MAGES: A Multiagent Testbed for Heterogeneous Agents

Thierry Bouron\textsuperscript{1,2}, Jacques Ferber\textsuperscript{2}, Fabien Samuel\textsuperscript{2}

\textsuperscript{1}ONERA, DMI.GIA, 29 Av. Division Leclerc, BP 72, 93322 Chatillon Cedex, France, tel: (1) 46 57 11 60, fax: (1) 47 46 90 25.
\textsuperscript{2}LAFORIA, Université Paris 6, T46, 4 Place Jussieu, 75252 Paris Cedex 05, France, tel: (1) 44 27 37 17, fax: (1) 44 27 62 86.
bouron@laforia.ibp.fr, ferber@laforia.ibp.fr

Abstract

This paper introduces a multiagent system, MAGES, which is used as a testbed for experimenting with different types of interactions between agents using heterogeneous architectures and behavior. This testbed is implemented as an extension of ACTALK, a generic actor language defined in SMALLTALK. It provides for: the ability to describe various agent architectures through a hierarchy of agent types, sophisticated environment models, communications between different grain-size agents, user-friendly interface and debugging tools.

1. INTRODUCTION

Distributed Artificial Intelligence (DAI) researchs need testbeds for experimenting new ideas. Several experimental platforms have already been built. They can be classified into two major types:

Type 1: testbeds for testing a specific type of agents, or a specific coordination technique (e.g. ETHER [14], ECO [8], DVMT [15], [17, 4])

Type 2: testbeds which are said to be "general" in the sense that they are used for testing various coordination techniques and to compare them. Systems such as MACE [9], MICE [5], or PANDORA-II [16] can be classified in this category.
This paper presents MAGES (Multi AGEnt System) a general multi-agent testbed system which allows for a maximum of flexibility in the definition of agent architectures and coordination techniques. It provides generic tools for building agents and their environments, plus a set of debugging and interfacing devices.

As a general testbed, the MAGES system is of the second type. Our claim is that it is more general than those listed above. Whereas they tend to provide tools and features for homogeneous agents (i.e. agents whose internal architecture, control strategy and basic behavior bear the same structure), MAGES is built along the idea that heterogeneous agents can cooperate without restraining themselves to a standard communication protocol.

The following section gives a general description of the MAGES system. Sections 3 to 7 describe its different features, and section 8 gives a brief account on the possibility of extending the system to introduce higher level structures such as group and team.

2. GENERAL DESCRIPTION

MAGES is intended to be used for research and educational purposes. Thus it was aimed to be flexible, simple, user-friendly and easy to debug.

Flexibility was the first requirement. In order to test various agent organizations, the system has to be flexible enough to be adapted to various coordination and interaction protocols. For instance it is possible in MAGES to define both behaviour based [4] or knowledge based [9] organizations. In the first case, agents react to environmental changes, whereas in the second, actions are a consequence of an interaction of intentional and explicit goals. Moreover the granularity of the agents themselves has to be flexible. For instance there are many ways to solve the simple prey-predator problem [5]. Some of them use very simple agents whereas other suppose more sophisticated architecture.

Simplicity is the second criterion. The system has to be easy to learn if we want students to use it. So all design features are built in such a way that it is easy to start with simple agents (i.e. agents whose knowledge and control is reduced to a minimum) and go up to more complex agents (e.g. agents which are based on a blackboard structure).

2.1. Hierarchy of various agent types

MAGES users can dispose of a whole set of agent architectures. Agents may differ in their knowledge and behavior, but also in their internal mechanisms : message interpretation, knowledge representation model, internal control. The system provides for an abstract behavior which can be adapted to the user's needs. Agents are organized along a type hierarchy and compose a library of "off the shelf" agents.
2.2. Environments

Environments are used to group agents within space and time, and make them react to environmental changes. "Environments represent the world or field where an agent exists" [16]. Environments can be used to implement reactive behaviors.

In MAGES, environments are implemented as specific agents whose behavior is to warn agents about events such as the entering or exiting of an agent in an environment. Because of their implementation, it is possible to describe various types of environments (grids or networks for instance), and to apply all agents development tools (i.e. debuggers, trace, inspectors) to environments.

2.3. Communications between heterogeneous agents

Simple agents use low level communications, and complex agents use frame based messages. In order for agents of various sizes and architectures to communicate, MAGES uses a special communication mechanism based on a translation process. Frame based messages sent to simple agents are converted into a mere command. On the opposite, low level communications sent to sophisticated agents are automatically transformed into message objects.

2.4. Iconic style interfaces and debugging tools

Because of the inherent concurrency of distributed organizations, it is difficult to follow the course of an agent if one does not have well design interfaces. Thus MAGES possess a whole set of trace and debugging tools based on iconic techniques.

2.5. Implementation

MAGES implementation is the result of a two years experience project. Present implementation is done in SMALLTALK [10] as an implementation of J.P. Briot's ACTALK [3], an actor language which allows for different models of actors to interact. Former realizations of MAGES were done in LISP, but SMALLTALK provided the right abstraction level for our implementation, and a great portability among different computers.

SMALLTALK has been extended to incorporate slots into objects of the language (and any SMALLTALK object can have slots with this extension), thus transforming the language into a frame based knowledge representation language, while keeping the uniformity and power of SMALLTALK.
2.6. Extensions

Due to its general organization, MAGES can easily be extended to support not built-in agent architectures and message interpreters. A general implementation protocol allows for extensions at the user level.

3. STRUCTURE OF AGENTS

3.1. General design

The MAGES system was designed to let different kinds of agents coexist within an environment. A flexible framework allows to experiment with various functional and structural models of agents, and provides the basic mechanisms for communication. Agents can be designed with the level of knowledge and reasoning capabilities that is best suited to a given organization. They can be dynamically created and destroyed, as adapted to the needs of the problem. Agents range from fine-grained to coarse-grained structures.

Fine-grained agents have a very simple reasoning process. They react to messages using a stimulus/response model, triggering any appropriate actions according to their internal state, their beliefs about other agents, and their rules. Their knowledge is limited to domain elements and, as such, they do not have the capacity to reason about the messages they receive, or otherwise develop control-oriented strategies.

At the other extreme, complex coarse-grained agents can work as individual blackboards, and have the capacity to reason both at the domain and control levels [13]. Such agents, while keeping all the capabilities of simpler agents to respond to reactive messages, can integrate messages as frame based objects, and thus have the capacity to reason about and decide when to process the messages. In this model, agents are considered as intelligent entities [9, 11], i.e as specialized problem solvers [15]. They have facilities for reasoning about both their skill domain (including the behavior of other agents) and their own behavior. The cognitive process (i.e. internal control) of each agent is explicitly represented, and agents can reason about it.

3.2. Inheritance tree

An inheritance tree of abstract agent classes was defined, providing the needed framework and protocol for creating specialized agent classes, and allows their instances to communicate (figure 1). The KernelAgent class factorizes the common behavior of all agents. It defines the basic structure and behavior of an agent, as well as its environment communication
protocol. At this level, the relevant structural information is defined in terms of acquaintances and environment. In this model, acquaintances represent the relations an agent has willingly acquired. They have a relative permanent character, as opposed to the randomly acquired perceptions conveyed through the environment. A basic protocol is defined for an agent to access its acquaintances in terms of goals, allowing it to add, remove or get a list of acquaintances registered for a given goal.

The agent internal control is described by its basic behave cycle, executed at each activation of his associated process. At the kernel level, this cycle is entirely defined in terms of message acceptance, resulting in a simple reactive behavior to external input. Instances of the ExpertAgent class augment this basic behavior by integrating an inference engine as their basic reasoning process. One further step of complexity is achieved by MessageAgents, which are able to integrate messages as part of their knowledge base. BlackboardAgents and HybridAgents are specialized MessageAgents, designed as a foundation for implementing various blackboard or hybrid models (using both frames and rules for representing knowledge). BlackboardAgents integrate KS triggering and KSAR activation, and provide the framework for successive levels of control blackboards. They can thus be used as a testbed both for BB1 style distributed blackboards, or for meta-level architecture models.

3.3. Action interpreter
The agent action interpreter is defined in terms of knowledge sources (KS). However, several levels of complexity have been provided for those KSs. Each agent class must define the class of KS that is best suited to its internal structure and control. Because all classes of KSs share the same minimal structure, it is possible to provide generic editing and tracing tools for them.

3.4. Example library

Several operational classes of agents were defined as examples of these functionalities. AgentTalkAgents and GridAgents are simple reactive agents, whose main difference lays in the implementation of their rule-triggering mechanism, done through implicit slot activation in the first case, or through explicit event propagation in the latter. Mages2Agents are complex blackboard agents, and implement a variation on the BB1 model, structuring their beliefs into domain and control layers, and providing a basic set of KSs for goal-driven control strategies.

4. ENVIRONMENTS

Environments are the physical contexts in which agents live. They define the outside worlds of agents. One goal of DAI testbeds is to represent different spatial and temporal environments in simulation applications. In the following section, several environment models are presented, compared and discussed. The MAGES environment model will be described then.

4.1. Related works

The environment idea has been mentioned in several works [9, 5, 7, 11, 16]. Table-1 lists seven features and their associated values by which environments of DAI systems can be classified. The following description is intended to supplement and explain the systems capabilities and their originalities. Different solutions have been proposed for each feature. These solutions are enumerated and explained before being compared and discussed.

4.11. An overview of environment types

Five systems are presented in Table-1. The MAGES model of environment will be described in section 4.2.

In MACE [9], the definition of an environment includes the outside world (i.e. the user and external process), the other agents and the MACE system; but the environment as it is
defined here, is represented in the kernel system by top-level demons. Agents create demons to be informed by the kernel system of some environment modifications. Global pattern-matching functions are used by the kernel system to detect these modifications.

<table>
<thead>
<tr>
<th>ENVIRONMENT</th>
<th>MACE</th>
<th>MICE</th>
<th>Playground</th>
<th>CoCo</th>
<th>PANDORA II</th>
<th>MAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>general description</td>
<td>Top level demons</td>
<td>Global structure with parameters</td>
<td>An object &quot;playfield&quot;</td>
<td>Global structure &quot;real world database&quot;</td>
<td>Passive objects with protocols</td>
<td>active object: Agent(s), with protocols</td>
</tr>
<tr>
<td>model</td>
<td>no</td>
<td>a 2D grid model</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>centralized with global perception</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes, without global perception</td>
<td>yes</td>
</tr>
<tr>
<td>distributed without global perception</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>recursive structure</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Interaction agents environment</td>
<td>Kernel system events propagated by demons</td>
<td>events propagation, predefined traps</td>
<td>events propagation, matching functions</td>
<td>events and actions</td>
<td>asynchronous messages</td>
<td>asynchronous messages &amp; events &amp; actions</td>
</tr>
<tr>
<td>passive mode</td>
<td>yes, explicit request of subscription</td>
<td>yes, explicit request of subscription</td>
<td>yes</td>
<td>no</td>
<td>yes, explicit request of subscription</td>
<td></td>
</tr>
<tr>
<td>active mode</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes, scanning cycle</td>
</tr>
<tr>
<td>Sensor Structure</td>
<td>no structure sense by the engine shell</td>
<td>a structure with field</td>
<td>observable collection</td>
<td>no</td>
<td>no</td>
<td>in the agent description</td>
</tr>
</tbody>
</table>

Table 1 - Classification of DAI testbeds environments

In MICE [5], the environment is represented by a two dimensional grid. Top-level parameters are specified to modify the environment behavior. Conflictual situations (i.e. collisions) are solved by agents; but they are detected by the system.

In Playground [7], the environment model is recursive and represented by objects: "the playfields". Global and predefined functions are used to detect the modifications in "observable collections".
In CoCo [11], the environment (i.e. "the real world") is a global structure shared by agents. When a modification occurs in the environment, it is automatically propagated in the agent database. An action model is implemented. The environment modifications are directly performed by agents through actions.

In PANDORA-II, [16] the environment is a passive object and it can detect conflictual situations. When a message is sent to an environment, it is multicast to agents in the environment. Primitives operations are provided for asserting an agent, removing one and returning the number of agents and the list of agents.

4.12. Features proposed

In table-1, the environment description is divided in two parts. The first one describes the features of the environment structure. The second one describes the interaction mechanisms between the agent and the environment.

The "model" parameter shows if a formal description of the environment model is proposed. The environment structure is said to be "global" when the environment is represented by global system parameters or by global functions. In global structure, the state of agents and their sensitivities are environmental characteristics. The environment can detect conflictual situations between its agents. The environment structure is said to be "local" when several environments live together. In a recursive environment model, an environment can interact with other environments in a more global environment.

Interactions between an agent and its environments are supported by messages, events or actions. Events and actions are monitored by demons or by predefined global functions. Actions represent only the interactions from agents to the environment. Two interaction modes are defined: the passive mode and the active mode. In the active mode, agents scan their environments at each cycle. In the passive mode, agents are informed by the environment when a modification has occurred. The environment knows the agents' senses and detects conflictual situations. It knows which agents are interested by some environment modifications. The agents sensors detect the modifications of the environment and describe the agent sensitivity; sensors are often implemented by straightforward functions and are not represented explicitly except in MICE [5].

4.13. Discussion about previous systems

Three points characterize these systems: architecture, interaction, and behavior.

Systems mostly use global structures to represent the environment [9, 5, 11]. A global structure cannot modelize heterogeneous environments, i.e. environments having several areas with different behaviors. The spatial distribution can be distorted when the outside
world is not considered as a unique entity. Object-oriented structures \[7, 16\] are better adapted to heterogeneous environments. They are also suited to homogeneous environments representation. One or more objects can be used to represent the environment and its areas. Moreover a recursive object model \[7\] allows the representation of interactions between environments in a more global environment.

Interactions between agents and their environment are supported by events except in PANDORA-II. Events are efficient but they are not flexible enough to represent interactions of multiple kinds between the agents and their environment. Events are monitored by straightforward functions and their modifications are limited.

Global structures are predefined and do not allow for the definition of new parameters to change their behavior description. The behavior of passive objects is described by primitives (e.g. enter or exit primitives). The behavior description of passive object is not autonomous. It does not allow for the redefinition of those primitives in each environment.

4.2. Environment model in MAGES

In this section we describe the capabilities of the MAGES environment model, and illustrate how it can simulate the prey-predator environment. This model is suited to homogeneous and heterogeneous environment representation. It can modelize spatial and temporal aspect of the environment. It is derived from previous environment approaches but it differs from them by four aspects: (1) environments are represented by active objects, (2) several structures with different sizes represent environments, (3) interactions are supported by events, actions and asynchronous messages, and (4) the environment model is recursive.

4.2.1. Model characteristics of MAGES environments

As a generic platform, MAGES proposes various structures for implementing environments. The user may choose one that seems appropriate for an application or he can design new ones. He can also experiment with efficiency and flexibility between different structures. Two basic organizations are presently implemented. The first one is based on a single agent which describes a global structure shared by agents. In the second one, each area of the environment is an agent. Areas have acquaintances: the agents which represent the neighboring areas of the environment. There is no global perception to detect conflictual situations, each agent having a local perception of the grid. Such a representation wastes a lot of agents; but as in our system agents of several size cooperate, very simple agents are used to represent grid squares.

Environments are represented as agents, that are autonomous and have their own behaviors. The environment behavior describes how an agent is added or removed and how
an action is asserted. (3) Events, actions and asynchronous messages are available to support the interactions between the agents and their environments. Interactions with several kinds of sensors are supported by messages which are more flexible than events. Interactions in an environment with one kind of sensor are supported by events which are more efficient than messages. Actions are described by specific objects. These objects are used in the agent's view of the environment. When these objects are modified, the modifications are directly propagated in the agent environments. (4) The environment model is recursive: interactions between environments in a more global environment can be modeled.

4.22. Environment in the prey-predator application

The environment model is illustrated by the prey-predator application which has been realized with two different environment structures and interaction mechanisms.

The environment is viewed by a two-dimensional grid (figure 2). The prey and predator agents move horizontally or vertically as they do in MICE [5]; but certain other aspects of the application have been modified. Four predators try to block a prey, but the prey starts from the top left and moves to the bottom right. His moves are indeterminate and are computed by a random function. The initial positions of the predators are also computed by a random function.

In the first implementation, each grid square is represented by an agent as in DVMT [15]. It can be freed or taken by an agent and guarantees the mutual exclusion for accessing a position in the grid. In order to avoid a collision between agents, moving agents emit an intention of displacement to the square they are aiming. The prey is detected only if it moves inside a sensitive area defined around each predator. The depth of this area is given by a number, which represents the area radius associated with each agent, and called its depthSensor. Each grid square which can be taken by the moving agent in N steps, where N is smaller than depthSensor, is scanned.

In the second implementation, the grid is represented by only one agent. The environment behavior is represented by protocols. Protocols define how an agent enters an environment, how an agent informs the environment of its sensor capability. Positions are represented by action-objects. When an agent changes the value of its position field, its position in the environment is also changed.
5. COMMUNICATIONS

5.1. Asynchronous model

The model presented here is organized around a high-level language used for communicating by sending and receiving messages. This language includes acts of communication and protocols. It can serve as an experimental testbed for experimenting with communications between heterogeneous agents.

Communications between agents always occur in an asynchronous mode. They involve both simple reactive messages, or more sophisticated message objects which form part of the agent's knowledge base.

5.2. Protocols

Protocols are defined for agents of different kinds to communicate through the use of a translation mechanism (see §.7). With this mechanism, agents can send messages to other agents without regard for the nature of addressees. This model is well-suited for experimenting with problem-solving strategies where some tasks are more efficiently accomplished by simple agents, while others need all the control capabilities of more complex agents.

Protocols are used to avoid infinite or useless dialogues that quickly saturate the structures of communication. Simple protocols begin or end a discussion. Complex protocols manage the dialogues between agents, for instance to interrupt endless communication. Moreover, the translation mechanism reduces lack of understanding by evaluating the linguistic skills of the listener. Useless communications are avoided and replaced by relevant ones, thus increasing the efficiency of the cooperation.

5.3. Resource sharing
To enforce mutual exclusion when several agents want to access the same resource agent, a protocol has been defined that involves three stages. In the first stage, the asking agent emits an *Intention*, informing the resource agent that he wants to seize it. In the second stage, the resource agent sends an *Intention Answer* to the asking agent, informing it whether it is free or not. If the answer is positive, the asking agent must then send a *Decision* to the resource. This last step allows the resource agent to resume accepting requests from other agents, should the asking agent finally decide not to seize the resource. This protocol was implemented through the use of specialized *Intention* objects, which modify their state as they go through the successive stages of the request (see §7).

### 5.4. Dynamic destruction of agents

As agents can be dynamically destroyed through the life of the system, there is no guarantee that once sent, a message will reach its address. To solve this potential problem, destroyed agents are replaced by a minimal structure called a *blackhole*, whose sole ability is to determine whether the sender of the message was an agent, and if so return the message to him, stamped with a *No More Receiver* seal. In all other cases, the blackhole will just absorb the message. The *No More Receiver* protocol thus allows agents to be informed when one of their acquaintances was destroyed.

### 6. USER INTERFACES

Debugging and tracing are very complicated tasks when agents execute in a concurrent way. Specific interfaces tailored to concurrent execution and message passing are needed, and, as a matter of fact, interfaced systems are easier to understand and to use. The interface is developed around the Model-View-Controller of Smalltalk [10]. Specific browsers have been introduced to display the agents, messages and knowledge sources classes. Generic classes of browsers have been defined; a *slot browser*, an *agent browser* and a *blackboard browser* have been realized for the frame extension and the different predefined agents.
Agents can be selected and manipulated through menus. They can be inspected. Every viewable agent has an icon and a position coordinate. Agents have multiple activities (e.g., motion and communication) and several views are needed to represent them. Displacements of agents are viewed on the grid. Specific windows are used to display the state of the agent mailbox, the sending and the reception of messages. Channels of communications are also viewed.

The user can interrupt the parallel execution. He can stop and inspect a message by intercepting with the mouse its graphic representation.

Collection of agents are represented by two kinds of windows: the AcquaintancesNetwork view, the Communications view. Channels of communication are drawn on the AcquaintancesNetwork views (figure 3 and 4). Communications view display the sending of messages and the state of an agent. Each AcquaintancesNetwork view is bound to a group of agents and displays the acquaintances links among agents included in the associated group. When an AcquaintancesNetwork view is opened, agents icons are arranged around a circle, but the user can change this disposition by choosing the move selection in the menu of an agent (figure 3). In the Communications view the number of messages in the mailbox is indicated by the size of a heap standing beside the agents icon. There are two different windows of this kind. The first one displays all communications between two agents. It is created when the user selects with the mouse the line drawn between two agents in the network acquaintances. The second one displays the communications between an agent and all its acquaintances. Such a window is opened when the user chooses the communications selection in the menu of the agent (figure 3).
The Board view is a graphical representation of the environment. It shows the grid with all the agents (figure 4). In the Board view, agents are represented by icons with a geometric shape (e.g. triangle, circle or square) and a specific color. The color of active grid squares are darker than the color of inactive grid squares (figure 2 and 4).

7. IMPLEMENTATION

7.1. Actalk

Kernel Agents are directly built as an extension of ACTALK actors [3] using the Smalltalk-80 system. In ACTALK, actors work as serialized objects [12, 1]. Each actor is animated by a process, and can be suspended and resumed. The model of communication is asynchronous, but the synchronous mode of communication defined in SMALLTALK can also be used, as long as consistency issues are kept in mind.

This language has several advantages for agent implementation: its modularity, its parallel execution and its graphical tools as described above. Moreover, ACTALK is a modular language. It is designed to implement several kinds of actor models (e.g. ABCL, Agha) and predefined to support extensions. In ACTALK objects are already active and autonomous, parallelism is already managed. The mechanism of asynchronous communication is already defined.
An ACTALK actor is basically implemented as a Behavior object encapsulated within an Actor object which acts as a serializer. The Actor part of an actor holds a mailbox in which the messages sent to the actor are accumulated. The Behavior part actually holds the actor's fields and methods, and is associated with an independent SMALLTALK process which polls the Actor part's mailbox and performs the execution of messages. The mailbox is implemented as a shared queue, thus enforcing mutual exclusion, and providing a semaphore controlled retrieval mechanism that maintains the process suspended as long as the mailbox is empty. A flexible scheduling mechanism provides an equitable repartition of process activation between the actors. Messages sent to ACTALK actors have the syntax of straightforward SMALLTALK messages, and when they are performed, the corresponding method defined at the Behavior level is executed. This is done through an overriding of the doesNotUnderstand method, which is redefined at the Actor level to enqueue not understood messages into the mailbox. This enforces asynchronous and serialized communications, as long as the actor is always accessed through its Actor part.

7.2. Message translation

This model of communication is well suited for simple reactive agents. However, defining more complex agents that can reason about their messages needs an extension of the model, which is achieved in MAGES through a translation mechanism. This mechanism provides a framework for agents of any type to receive any kind of messages. Thus, agents that inherit from MessageAgent process reactive messages as frame based objects which are inserted into the agent's knowledge base.

The doesNotUnderstand method is redefined for MessageAgents, so that reactive messages sent to them can be converted into frame based objects, which will then be inserted into the agent's knowledge base using the abstract getMessage:aMessage method. Agent classes that inherit from MessageAgent must define this method, doing whatever is appropriate to insert aMessage into their knowledge base. They also must define the messageClass method, returning the class of message their instances can understand.

Conversely, the receive:aMessage from:anAgent method, which is the basic system-level message reception method for MessageAgents, is redefined for reactive agents so as to call the doesNotUnderstand:aMessage method with a translated form of the message. This allows complex agents to send messages to other agents without regarding for the nature of the adresseses. A similar translation mechanism is applied when different kinds of complex agents communicate.
The translation itself is done at the message class level. To avoid having to define a myriad of conversion methods covering every possible combination of message classes (which would mean defining new methods for existing classes everytime a new message class is defined), the translation process uses a common interchange format defined by the class StandardMessage. This public format includes such information as the receiver and the content of the Message. The content field of a StandardMessage is implemented as a Dictionary, allowing keyword access to the values. Every message class only needs to define two methods: asStandard, at the instance level, must return the StandardMessage version of the receiver; newMessage:aStandardMessage from: sender, at the class level, must construct a new instance of the class according to the parameters. The system applies this translation process only when necessary, so agents whose message classes are compatible can directly communicate, without the loss of information and efficiency that may result from the translation.

### 7.3. Environment protocol

Interaction between agents and environments has been implemented through a generic protocol, which involves both explicit message passing, and implicit event propagation. While all the registering protocol is done through explicit asynchronous message sending, environment state changes are channeled through event propagation. An agent's representation of its state within the environment is materialized by a specialized Environment object. Changing the state of this object generates an event that informs the
agent's environment of the new state. The environment can then change its own state, and informs interested agents by asynchronous message passing. Once again, no assumption is made on the structure and semantics of the *Environment* objects, nor on the way environments will treat them. This model thus provides great flexibility for experimenting with various kinds of environments.

7.4. Intentions

Requests upon resources that involve mutual exclusion can be managed by encapsulating the resources as agents, associated with an asynchronous protocol involving specialized *Intention* objects. Such an object goes through three successive states, corresponding to the three stages of the *Intention*: request from the agent, answer from the resource, and final decision of the agent. The fields of such an object are updated with each phase transition, to reflect its current state, its status, and the agent that must receive the continuation of the request. An intention passes from the request to the answer phase through the messages `acceptFrom: anAgent` or `refuseFrom: anAgent`. It goes from the answer to the decision phase through the messages `actionFrom: anAgent` or `forgetFrom: anAgent`.

8. BUILDING HIGHER-LEVEL AGENTS

In complex experiments, agents act collectively and they may be grouped in social organizations. In this section, organizations of agents are proposed. Several organizations have been already built in previous works. They can be classified into two major types:

**Type1**: Organizations are abstract objects [9, 5]. They are represented indirectly by commitments and expectations of their members. They are derived from Weber's approach [2] and they do not really exist by themselves.

**Type2**: Organizations are represented explicitly by structures [16].

These two approaches can be modelized in MAGES. In the former, the organizations are represented in agents themselves and they do not need special structures. In the latter, special structures are needed to modelize organizations. A generic structure called a group is proposed. Groups are used for grouping agents and for defining their initial context (e.g. for initializing their acquaintances field).

Groups and environments are different. An environment represents an aspect of the physical outside world. Its behavior cannot be changed dynamically. On the other hand, a group is used to represent sociological aspects. Its behavior can be changed if there is an agreement between its members (e.g. election, consensus, leader domination).
8.1. Groups

In our model, organizations are represented as agents. The Group class factorizes the common behavior of all organizations. It is inherited by specialized and more complex structures such as the Team class. Large social organizations are presented as collections of groups of agents [6]. Features of groups are illustrated by the figure 6.

A group is defined by: (1) its group members, (2) a message interpreter, and (3) a behavior.

(1) Group members are agents or groups of agents. They are known by their groups which can send them messages. (2) Messages can be sent either to an agent or to a group. A message interpreter attached to a group contributes to its autonomy. Groups use a specific message delivery strategy to distribute messages among their own agents. Groups can broadcast messages to their members. As such they can be used to represent Maruichi’s communication groups [16]. (3) Each group has its own behavior which describes how to add or remove an agent.

8.2. Internal group organizations

The description of these organizations is beyond the scope of this paper. Some of them are just briefly presented to illustrate our discussion.
Several kind of sociological organizations may be defined in a group: hierarchic with a leader, democratic, arbitrary. Different message delivery strategies are proposed in groups to distribute messages among the “group members”. Specialized agents (i.e. managers as in the MACE system [9]) can deliver communications received by the society, or agents may look frequently at the message board of their groups.

8.3. From Classes to Groups

Groups are used also to initialize the context of an agent when it is dynamically created. In MACE these initializations (e.g. the acquaintances initialization) are realized by classes descriptions or global functions which return for example a list of addresses of acquaintances by goals. The MACE classes describe the implementation structure of agents and their social contexts (e.g. their acquaintances network). In our model classes are only used to describe the implementation structure of agents and groups are used to insert an agent in a social context. As groups are included in the multi-agent model, their behaviors may be changed and their process of initialization too.

8.4. Teams in the prey-predator experiments

In the prey-predator experiments teams are represented explicitly. The Team class inherits from the Group class. A team is also defined by (1) a general goal and an exhaustive list of conflictual goals, (2) protocols to find a team, and (3) protocols for registering in several teams.

A general goal and an exhaustive list of conflictual goals are associated to each team. The goal of a team is not always the same than the goal of its members. For example in the prey-predator example, the goal of the predators team is to catch a prey and the goal of each predator is to take one grid square next the position of this prey.

When an agent is dynamically created, it uses the contract-net protocol to be incorporated in a team. It sends a message to all teams that informs them of its availability and it chooses a team among the positive answers. When an agent subscribes to a team, it must indicate the other teams in which it is already registered. The team checks if the goals of these teams are not included in its list of conflictual goals.

9. CONCLUSION

We have presented MAGES, a general testbed for experimenting with various grain size agents, message types and protocols. The use of an object oriented language such as SMALLTALK and of an already available actor language such as ACTALK [3] has eased our
task, particularly for embedding sophisticated interface and debugging tools. However, the
goal of providing a generic testbed has been achieved. Interactions of many kinds can easily
be represented in the system, and the modularity of the architecture makes the addition of
new agent types a very simple operation.

The whole platform has been already tested on simple applications such as the prey-predator
problem and on a simulation of a Dungeon/Dragon style game where all characters, either
good or evil, are represented as agents, and where all the rooms and caves are described as
environments. The ECO [8] problem solver, which solves problems by simple agent
interaction, has already been incorporated into the MAGES kernel. We currently use the
MAGES system for research and educational purposes in the context of a post-graduate course
on distributed A.I.

Acknowledgements

We would like to thank all the students of DESS GLA who have participated to this
implementation of MAGES and those who have implemented the previous versions of
MAGES. Special thanks to Jean Pierre Briot and Loïc Lescaudron for their beautiful ACTALK
implementation and for all their friendly help.

REFERENCES


Environment*, ECOOP’89, LITP University Paris VI, 1989


of All Ages*, OOPSLA 89 pp. 123-137.


