participants, each of them being possibly in acquaintance with the initiator.

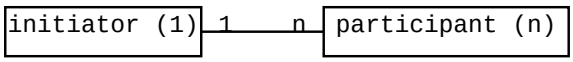

Figure 3: A simple organization for the Contract Net Protocol

3.3.2 Influence Types

The influence type specifies the kind of interaction a role can receive and generate. In order to be general, we use the term influence to cover in the same abstraction forces for physically situated roles, events and up to speech acts (ACL, KQML, etc.) for socially situated roles (Ferber and Müller, 1996).

3.3.3 Role Descriptions

A role is defined as a recurrent behavioral pattern within an organization. The role description represents the way this behavioural pattern is generated, being the internal account of the external pattern. To formalize it, we chose to use a formalism strongly based on (Hilaire, 2000), which is a combination of Statecharts (Harel, 1987) and Object-Z (Duke et al., 1991). The formalism is quite intuitive, so we expose it quickly on an example, without further formal description (for details refer to (Hilaire, 2000)).

Figure 4 pictures a typical role description corresponding to the initiator role of figure 3.

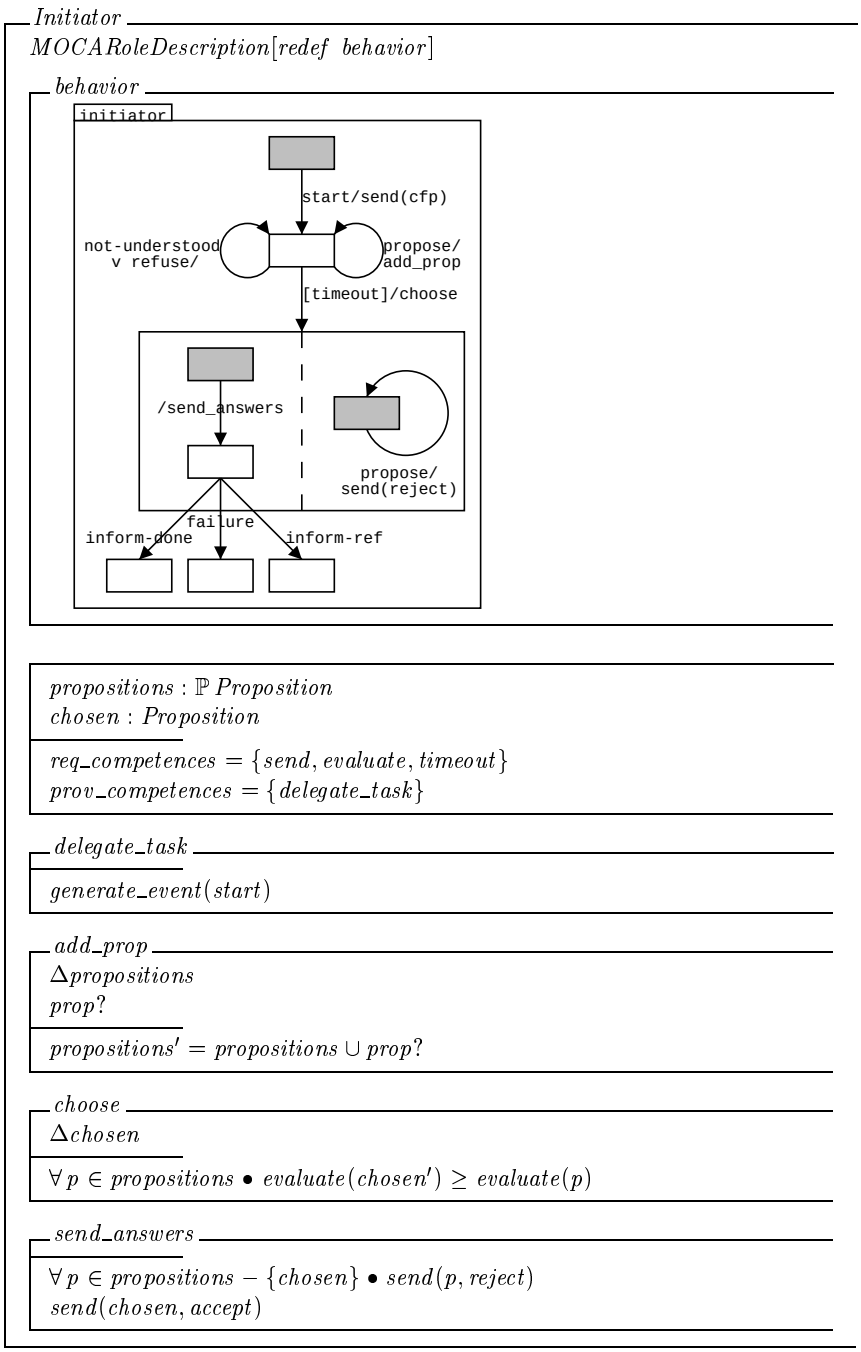

Figure 4: The *initiator* role description

This description is based on Hilaire's framework (Hilaire, 2000), which can be seen in the inheritance statement of the first line. The heart of the description is of course the statechart specifying the behavior. Statecharts are an extension of finite state automata used in UML, with and-or hierarchy of states; hence the dotted line in figure 4 means that the two halves of the super-state have to run in parallel. Note that we use a slightly non-standard notation for default states, which are pictured as gray boxes.

Transitions are labeled with three-parts expressions of the form *influence[condition]/action*. This means that the arrival of *influence* triggers the firing of the transition, which can take place only if *condition* holds. If the transition is fired, then *action* is executed.

Each part of the expression can be omitted.

Actions can be calls to agent competences (e.g. *send*); the required competences of the agent must be listed (variable *req_competences*) in order to check if the agent is able to take the role. Usually, at least *send* will be listed there, as it corresponds to the capacity of sending influences to other agents. Our model also allows for internal methods and variables, specified in object-Z notation (e.g. *add_prop*).

An important feature of MOCA roles is that they can also *provide* competences to the agent. Thus endorsing a role can give the agent new abilities, permanently or not. For example, the role described here provides the competence *delegate_task*, which can be used by another role in the agent to delegate a task to another agent. When this competence is called, the execution of the role is started (through the generation of the *start* event).

From a componential point of view, roles are a particular kind of component, and required and provided competences correspond to its input and output ports, respectively.

3.3.4 Competence Descriptions

The competence descriptions are specifications of the services an agent has to provide in order to be able to play the role; typical examples are the ability to send and to receive messages, to compute an offer or evaluate between offers for a contract net. This notion allows the description of the role at a very abstract level, essentially the management of interactions.

3.4 The Multi-Agent Level

At the multi-agent level we find the instantiation of the various concepts at the organizational level.

3.4.1 Groups

A group is an instance of an organization made of agents playing the various roles of the organization. The only difference from the same notion in MadKit is that this instantiation is made explicit.

3.4.2 Roles

The role in MadKit is just a name in a group; it is not expected that roles with the same name in different groups are linked to the same behavior. In our higher abstraction level, two roles with the same name in two different organizations can produce different behaviors, but two roles with the same name in two different groups instantiating the same organization will have the same behavior. Each role in an agent is the instantiation of the role's state (automaton state and internal variables) and a reference to the role description.

3.4.3 Acquaintances

The acquaintances of an agent are, as usual, the agents it can communicate with. The link between acquaintances and relationships is that an agent can be in acquaintance with another agent if and only if there is at least one relationship between the roles played by the respective agents.

3.4.4 Agents, Influences and Competences

An agent is an autonomous entity able to create new groups by instantiating organizations, to take and to leave roles in existing groups. An agent can play none to many roles at any given time. The typical execution of an agent includes:

1. Get the influences coming from other agents dispatch them on the different roles.
2. Manage components communication (incl. competence calls).
3. Send the influences emitted by its roles to other agents.
4. Control the execution of its components in order to avoid negative interferences.
5. Manage its components (incl. taking and leaving roles).

The three first tasks are more or less straightforward.

The fourth one is more problematic and will be described in section 3.6.

As for the last task, it is out of the scope of MOCA: the very idea of the approach is to provide tools that allow an agent to take and leave roles at runtime, but the reason why he would do so is left to the designer of the system.

3.5 Organizational dynamics

In order to be able to create a group instantiating an organization and to enter and leave it, we need a mechanism to do so. This is clearly a "meta" level with respect to the organizational one, but to preserve conceptual homogeneity of our model, we choose to implement this mechanism as an organization.

The idea is to have a *Managing group* in the system in which all agents take part[2]. The corresponding *Managing organization* has three role descriptions (RD's) which are the *YellowPages* RD, the *Manager* RD and the *Requester* RD.

We are not going to describe the details of the organizational mechanisms in MOCA (for more details refer to (Amiguet, 2003; Amiguet et al., 2002)), but they allow for group creation and management without any inter-group communication. However, if we want our system to be more than a few groups functioning independently, we have to provide a collaboration mechanism between groups. The way to do this without introducing explicit inter-group communication is to allow agents to have several roles in different groups at the same time.

Hence the local coordination of roles inside the agents achieves groups coordination at the system level. The way roles are coordinated by an agent is of course part of the agent's autonomy, but in order to ease the agent's design we provide a generic and powerful implicit communication mechanism: we allow roles to provide competences to the agent. Hence the role executions rely only on agent competences, but the agent has the possibility to use roles to implement some of them. This allows elegant implicit role communication inside the agent, but to be effective, it requires a way of dealing with role conflicts.

3.6 Role Conflicts Management

Whenever roles imply a given behavior, the problem arises of roles conflicts; to cope with that, we introduce the notion of *mutual ignorance*. A system is said to ensure mutual

[2]In a large system, it is possible to have several of them to avoid a bottleneck effect.

ignorance on roles if roles can execute without having to know whether the agent has other roles running.

In MOCA, mutual ignorance is generalized on components execution and is ensured by the following mechanism:

1. Whenever a component A calls a competence of component B, A is registered as a user of B. When A makes its last call to this competence, B will be notified automatically.

2. Component B has the possibility to *accept* or *reject* competence calls. An *acceptance policy* must be provided to determine whether B accepts one or several users, or one "full" user and several "read-only users", etc.

3. If the call is rejected, A will *block* the corresponding transition in its statechart. B will notify A when it is ready again to get calls.

This provides only a very coarse idea of the role conflict mechanism in MOCA (for more details refer to (Amiguet, 2003)). The result is a mutual exclusion on zones which are automatically determined at runtime depending on the participating components and the nature of the involved resource. This mechanism can be proved to ensure mutual ignorance, and this with *minimal exclusion zones* (Amiguet, 2003).

3.7 MOCA and AsOP

The MOCA approach allows the programmer to design separately agents on the one hand and organizations on the other hand, and then combine these elements freely in a running system.

This is clearly a two-dimensional separation of concerns: agents, as the primary decomposition, form "vertical slices" and organizations form "horizontal slices". Orthogonality is ensured by the two characteristics that agents cannot communicate outside of a group and groups cannot communicate without a shared agent.

In other terms, in any MAS, interaction protocols crosscut over agents. MOCA allows them to be represented as unified aspects, called organizations.

This is the main contribution of MOCA towards an Aspect-Oriented Agent Programming. However, the treatment of other (crosscutting) concerns can also be simplified with this approach: see (Ferber and Gutknecht, 1998) for a reflexion about how MadKit-based approaches enhance security and (Amiguet, 2003) for perspectives on formal validation of MOCA-based systems.

From a pure AsOP perspective, MOCA has the outstanding characteristic that it offers runtime weaving of aspects including a mechanism to resolve conflicts.

4 Example of Use

This architecture has been tested on a number of examples among them the famous prey/predator benchmark (Müller et al., 2001) and a foot-and-mouth epidemic simulation. This last example was first proposed by (Durand, 1996), and was restated in a somewhat simplified formulation by (Hilaire, 2000). We have based our experiment on Hilaire's model.

Figure 5 pictures the structure of the system, which consists of three agent types and two organizations. The meaning of the *Stockbreeder* and *Livestock* agent types is

straightforward; note that the *Disease* has also been reified as an agent. We now quickly describe the two organizations:

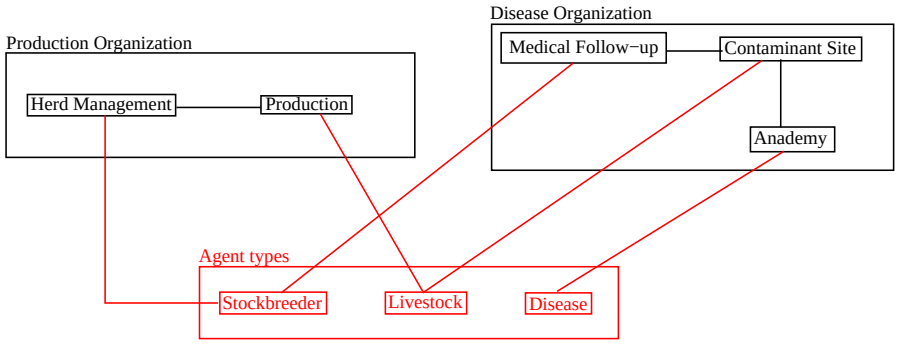

Figure 5: The foot-and-mouth epidemic example

The Production Organization is composed of two roles: the *Herd Management* role is responsible for selling animals, and moving them to grass-land in spring and back to stalls in fall. The *Production* role is responsible for simulating the birth of calves.

The Disease Organization is composed of three roles: the *Medical Follow-up* role is responsible for detecting the presence of the disease and initiating a treatment; The *Contaminant Site* role deals with the effect of the treatment and the *Anademy* role simulates contagion and healing.

A more complete description of this simulation, including the full specification of the roles, can be found in (Hilaire, 2000).

Figure 6 pictures the result of the simulation: the number of sick animals increases in winter, because they are located in stalls, were contamination rate is high; it decreases in summer when the animals are outside. The total number of cows varies as a result of the births and selling. Our results are very similar to those of (Hilaire, 2000) and could be obtained with a low cost in term of development time.

5 Related works

As noted in (Garcia et al., 2002), software engineering approaches for MAS development fall in two categories: (i) agent-based software engineering, and (ii) object-oriented software engineering for agent systems. Both can take advantage of aspect-oriented approaches.

(Garcia et al., 2002) is an example of an aspectual approach of the second category, where aspects are used to modularize agent capacities and properties. Note that the aspects usually don't cut across agents; the approach is therefore quite similar to coarse-grained component-based approaches like Voyelles (Ricordel, 2001).

MOCA falls into the first category. We don't know of any other agent-based SE approach explicitly referring to AsOP; however, several organization-based works bear some similarities with our proposal.

As in MOCA, the focus of (Durand, 1996), Aalaadin (Ferber and Gutknecht, 1998), (Ferber and Gutknecht, 1999), Opera (Dury, 2000), the model of (Hilaire, 2000), MOiSE+ (Hübner et al., 2002), Gaia (Wooldridge et al., 2000), and other related models is on the organizational structure of a multi-agent system, as well as on the study of agents'

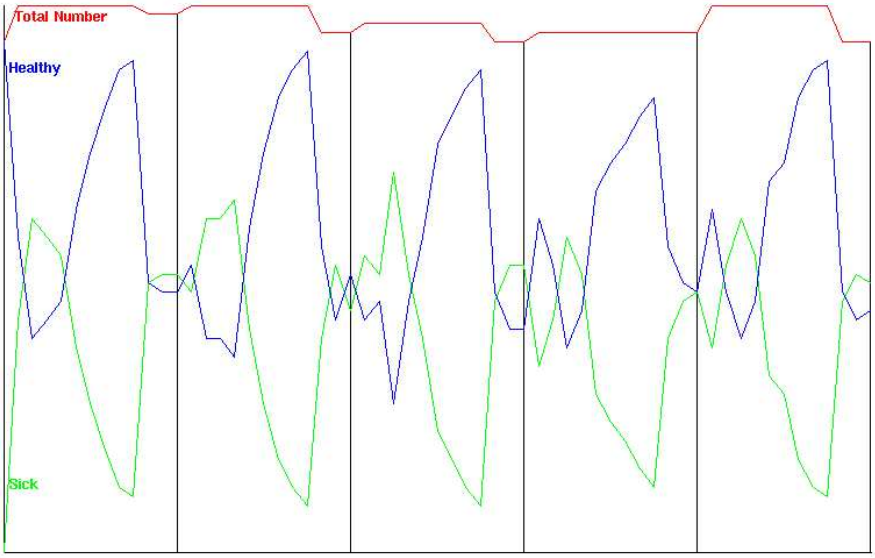

Figure 6: The results of the simulation

behaviour induced by this structure. (Unlike in MOCA, social dynamics is missing in several of these models).

The works on Aalaadin (Gutknecht, 2001) and MadKit (www.madkit.org) are based on the Agent-Group-Role model which allows for a structuration of the MAS in levels and for an organizational dynamics, while considering concerns as security, heterogeneity of agents and reusability. However, roles in Aalaadin are not given a behavioural content and the notions of organisation and group are *abstract concepts*. Cassiopee (Collinot et al., 1996) relies on the expression of both collective behaviours and elementary behaviours of agents. It follows that the behavioural dimension is strongly present, but the model has no structuration in levels and no reified organizational concepts. MOCA inherits equally from Hilaire's Object-Z/statechart model (Hilaire, 2000), employed to represent behaviours associated to roles, which allows for the abstraction and formalization of collective behaviours. While Hilaire's model covers the design and test of organizational structures, it neither covers the conceptual dimension, nor allows for organizational dynamics or conflict management for multiple role endorsement. Opera (Dury, 2000) is built around a specification of interactions, which is revisited within the *Role description* concept of MOCA. However, MOCA allows for richer behaviours associated to roles, and for a finer management of role conflicts within an agent.

In (Wooldridge et al., 2000, and related papers) Gaia represents an organizational methodology for agent-oriented analysis and design, which bears some similarities with MOCA, as it is based onto two abstract models - a "mixed" role model and an interaction model. While Gaia is a valuable methodology for Agent-Oriented Software Engeneering, organizational concepts are present only at design phase while nothing is said about multi-organizational systems and multiple role endorsement.

In MOISE + (Hübner et al., 2002; Hübner, 2003), whose initial purpose was to reconcile organizational and global-plan centered models, three dimensions are used to describe a MAS: the structure expressed through roles and groups, the functioning (global plans and tasks) and deontic relations or other norms (agents' obligations, norms, responsibilities, permissions etc.) The structural dimension is centered on three main concepts - roles, role relations and groups - which are employed to build the individual, social, and collective

structural levels of an organization. MOISE + doesn't rely on a behavioural definition of roles, but sees a role as a set of constraints that an agent ought to follow when it accepts to enter a group. The constraints imposed by the role are defined in relation to other roles (in the collective structural level) and in relation to global plans. the social level is used to specify relations between roles, that is links used to constrain agents after they have accepted to play a role. The collective level imposes some constraints regarding the roles an agent can play at the same time. The functional dimension describes the way global goals are decomposed by plans and distributed to the agents by missions. This decomposition supposes, at collective level, a global plan decomposed as a social schema and, at individual level, missions an agent may commit to. Deontic dimension relates the structural and functional dimensions. The functional and deontic dimensions insure an organizational dynamics. Besides the attribution of behaviours to roles, the reification of the notions of organization and role, MOCA relies on other choices different from MOISE +: on a specification of roles in the organizational structure (lack of inheritance) and on the fact that role endorsement is not only multiple but equally a matter of agents' autonomy, as driven by agents' internal goals.

6 Conclusion

In allowing the programmer to design organizations and agents independently, the MOCA model can be seen as an Aspect-Oriented approach of agent-oriented software design. From this point of view, agents form "vertical slices" and groups "horizontal slices" in the MAS. MOCA enforces a complete orthogonality of these slices, thus allowing to develop very reusable modules (both agents and organizations).

For the time being, MOCA has only been tested on relatively small systems (Amiguet, 2003; Amiguet et al., 2002). One important perspective is therefore larger scale testing. As MOCA is planned to be included in the next version of MadKit, this will probably encourage its use on larger-scale systems.

The current implementation, though functional, is more a proof of the concept than a production-ready platform (due, among others, to its heavy use of Java reflection). A reimplementation is in work as a part of the MIMOSA project (Collective, 2004) that should result in a more efficient and user-friendly platform.

Objects, components and agents provide very powerful structuring paradigms for building software systems. However, a stronger structuration also implies a stronger splitting of some concerns over the different parts of the system. It is therefore necessary to explore ways of dealing with these crosscutting concerns to keep them under control. MOCA is an attempt to provide solutions to some of the problems occurring along this way. Even if this model is far from perfect, we think that multi-agent programming could benefit a lot from a deeper understanding of AsOP techniques. Along the way, techniques could be developed (like, MOCA's conflicts management mechanism) that are of interest to the AsOP community.

References

Aksit, M., Bergmans, L., and Vural, S. (1992). An object-oriented language-database integration model: The composition-filters approach. In Madsen, O. L., editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 372–395, Berlin, Heidelberg, New York, Tokyo. Springer-Verlag.

Amiguet, M. (2003). *MOCA: Un modèle componentiel dynamique pour les systèmes multi-agents organisationnels*. PhD thesis, Université de Neuchâtel.

Amiguet, M., Müller, J., Báez-Barranco, J., and Nagy, A. (2002). The moca platform: Simulating the dynamics of social networks. In Sichman, J. S., Bousquet, F., and Davidsson, P., editors, *Multi-Agent-Based Simulation II*, number 2581 in LNAI, pages 70–88. Springer-Verlag.

Bodkin, R. (2003). aTrack project home. `https://atrack.dev.java.net/`.

Clarke, S. and Walker, R. J. (2002). Towards a standard design language for AOSD. In *1st International Conference on Aspect-Oriented Software Development*, Enschede.

Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N., and Ong, J. S. (2001). Structuring operating system aspects. *CACM*, 44(10):79–82.

Collective (2003). The foundation for intelligent physical agents (FIPA). `http://www.fipa.org`.

Collective (2004). Le projet MIMOSA. `http://lil.univ-littoral.fr/Mimosa/`.

Collinot, A., Ploix, L., and Drogoul, A. (1996). Application de la méthode cassiopée à l'organisation d'une équipe de robots. In Müller, J.-P. and Quinqueton, J., editors, *JFIADSMA'96*, pages 137–152. Hermes.

Duke, R., King, P., Rose, G., and Smith, G. (1991). The Object-Z specification Language. Technical report, Software Verification Research Center, Departement of Computer Science, University of Queensland, Australia.

Durand, B. (1996). *Simulation multi-agents et épidémiologie opérationnelle*. PhD thesis, Université de Caen.

Dury, A. (2000). *Modélisation des interactions dans les systèmes multi-agents*. PhD thesis, Université Henri Poincaré.

Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., and Ossher, H. (2001a). Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38.

Elrad, T., Filman, R. E., and Bader, A. (2001b). Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32.

Ferber, J. and Gutknecht, O. (1998). A meta-model for the analysis and design of organizations in multi-agent systems. In *ICMAS'98*, pages 128–135. IEEE Computer Society.

Ferber, J. and Gutknecht, O. (1999). Operational semantics of a role-based agent architecture. In *ATAL'99*.

Ferber, J., Gutknecht, O., and Michel, F. (2004). The MadKit project (a multi-agent development kit). http://www.madkit.org.

Ferber, J. and Müller, J.-P. (1996). Influence and reaction: a model of situated multi-agent system. In *ICMAS'96*, Kyoto. AAAI Press.

Garcia, A., Silva, V., Chavez, C., , and Lucena, C. (2002). Engineering multi-agent systems with patterns and aspects. *Journal of the Brazilian Computer Society*.

Grundy, J. (2000). Multi-perspective specification, design and implementation of software components using aspects. *International Journal of Software Engineering and Knowledge Engineering*, 10(6).

Gutknecht, O. (2001). *Proposition d'un modèle organisationnel générique de systèmes multi-agents et examen de ses conséquences formelles, implémentatoires et méthodologiques*. PhD thesis, Université des Sciences et Techniques du Languedoc.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274.

Hilaire, V. (2000). *Vers une approche de spécification, de prototypage et de vérification de systèmes multi-agents*. PhD thesis, Université de Franche-Comté.

Hübner, J. F. (2003). *Um modelo de reorganização de systemas multi-agentas*. PhD thesis, Universidade de São Paulo.

Hübner, J. F., Sichman, J. S., and Boissier, O. (2002). Spécification structurelle, fonctionnelle et déontique d'organisations dans les SMA. In Mathieu, P. and Müller, J.-P., editors, *Systèmes multi-agents et systèmes complexes, JFIASMA'02*. Hermès.

IBM (2004). Hyper/J: Multi-dimensional separation of concerns for java. `http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm`.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, G., Loingtier, J., and Irwin, J. (1997). Aspect-oriented programming. In *1997 European Conference on Object Oriented Programming (ECOOP 97)*, pages 220–242. Springer.

Kniesel, G., Costanza, P., and Austermann, M. (2004). JMangler – a powerful back-end for aspect-oriented programming. In Filman, R., Elrad, T., Clarke, S., and Aksis, M., editors, *Aspect-Oriented Software Development*, chapter 9. Prentice Hall. to appear.

Lemaître, C. and Excelente, C. B. (1998). Multi-agent organization approach. In *Proceedings of the second Iberoamerican Workshop on Distributed Artificial Intelligence and Multi-Agent systems, Toledo, Spain*.

Lieberherr, K. J. (1996). *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston.

Müller, J.-P., Amiguet, M., Baez, J., and Nagy, A. (2001). La plate-forme MOCA: réification de la notion d'organisation au-dessus de madkit. In El Fallah Segrouchni, A. and Magnin, L., editors, *JFIADSMA'01*, pages 307–310.

Ossher, H. and Tarr, P. (2000). Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer.

Pinto, M., Fuentes, L., and Troya, J. M. (2002). Dynamic aspect-oriented platform. `http://www.lcc.uma.es/~pinto/AOP.htm`.

Popovici, A., Gross, T., and Alonso, G. (2002). Dynamic weaving for aspect oriented programming. In *1st International Conference on Aspect-Oriented Sortware Development (AOSD)*, Enschede.

Ricordel, P.-M. (2001). *Programmation Orientée Multi-Agents: Développement et Déploiement de Systèmes Multi-Agents Voyelles*. PhD thesis, Institut National Polytechnique de Grenoble.

Sassen, A.-M., Amorós, G., Donth, P., Geih, K., Jézéquel, J.-M., Odent, K., Plouzeau, N., and Weis, T. (2002). QCSS a methodology for the development of contract-aware components based on aspect oriented design. In *Early Aspects Workshop, AOSD 2002*, Enschede.

Schantz, R., Loyall, J., Atighetchi, M., and Pal, P. (2002). Packaging quality of service control behaviors for reuse. In *ISORC 2002, The 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing*.

Schult, W. and Polze, A. (2003). Speed vs. memory usage - an approach to deal with contrary aspects. In *The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*.

Sullivan, K., Gu, L., and Cai, Y. (2002). Non-modularity in aspect-oriented languages: integration as a crosscutting concern for AspectJ. In *The First International Conference on Aspect-oriented software development (AOSD)*, pages 19–26, Enschede, The Netherlands.

Sutton Jr., S. M. and Rouvellou, I. (2002). Modeling of software concerns in cosmos. In *1st International Conference on Aspect-oriented software development (AOSD)*, pages 127–133, Enschede, The Netherlands.

The AOSD Steering Committee (2004). aTrack project home. `http://www.aosd.net/`.

Wooldridge, M., Jennings, N. R., and Kinny, D. (2000). The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3:285–312.