Towards a multi-agent model for the office information system: a Prolog based approach[1]

AUTHORS: Bruno Dillenseger (phone +33 31 75 91 39, email dillenseger@sept.fr, fax +33 31 75 06 31) and François Bourdon (phone +33 31 75 91 19, email bourdon@sept.fr), SEPT SCE/ARC, 42 rue des Coutures, BP 6243, F-14066 Caen cedex, France.

KEYWORDS: object-oriented distributed system, Prolog, COOL, Computer Supported Cooperative Work, Multi-Agent platform, service negotiation, cooperation, client-server seek.

ABSTRACT: The background of this article consists in a distributed object oriented approach for the development of logically and physically distributed applications in the field of Computer Supported Cooperative Work. The use of an object oriented distributed system (COOL) is a first step towards the implementation of solutions to these naturally distributed problems. But this support is insufficient to directly implement high-level interactions and behaviour: the entities have to adjust themselves to a dynamic system where services and servers appear, evolve, and disappear.

This article introduces both a model and a tool which are designed to enhance the entities cooperation ability and autonomous adaptability. The model springs from investigations in the field of Multi-Agent Systems and consists in a uniform representation of every resource of the system (hardware, software, user) by agents.

The implementation tool and associated goodies are described in some detail. They are based on an enhanced Prolog interpreter essentially integrating the COOL features: various communication modes, active objects, migrating objects... Our Prolog-based approach is discussed and illustrated by the implementation of a distributed application and some generic multi-agent system structures and protocols. We also show how several artificial intelligence techniques are involved in the solutions we propose.

1. Introduction

1.1 Autonomy and integration of the information system

The unpredictable evolution of the office [13], in a geographically distributed environment, makes it necessary for the information system to adapt itself to the resulting organizational and operational changes. Thus, autonomy increasingly becomes an essential property for the overall information system and its components.

Moreover, as users are confronted with a profusion of autonomous and evolving services, they need a guide to find the most relevant services for their current tasks. This help requires interactions between the autonomous computer entities, and should rely on a standard interface language. This common language is supposed to create a basic semantic universe that should be shared by any sender and any receiver of a message. This *integrator principle* concept was introduced by Jean Erceau and Michel Barat in [2].

Beyond a functional representation of the offered and/or requested services, the server quest has to feature negotiation possibilities to make the requests suitable for the “market” tender, learning (memory) capabilities to avoid systematic and useless quests, and sophisticated communications that allow manual (human) actions when negotiations become hard.

1. in collaboration with LAIAC, Université de Caen.

1.2 Towards tools and techniques

The combination of the distributed systems technology with an object-oriented approach makes it possible to conceive applications as collections of independent and interacting entities. This approach is quite relevant to many families of problems that are logically and physically distributed by nature. The SEPT[2] is confronted with such issues in the field of Computer Supported Cooperative Work: automatic circulation of electronic documents (workflow) in CIDRE [4], cooperative editing (groupware)...

As a result, we have adopted a Chorus micro-kernel based distributed system and its associated object-oriented layer: COOL [8]. Several distributed applications have been written in C++ on top of this layer and have shown the great attractiveness of COOL in terms of distribution, transparency, communication, encapsulation, and coarse-grain parallelism. But this layer by itself is not adequate to directly support the high level cooperation and behaviour which is required to provide the entities with sufficient autonomy and adaptability.

Our response to the need of an integrator principle is double. First of all, we will introduce a multi-agent model for all the resources of the information system and their interactions. Then, we will describe a COOL and Prolog based tool set which is likely to support a high level of shared language and semantics. The implementation of our model is described through a distributed meeting room reservation service.

2. A multi-agent model

2.1 The agent-oriented approach

Our notion of what an agent is springs from the field of Distributed Artificial Intelligence and, more precisely, is inspired by research on an agent-oriented method [6]. We are interested in the conceptual aspect of the agents, since the problems we face are inherently distributed. But, we are also interested in the implementational point of view, as we benefit from the support of an object-oriented distributed system. Moreover, agents are increasingly being regarded as a software engineering concept. *Agent-oriented programming* is presented in [11] as a specialization of object-oriented programming (agent *mental state* concept which takes into account the notions of belief, decision, obligation, basic types of communication coming from the speech-act theory).

Among the abundance of DAI-theoretical issues such as cooperation matters [12], distributed plan generation and execution [10], knowledge and belief representation [9], our concern is to find an answer to this fundamental question: when an agent has to complete a task it can not solve on its own, how is it going to find the most appropriate cooperating agent and then, how is it going to negotiate a service?

This is an essential issue for any object-oriented distributed system, which is a particular multi-agent system: when a client object intends to run a method of a remote server object among many other available ones, it has to use its knowledge and some special system structures. The function of these structures and associated protocols is to help the link to be dynamically made between the servers and the clients. The ODP[3] traders [1], for instance, are used by the server objects to declare the exported services, and by the client objects to seek the relevant servers.

2. “Service d’Etudes communes de La Poste et de France Télécom”, a joint research lab for La Poste and France Télécom.
3. Open Distributed Processing

2.2 Making a uniform system

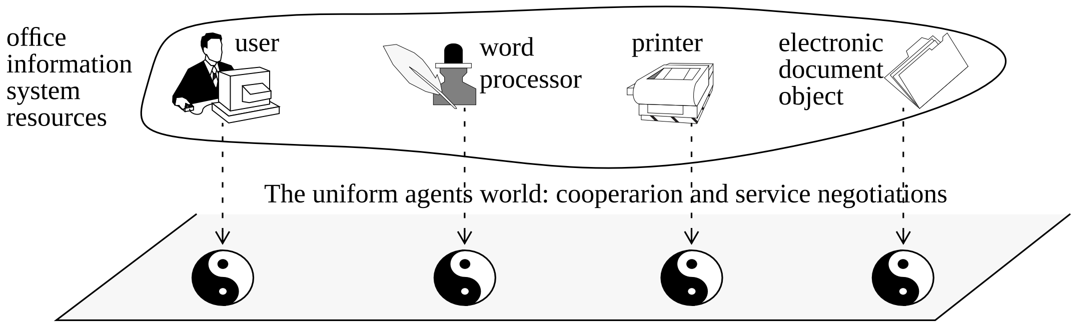
The first step of our "integrator principle" relies on a homogeneous vision of the office information system. The network links a set of machines (*sites*) together, each of them holding a few *resources*. These resources are likely to appear, disappear or evolve: application software (e.g. databases, editors), migrating application entities (e.g. circulating electronic documents), peripheral hardware (e.g. printers), connected users. Each resource owns a particular set of available *services* which are associated to some special *attributes*.

The *structural attributes* are static, or weakly dynamic in case of upgrades. They are typical of a particular service implementation. For instance, they may consist in the printer brand, the printing process and speed. The *conjunctural attributes* are typically dynamic because they depend on the current context. In the printer case, they could hold information about the size of the printing queue, the number of remaining paper sheets.

These resources may need some other resources' services. When they do so, they act like *clients* looking for *servers*. As a result, they are confronted with the basic problem: they have to find the right service and therefore the suitable server. This implies we must cope with two issues:

(1) to be able to contact the resources even though the system is dynamic;
(2) once the contact is made, to be able to negotiate as precisely as possible.

First, both points (1) and (2) require a great deal of interoperability. We propose to support it with an overall integration of the system resources into a uniform representation. Thus, each resource is associated with an agent whose role it is to represent it for the other resources. This principle results in the creation of a uniform layer by projecting the system entities (cf. figure 1.). This layer is the place where the servers queries and the service


office information system resources
user
word processor
printer
electronic document object
The uniform agents world: cooperarion and service negotiations
figure 1: making a uniform system representation

negotiations occur. The agents cooperate according to their skills, i.e the services they offer. The service invocation is the basic cooperation abstraction.

An agent behaves like a unique controller and representative of a resource and its associated services. Its behaviour consists in permanently listening to the requests, both internal (i.e. from the resource it represents) and external (i.e. from the other resources). Moreover, it may follow a private activity consisting, for instance, in supervising some tasks it is responsible for, or consulting the system (i.e. other agents) and organising its knowledge for learning concerns.

Secondly, point (1) introduces the need for special structures and associated protocols which aim at providing servers with clients and vice versa. The first idea is the services directory and the implementation of trader agents, but other structures could be imagined (e.g. by analogy with classified advertisements, yellow pages). Any agent, at creation time, has to know at least one of these structures, as well as its associated usage protocols. Finally, the agents have to share some semantics and a negotiation language to achieve point (2).

3. A multi-agent layer

3.1 An object-oriented distributed system: COOL

The Chorus Object Oriented Layer was specified by SEPT in order to allow the actual distribution of applications such as the Intelligent Circulation of Distributed Folders. COOL v1 [8] mostly embeds the communication features of the Chorus distributed system micro-kernel into an object oriented layer. It is available through a C++ `COOL` class.

Any `COOL` object owns blocking (call/reply) and non-blocking (send/receive) communication methods, and communication groups management methods. It is able to migrate from one site to another through the network, to synchronously call another `COOL` object's method (in the same address space with COOL v1, or in a transparent distributed way with COOL v2 which is being released). It may own a mail-box and may be either passive or execute its own activity. A semaphore object class is also available to cope with concurrent access issues. Finally, the communication system is completed with a distributed symbolic naming service which allows the objects to register their mail-box address using chosen names.

3.2 Introduction of a high level language: Prolog

The use of C++ is justified by its object-orientedness and its efficiency inherited from C, the UNIX systems' traditional language. But it appears to be limited and not to meet some of our needs: knowledge representation, high level negotiation language, "intelligent" and adaptive behaviours... The steps we took towards Distributed Artificial Intelligence through our multi-agent model, called our attention to the traditional languages of this field. As we could not do without the COOL facilities, we decided to integrate one of these languages as a COOL/C++ object.

Prolog was chosen for several reasons. First of all, we need a high-level interpreted language to provide our agents with sufficient evolution capabilities. Thanks to their uniform data and program representation, Prolog programs can easily exchange knowledge as well as know-how, by adding communication predicates. Within our model, this facility allows the new procedures or old procedures changes to be propagated to the agents. This way, agents can dynamically upgrade themselves, without interrupting their functioning. Besides, this technique is also suitable in a heterogeneous environment.

Moreover, Prolog gives a common communication language base which is shared by every agent. Prolog is a very efficient support for the definition and interpretation of new languages that may be designed for our specific needs (e.g. constraints language definition and interpretation for service negotiations). At last, our agents benefit from the unification engine in order to implement some more or less clever reasoning.

A multi-agent model for Human-Computer Cooperative Work has already been carried out in the IMAGINE [7] Esprit project. It has resulted in the development of a parallel Prolog(4) based multi-agent environment and tool-box. The approach is very near ours: mixing both the Computer Supported Cooperative Work research, which aims at assisting the cooperation between human agents by the use of computer resources, and the DAI anthropomorphic cooperation models, so as to make it easier for the users and the applications to interact, cooperate, by integrating them into a unique model. But this work does not benefit from the support of an object-oriented distributed system such as COOL.

4. IC Prolog II

3.3 Prolog Upgrade for Multi-Agent systems

3.3.1 A Prolog interpreter in a COOL object

PUMA results from the integration of an Edinburgh-type Prolog interpreter into a COOL object. It is important to note that our main concern was not to choose a state of the art Prolog implementation but just to prove the feasibility and the attractiveness of such an approach. Neither were we interested in the communicating Prologs as we wanted instead to take advantage of our object-oriented communication layer. Nevertheless, a further step is needed to determine the most suitable logic programming language. The experimentation on our current system will make this study easier.

The first point of view is to consider PUMA as the integration of a Prolog interpreter in a C++ object, in a distributed environment. The basic **`puma`** class holds the complete access and control interface to the Prolog kernel but does not define any activity. Its constructor method takes two arguments: the symbolic name of the object and the initial Prolog file name. This class is typically bound to be used by operational classes through inheritance. It is also likely to be used directly to create a Prolog server object, either reachable through method invocation or by running its creation reflex (if it is defined in its initial file).

Two classes are derived from the `puma` class. The `interp` class defines an interactive interpreter activity and its instances behave exactly like Prolog interpreters that offer embedded COOL features through new predicates. This class is quite useful for testing, observing, and perfecting other `puma` objects and, to a certain extent, even non-`puma` `COOL` objects. The **`agent`** class defines a generic agent activity which consists in repeated and continuous calls to a particular predicate representing the elementary activity step. Typically, it consists in reading and processing one message -if any- from the mail-box. There may be also an individual task to run but each step should be as fast as possible(5) in order to keep the agent available enough and to avoid a mail-box overflow.

At last, any `COOL` derived class can be automatically and almost transparently linked to a private PUMA agent by inheriting from the **`MASobject`** class. This agent is available for synchronous/asynchronous invocations and automatically follows the object it is associated with when it migrates. Some creation, migration and destruction Prolog reflexes can be programmed for the agent. The use of this class is related to the projection principle of the system resources in a homogenous agents universe.

3.3.2 The extended Prolog dialect

PUMA may be also considered as the integration of COOL in a Prolog language, since almost every COOL feature is embedded in new system predicates (see [14] for further details). Most of them are dedicated to communication:

- symbolic naming service (myname/1, vanish/0, appear/1, available/1),
- asynchronous communication with group features (send/2, receive/1, functmode/2, broadcast/2),
- group management (addtogroup/1, mygroups/1, quitgroup/1),
- migration (migrate/1, follow/0),
- synchronous communication (phone/1, ringup/1, ringup/2, accept/0, refuse/0, hangon/1, hangup/0),
- dynamic COOL object - typically `puma` object - creation (create/3),
- C++ level synchronous invocation (interrupt/1, ireturn/1).

5. an auxiliary agent may be dynamically created to manage a long lasting task.

A specific advantage of the Prolog approach appears when migrating active objects. As a matter of fact, activity migration is a complex issue which is not solved by COOL: the COOL object activity is restarted from zero after each migration. But, with a `puma` object, migration is transparent for any Prolog program that calls the `migrate/1` predicate, thanks to the Prolog state save/restore feature.

3.3.3 Cooperation between the C++ and Prolog levels

This double faced integration results in the possibility for the programmer to conceive compound objects whose activity is a C++ program making Prolog calls from time to time, or a Prolog program calling some C++ procedures, or the combination of both. These invocations may be either synchronous or asynchronous thanks to the COOL features, the `PUMA` class interface and the new predicates. From a general point of view, these facilities are very interesting simply because these languages suit different tasks.

4. Implementing the model

4.1 A multi-agent meeting room reservation system

We will now describe the implementation of our model in an actual and simple case, using the tools we have just presented. It consists in a multi-agent reservation system which uses two main agent types: the *room agents* and the *user agents*.

Each meeting room is exclusively represented by a private agent which maintains some information about its structural attributes (equipment, size, place...) and conjunctural attributes (reservations planning, scheduled repairs...). The agent is the complete interface to the typical meeting room services: reservation and cancellation.

When a user wants to reserve a meeting room, the corresponding service can be invoked via its user agent which manages to contact the right servers (i.e. the room agents). Hence, this agent is not only in charge of representing the user in the system, but it is also the entry point towards the whole available services set.

4.1.1 The agents

Each user and room agent is a *permanent* agent as it represents a resource. It has to hold a minimal user interface to allow administrators to move it onto another site, or to kill it. The agent + control interface compound entity is implemented with the `interface` class which inherits from the `MASobject` class. This results in the definition of two parallel activities: the agent activity - typically a Prolog program - and the interface activity, typically waiting for an event from the user. The control interface also allows users to invoke the underlying agent. Each agent grants access to a specific set of system wide or local services. A service may be either *internal* or *external*, depending on whether the agent can execute it by itself or by invoking another agent.

Each permanent agent loads a particular Prolog file which contains the definition of two internal services: the agent attributes viewing service and editing service. Room agents, by loading a specific Prolog file, gather new internal services such as planning consulting, room reservation and cancellation. As far as user agents are concerned, they load another specific file which defines an internal memento service, but also two external services: room reservation and reservation cancellation. Thanks to the Prolog declarative representation, agents are able to modify their services or learn some new services dynamically and then to propose them to the interface.

Some *temporary* agents are dynamically created in order to comply with some special tasks, when a service takes a long time or needs migration to be executed. This kind of agent need not have any user interface and it is simply made of an `agent` class object and a specific sequence of Prolog files. As it is specialized in the domain of the task it must carry out, there are many derived types of temporary agents.

For instance, the memento consulting service is an internal service for any user agent but it is run by a dynamically created auxiliary agent. Otherwise, since this service is user interactive, it would disturb the agent's functioning. When running the service, the temporary agent gets the information it needs by directly accessing its creator's Prolog kernel, thanks to the synchronous invocation predicates. This technique allows the parallel execution of many local internal services, without replicating information. It prevents the system from redundancy and the resulting memory waste and consistency problems.

4.1.2 The negotiation language

Neither the uniform representation of the system resources with client/server agents, nor the availability of a common language (Prolog) are enough to create a real *integrator principle*. It is necessary to introduce system-wide shared semantics; otherwise, any communication and then any cooperation attempt is illusory.

Consequently, as the service abstraction is the key element of our cooperation conception, being able to precisely describe the expected service is essential for the agents. To face this issue, agents share semantics for a given set of symbols to represent particular services and attributes. They know a common constraints expression language to specify constraints on these attributes. They also know the functioning of some dedicated structures and associated protocols which aim at providing servers with clients and clients with servers.

Our constraints language allows us to specify whether or not an attribute should belong to a set or an interval. The Prolog generic constraints solving engine we implemented can be enriched by each agent to solve special cases by introducing default constraints and new domain dependant rules.

Moreover, the language takes into account a satisfaction criterion which enriches the "or" notion with ordered preferences. Satisfaction is computed from a Prolog procedure and the computed values of some attributes. Thus, this language is quite expressive and allows the formulation of requests as complex as "reserve a room for 10 persons, with an overhead projector, from 8:30 AM to 11:00 AM, preferably on the first Monday of January 95, otherwise on another Monday of January 95".

4.2 A cooperation protocol example

In the system we are presenting, cooperation is based on a special group structure which relies on the Chorus group communication features. We will describe two special points: the structure management and the associated server quest protocol. Both are designed to reduce the number of messages in order to avoid the drawbacks of some protocols such as the contract net protocol [12]: communication network overload, agents' mail-boxes overflow, agents message processing overhead.

4.2.1 The artisan group structure management

The structure we implemented consists in making the agents from a given domain maintain a kind of an artisan association which aims at satisfying the clients instead of competing. As soon as a room agent is created in the system, it informs the room group of its existence by broadcasting a declaration message. Then, each member of the group returns a message so as to state their adherence to the group. Since each declaration message holds the structural attributes (surface, equipment, place...) of the sender agent, each room agent permanently knows the other room agents and their associated structural attributes. Conjunctural attributes (e.g. reservations planning) are not transmitted to avoid misusing the group broadcast facility. Nevertheless, the good functioning of this structure management relies on two assumptions:

(1) the artisan group is slowly dynamic, i.e. the creation, removal of a member and the evolution of its structural attributes are uncommon phenomena;

(2) the number of members is limited.

When there are n members in a group, the declaration of the (n+1)th member causes 2n messages to be sent. n of these messages are sent within the same broadcast. Even if the broadcast cost is difficult to evaluate since it depends on the network type, there are n individually sent messages however, which come in the same mail-box. If assumption (2) is not met, a mail-box overflow may occur. Generally speaking, as we allow data replication, both assumptions insure that maintaining a satisfactory level of consistency is not too costly.

4.2.2 Invoking the artisan group structure

As the members of an artisan group know each other, it is no use broadcasting a service request on the group. When a user agent wants to reserve a meeting room, he just tells it to one room agent. As the client is not supposed to know any server in advance, it invokes the room communication group using the Chorus group *functional* mode. This mode consists in sending a unique message to a random member of the destination group.

Since the user agent is not an expert in the room reservation domain, it is not able to put a complete service request into words. Therefore, the room agent replies by creating a temporary agent, a kind of a *commercial agent*, whose mission is to manage with the client's request. The commercial agent copies the room agents list and their associated structural attributes from its creator by directly accessing its Prolog kernel. Then, it migrates to the client site and interactively builds a complete service request with the user. It asks him/her for the main constraints but also directly gathers some information from the user agent's Prolog database. Thus, the physical and logical distribution of the problem is respected: the complete request is built by an expert agent coming from a remote site.

Then, the commercial agent lists the relevant rooms according to their structural attributes, and sorts the list according to the satisfaction criterion(6). Finally, the commercial agent successively contacts the potential servers as long as it receives negative answers for unavailability matters. When a server accepts the request, the commercial agent gives the service contract to the client agent which informs the user and records the information.

As a result, our server quest protocol is clearly optimized: communications are spread over time with few messages and no broadcast. Although the commercial agent migration is rather costly, it has to be underlined that it takes place only once. Moreover, it prevents the negotiation phase from generating many messages that would have been necessary for a remote interaction between the user and the commercial agent. At last, the fact that the commercial and the client agents are in the same address space allows fast and communication system load independent interactions.

4.2.3 Support for other structures

The artisan group structure is dedicated to small groups of servers. But a client may look for a server which doesn't necessarily belong to a group but just matches several constraints.

6. The satisfaction criterion defaults to a best fit between the needs of the client and the room attributes.

For instance, a mail service should make it possible to reach any member of an office whose car is red, in order to tell him that the lights are on, but without disturbing everybody. The user agents are suitable to hold such information as structural attributes, but the artisan group structure does not fit a system-wide search.

To cope with this issue, we propose the creation of a directory agents artisan group which implement the "seek agent" service. Each of these servers registers the permanent agents with their structural attributes, depending on whether these attributes meet a specific set of constraints (the *filter*). A directory agent may split if its database becomes too big, and transform into two more specialised directories, according to a taxonomy process. A directory agent may also replicate and move (see a model for automatic positioning in [3]), by analysing the main sources of the requests. Directory agents may also merge if they are small and rarely invoked. Consequently, we can imagine introducing a unique directory agent with an empty filter and observe the logical (filters specialization) and physical (positions in the system) emergence of the directory agents group.

4.3 Practical concerns

The room reservation system and the artisan group structure was run on three Chorus micro-kernel based PC(7) and showed the great efficiency of the group protocol and the communication system which is integrated in the micro-kernel. It was also tested on a stronger hardware configuration (SPARCstation 10) which offers more power but introduces a serious communication bottleneck, as it still runs a Chorus simulator on top of UNIX instead of a native Chorus micro-kernel based system.

The Prolog kernel is compiled as a library and is included in every address space. As a result, the sizes of the `agent` and `interp` classes (cf. 3.3.1) do not exceed 17K. For each object, dynamic memory allocation essentially springs from the Prolog database which fills about 100K. Consequently, a PUMA object creation or migration is rather costly, particularly for memory concerns. On the other hand, it has to be outlined that the Prolog activity goes on through migrations.

5. Conclusion

Within the Computer Supported Cooperative Work domain, this article presented PUMA, a Prolog based layer which benefits from the support of the COOL object-oriented distributed system. We introduced a multi-agent model for the office information systems and we illustrated it with a practical application. We wanted to show the feasibility and the power of a collaboration between a compiled language (low-level efficiency, control) and an interpreted language (dynamic aspects, expressiveness).

The main advantages we found from Prolog are the uniform representation of knowledge and know-how, the ability for a program to evolve, the support for defining languages, and the unification process to implement an evolutionary constraints solving engine. Some reasoning needs were presented in terms of taxonomy, specialization/generalization, automatic positioning. Learning capabilities also should be introduced in some agents to foresee the behaviour of the resources they represent.

As a matter of fact, PUMA is a first step towards the introduction of Distributed Artificial Intelligence in the office information systems, and there is still a lot of work to be carried out: development of other applications (directory agents group, intelligent mail service,

7. These computers are 66 MHz 486 PC, with 20 Mbytes RAM, running CHORUS/Fusion for SCO UNIX.

evolution of CIDRE), evolution of COOL, choice of the most suitable logic programming language, introduction of more “intelligent” features... We will be more and more involved in studies in these areas.

ACKNOWLEDGMENTS

This work was led in collaboration with Patrice Enjalbert from LAIAC - Université de Caen.

Special dedication to Christophe “tof” Trompette, the originator of the first PUMA interpreter.

REFERENCES

[1] *Basic Reference Model of Open Distributed Processing: non-normative (descriptive) specification of trader.* ISO/IEC JTC 1/SC21 WG7 N, Standards Australia 1993

[2] Michel Barat, Jean Erceau. *Utilité et utilisation d’un principe intégrateur dans un outil de conception de systèmes complexes multi-experts.* 2ème congrès européen de systémique, vol. III pp 860-869. Prague, october 1993.

[3] François Bourdon. *The automatic positioning of objects in COOL.* IEEE/CS, 14th International Conference on Distributed Computing Systems, Poznan, june 1994.

[4] Jean-Marc Deshayes, Vadim Abrossimov, Rodger Lea. *The CIDRE distributed object system based on Chorus.* Proc. of TOOLS’89.

[5] Marc Desreumaux. *Pourquoi les entreprises ont-elles besoin de systèmes d’information flexibles ?* Tome 7, 1er congrès biennal AFCET, Versailles, june 1993.

[6] Jacques Ferber. *BRIC : essai de mise en perspective d’une méthodologie multi-agent.* Journée d’étude AFCET “méthodes orientées agents”, Paris, september 14th, 1994

[7] Hans Haugeneder. *IMAGINE Final Project Report.* ESPRIT project 5362, IMAGINE Consortium

[8] Rodger Lea, Christian Jacquemot, Eric Pillevesse. *COOL: system support for distributed programming.* Communications of the ACM, Vol.36, No.9, 1993.

[9] Claire Lefèvre, Claire Beyssade. *Système multi-agents et modalités épistémiques.* Premières journées francophones IAD & SMA, Toulouse 1993

[10] V. Lesser, D. Corkhill. *Distributed problem solving.* Encyclopedia of Artificial Intelligence, Vol. 2 (1987), 245-251

[11] Yoav Shoham. *Agent-oriented programming.* Artificial Intelligence 60 (1993), Elsevier, 51-92

[12] R.G. Smith. *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver.* IEEE transactions on computers Vol. C-29, No. 12 (december 1980), 1104-1113

[13] R.A. Thiétart. *Ordre et Chaos dans les Organisations.* Journée d’étude AFCET “Les Systèmes d’Information, Autonomie et Chaos”, Paris, november 24th, 1993.

[14] Christophe Trompette. *Etude de modèles de négociation dans un univers multi-agent.* Rapport de stage de DEA, Université de Caen, september 92.