Negotiation and Conflict Resolution within a Community of Cooperative Agents

Eugénio Oliveira, Fernando Mouta & Ana Paula Rocha

Faculdade de Engenharia da Universidade do Porto
Rua dos Bragas, 4099 Porto Codex, Portugal
{eco, mouta, arocha}@fe.up.pt

Abstract

UPShell, a tool for building up coarse grain, semi-autonomous cooperating agents (which are expert systems) is here presented.

Negotiation and conflict resolution protocols have been integrated into the agents whose architecture is also presented. Moreover, several basic functionalities dealing with task scheduling as well as cooperation policies, are briefly specified.

A realistic scenario, on a sophisticated electricity distribution network management application is also presented to illustrate the referred concepts and tool features

Key words:

Cooperation, Multi-Agents, Conflict Resolution, Negotiation

1. Introduction

An environment to generate and exploit a community of cooperative knowledge based systems, UPShell (University of Porto Shell), has been developed at our group, in the framework of an Esprit Project (Archon)[1].

The UPShell [1] may be used either by domain experts in order to generate specific Intelligent Systems (IS) or, in a consultation mode, to solve complex applications. In the first case, different ISs may be generated and, if needed, the Shell can be instructed to build up a cooperative system consisting of several of these expert systems. In order to make this possible, special modules including additional knowledge have to be automatically derived for the use of the cooperative community. By adding this new knowledge as well as new functionalities, ISs are transformed into cooperative agents. In the consultation mode, the shell provides the means for user interaction with either a separated IS or a set of agents pursuing an overall goal.

UPShell generates different Agents which run as a set of separated Unix processes that can be distributed by different machines. UPShell is implemented in Prolog under Unix and also encompasses capabilities to use Xwindows and Unix inter-process communication facilities.

This paper is concerned with the enhancement of cooperative capabilities of UPShell (for a description of its basic functionalities see [1]). Methodologies for multi-agent negotiation as well as for conflict resolution are here described.

While in section 2, agents architecture is presented, section 3 enunciates and describes important capabilities we are using for the sake of sophisticated cooperation. A tentative classification of all different kinds of cooperative and conflict cases we may find on coarse grain semi-autonomous multi-agent applications is also presented in section 4. In section 5 the basic functionalities on which cooperative capabilities rely, are specified. The paper ends with a realistic application example and conclusions.

2. An architecture for cooperative agents

In this section we describe, in broad terms, the main architecture of each agent.

The basic architecture of the agents [2] consists of two conceptually separated subsystems:

[1] Esprit II project ARCHON, ARchitecture for Cooperative Heterogeneous ON-line systems (P-2256), aims to construct a general purpose multi-agent system architecture for industrial applications.

- An upper layer for cooperation and coordination of agent activity (Cooperation Layer - CL);
- The underlying Intelligent System that has the domain specific problem solver capabilities (IS).

It is the responsibility of the cooperation layer to coordinate inter and intra agent activities. This layer is further divided into:

- a communication module for information exchange with other agents;
- a Decision Making Module (DMM) for:
  - choosing cooperation policies according to the agent workload and overall community situation, and
  - controlling the communication module;
- a control module (monitor) for scheduling IS problem solving activities.

For these purposes (namely to make decisions after situation assessment) the DMM uses:

- a Model of Acquaintance Agents (AAM) where relevant information about other agents knowledge and workload situation is stored;
- a Self Model (SM) with information about several aspects of the underlying IS skills and workload;
- sets of rules including knowledge on cooperation and local control;
- an internal Data-BlackBoard/Goal-BlackBoard (DBB/GBB).

The IS structure is similar to a rule+frame based expert system. Some sophistications were made in the rule interpreter in order to be able to attend the upper layer control directives and maintain several tasks simultaneously by means of indexing scheme for the task environments.

UPShell may be used either by domain experts in order to generate specific intelligent systems, or by other users in a consultation mode. In the first case the Shell automatically build up the self model (a model of the IS skills). In consultation mode the Shell can be instructed to build up a cooperative system out of those different ISs. In order to do this, specific knowledge (and code) have to be automatically produced concerning the skills and capabilities of each IS which may be important for the other ones in the same community (the model of acquaintances). In this case, ISs are transformed into Cooperative Intelligent Systems (CISs) or agents through the addition of an upper cooperative layer on their top. In the consultation mode the Shell provides the means to give either to a separated IS or to a set of CISs an overall goal to be pursued (if necessary under time constrains).

3. Capabilities of cooperative agents

3.1 Local task scheduling

The monitor is the cooperation layer's module that controls and monitors the underlying IS. The aim of local task scheduling is to guarantee the accomplishment of every task in the agenda according to its deadline. Therefore, priorities are computed for each task taking into account dynamic information reflecting agents computation load as well as waiting times for task inputs. However, requests need to be formulated with specific temporal deadlines attached, and not priorities, letting to the receiving agent the job of computing the more appropriate priority value (for task scheduling purposes) according to its own commitments.

If, for example, an urgent request arrives or if a requested task is approaching the deadline without being executed, the scheduling process of tasks in the agenda should be redone in order to accomplish this specific task within the time limits.

The scheduling will take into account the capabilities of the underlying IS among the following:

Task switching capabilities:

The monitor is able to control IS task execution in three different ways: task swapping, task redoing and task restarting.

The IS, during the execution of a task is able to suspend the current execution and to switch to execute a new task. When the new task has been executed, the IS is able to continue the previous execution from the breakpoint.

To do task swapping it is necessary to suspend the IS's execution of a task so that it can be restarted later.

To execute a task, the monitor accesses the goal-plan in the self model, to find the list of Basic Processing Elements (BPEs) that form the plan of that task. Then, it sends to IS, all the needed input to that task as well as the first BPE. When IS finishes the BPE, it will send a message "context(Task, BPE)" to the cooperation layer with the answer to this BPE. If the monitor is instructed to continue the execution of this task, the following BPE of this task will be sent to the IS.

It may happen however that DMM has instructed the monitor to redo, restart or swap to a specific task and that precise moment (between BPEs) is the right one

to control the underlying IS appropriately.

Multi-task execution:

The IS is able to do several tasks simultaneously, and the monitor is capable of monitoring the execution of several tasks at the same time.

3.2 Detecting the need for cooperation

When an Agent detects the need for help (unable to execute a task), it will check in its acquaintance models if there are agents able to do it. If there are one or more, the first step is to start a negotiation process. Let us call "organizer" the agent that initiates the negotiation and "respondents" the agents that receive the request for negotiation. The goal of negotiation is to have one or more respondents committed to execute a task to help the organizer.

The two main cooperation forms for distributed problem solving are task-sharing and result-sharing [3]. Task-sharing is a kind of cooperation in which individual problem solving nodes assist each other by sharing the computational load for the execution of sub-tasks of the overall problem. Result-sharing is a mechanism of cooperation through which results produced by a node are sent to other nodes that may benefit from these results.

Next, we list the situations when an agent decides to initiate some form of cooperation:

• Task-sharing:

As organizer:

When decomposing a task, if some sub-tasks can not be dealt with locally, or if some inputs to this task can not be gathered locally.

Also, the IS of an Agent, during the execution of a task may need some data that it does not own. Then it will ask for it, through the monitor, to the cooperation layer. If the data is not available here, it has to be requested to another agent.

As respondent:

An agent receives a "task-announcement" message and tries to answer according to one of several different protocols.

• Result-sharing:

As organizer:

The cooperation layer, after receiving a result produced by the IS, consults the AAM to see wether the other agents may benefit from these results. If this is the case, results are sent to them.

As respondent:

An agent receives incoming data volunteered by any agent (not answer to a previous query or protocol).

If the received data is part of the input to an executing or executed task and if it is more reliable than the previous one, then the corresponding task is restarted or redone.

If the received data, eventually together with data already existing, completes all the necessary input to a certain task, the state of this task is changed to "ready" in order to be executed.

3.3 Negotiation

Cooperation, mainly task-sharing, may lead to the need of negotiation among several agents.

Once an agent wants to formulate a request, in order to get an answer from other agents, it needs to establish a protocol with the possible respondents for that specific request. This protocol will permit the exchange of information between the agent interested in the answer (organizer) and the possible respondents. During this protocol, the organizer informs other agents about the constraints associated to its request (mainly a deadline) and the potential contracted agents will respond informing about the expected time needed for having that task executed and the expected quality of the answer (bid message) [3].

It is up to the organizer to evaluate the bid messages and to contract one (or possibly more) respondent.

3.4 Conflict resolution

Conflicts are always possible to happen in a cooperative multi-Agent community [4]. Conflicts may appear either during task sharing or result sharing operations, and may also be classified either as positive or negative. Therefore we may recognize :

• Positive conflict in task sharing:

Whenever several agents are able to execute a task requested by the same organizer. This situation has to be tackled down by means of negotiation functionalities.

• Negative conflict in task sharing:

Whenever there are no agents willing (or able) to execute a task, requested by the organizer, in due time. This situation requires a new plan to be issued by the organizer's monitor in order to find a new and feasible plan for that specific task execution.

• Positive conflict in result sharing:

Whenever several agents, which are trying to execute the same task, produce different but complementary

results for the same request (or similar results with different credibilities associated).

• Negative conflict in result sharing:

When several agents, who are performing the same task, produce either inconsistent or antagonistic results. Here we are referring, in the later case, to different alternative answers to the same request (they can also be seen as different values for the same receiver's task input), or, in the former, to inconsistencies detected between different (but somewhat related) informations which are inputs to that task. The capability to deal with these situations and resolving the conflicts rely mostly on the availability of a functionality for measurement of the information quality as well as domain dependent knowledge.

4. A classification of cooperative and conflict situations

The study of different types of situations where agents may need cooperation, possibly to solve conflicts among them, has lead us to a broader Classification of Cooperation and Conflicts Cases (CCCs). This classification has guided us in suggesting appropriated policies - as much as possible, application domain independent - to deal with such cooperative situations. Some of these situations will require different degrees.of negotiation while others will need conflict resolution techniques to be applied. Cooperation is needed when there are different forms for agents of sharing concerns (either tasks or results).

Let us use the symbols N <---> M to represent the relationship "number of organizers" <---> "number of respondents".

CCC1 - One agent requests another specific agent for help and the latter is idle. This is a task sharing type of cooperation where a client/server protocol may be established. It is a 1<--->1 relationship.

CCC2 - One agent requests for help an agent which already has some tasks in its agenda. A negotiation regarding task priorities should be established between both. This task sharing situation still is a 1<--->1 relationship.

CCC3 - One agent requests some service from several other agents. Here we have several potential respondents to one organizer (a 1<--->M relationship) and a negotiation through contract net can take place.

CCC4 - Several agents having the role of organizers try to negotiate the execution of tasks with one single agent having the role of respondent. This task sharing cooperative situation is of N<--->1 type. But here, we still can discriminate some different possibilities for the interaction taking place :

4.1 - The respondent is being asked to simultaneously execute several different tasks.

4.2 - The respondent is being requested by the different organizers to execute the same task.

4.3 - In both former cases, it can happen that there is only one respondent being requested by each one of the organizers. This may be seen as a N*(1<--->1) relationship.

4.4 - Also it can happen that all interactions between the organizers and that particular respondent are just one part of a global negotiation taking place simultaneously. This means that the organizers may also be trying to negotiate with other different respondents. Hence this is a relationship of the type N<--->M (from the point of view of a respondent it is a relationship of the type N<--->1, and from the point of view of the organizer it is a relationship of type 1<--->M).

CCC5 - Several responses, which happen to be coincident, coming from different agents arrive at another one. This leads to a result sharing type of cooperative situation.

5.1 - If they are available at the same time to be taken into consideration together, the information credibility is augmented.

5.2 - If they arrive in different points in time and the first one has been already used, the others may be either ignored or just kept with the time tag updated.

CCC6 - Several results (answers to previous requests or information that has been volunteered) coming from different agents are different. Again this result sharing cooperative situation may be further decomposed into:

6.1 - Results are in conflict and they are completely antagonistic. A choice of one alternative must be made according to some heuristic (eg. respondents' credibility regarding that particular information).

6.2 - Results are in conflict because they contain some sort of inconsistency. A negotiation may take place between agents involving exchange of constraints and redoing of some sub-task by at least one of them.

6.3 - Results though different can be seen as complementary and, therefore, may be merged into a single one.

CCC7 - An agent did not get an expected input (from another agent in response to a previous request) in due time. The organizer must overcome this situation (eg. choosing another plan).

CCC8- An agent is no longer interested in an information it previously requested to others. This new situation should be communicated to those agents that had been contracted for the task.

CCC9- An agent deduces it can not accomplish a requested task in due time (according to some deadline - temporal or other - the organizer had attached to its request). This information must be sent to the organizer so that it can look for a possible alternative solution to its current problem.

CCC10 - Detection of a kind of deadlock (may be due to the fact of incorrect task decomposition).

CCC11 - Cooperation takes place as a consequence of data volunteered by some agents. These messages, whenever received, may either trigger a new task or enhance, through result sharing, available information. It also may imply an action of receiving agent to redo (restart) an already executed (under execution) task.

This Classification while being application independent will be useful for the sake of conflicts and cooperative situations identification within different types of applications.

5. Functionalities for negotiation and conflict resolution

Capabilities mentioned in Section 3 basically rely on the following functionalities, which are integrated in the agents cooperation layer:

- Computation of local task priorities
- Generation of task deadlines
- Switching priorities facility
- Generation of bid messages for negotiation
- Evaluation of bid messages for negotiation
- Conflict resolution on the same kind of input for a specific task
- Conflict resolution on different inputs for a specific task

These functionalities, also rely on information structures that are maintained in the agents knowledge bases (self model and agent acquaintance model). Moreover, the cooperation layer is provided with knowledge sources with appropriate knowledge to recognize when and how to use the described functionalities in order to decide on either cooperative or local activities..

These functionalities mainly involve generic knowledge which can be shared by all different kinds of application agents. Nevertheless, when dealing with specific conflicts, agents social activity has also to be knowledgeable about the domain in order to be effective on finding compromises.

5.1 Functionalities for task scheduling

Whenever an agent receives a request to execute a task (Tsk), it has to compute a priority (Pri) to attach to the task on the agenda. The agenda contains a list of tasks to be executed by the IS by decreasing order of priority.

The priority depends on the actual missing time (Mis) to the task deadline, an estimation of the average execution time (AvrExe) for that task (known at self model), and the time already consumed by that specific IS task execution (Exe) process.

The actual missing time to the deadline (Dea) associated with the task, is Mis = Dea - (Cur - Arr), where Cur is the absolute current time at some instant, and Arr is the absolute arriving time of the task request.

The value of Exe is 0 if the IS has not yet started to execute the task, and (Cur - Lau) if its state is running, where Lau is the absolute task launching time.

The Maximum Waiting time (MaxWai) until a task begins to be executed in order to enable the agent to meet the task deadline, is calculated as follows:

$$MaxWai = Mis - (AvrExe - Exe)$$

The priority of the task (Pri) will be an ordinal number, such that, the lower is MaxWai, the higher is Pri. Pri should be recalculated for all tasks of an agent (contained on the agenda) whenever a new task is arriving to the agent.

Tasks to be executed either by the agent IS itself or to be requested for execution to other agents, should have a deadline associated. Whenever an agent has deadline Dea1 to execute a task and it has to calculate new deadlines (Dea2), either for sub-tasks to be executed or for inputs to be received from others, these new deadlines are calculated by the formula:

$$Dea2 = Dea1 - (AvrExe + \Delta t)$$

where Δt is a quantum of tolerance time (e.g. 10% of AvrExe for communications and other possible operations).

In order to be sensitive to specific urgencies and constraints, as it may be the case for requests made under a client/server protocol, it may be useful if the agent is able to switch priorities between a new incoming task and another task already in the agenda. This functionality is used when a task requested under a client/server protocol would have to be rejected either because the calculated expected time to have the task executed is greater than the deadline issued, or because its inclusion into the agenda would delay other tasks beyond their deadlines.

Whenever a task request is received, to be accepted or rejected depends on the following computations:

First, according to current agent agenda, the priority that task could have in the case of a future acceptance is calculated.

Then, considering included the new task in the agenda, for each task in the agenda, that has priority lower than the new one and also for this new one, their new values of expected time necessary for being executed are computed. If any of these values is greater than the corresponding missing time, the new task can not be accepted.

The new expected time (Exp_i) necessary to complete a task (Tsk_i)will be:

$$Exp_i = \Sigma_k (AvrExe_k - Exe_k)/N + AvrExe_i$$

where k refers to tasks with priorities $Pri_k > Pri_i$, and N is the level of that Agent (IS) multi-tasking capability.

The switching priorities functionality will try, in these circumstances, to find out another task in the agenda with greater priority, that is already being executed or can be executed by another agent meeting its deadline. If this is the case for the task Tsk_j, it will recalculate the new expected times if positions are switched between Tsk_j and Tsk_i. If the new expected times of the task under consideration Tsk_i, and the ones with lesser priority are admissible according to their missing times (dependent on their deadlines), then the priorities are switched.

5.2 Functionalities for negotiation

On receiving a request for negotiation (message of type[2]: "task_announcement($Task_j$, $Agent_i$, Deadline)" or "slave($Task_j$, $Agent_i$, Deadline)" the cooperation layer of an agent will produce a bid message expressing the expected time needed for having the task executed.

As we have seen before, the priority this task could have is calculated , and then, considering it included in the correct position in the agenda, the expected time for this task execution as well as for the others with lower priorities than this specific one are checked against their deadlines.

A bid message will be issued with information about the possibility to accept the task, the expected time necessary to execute it, and the expected quality of the results (known at self model).

The agent that receives the bid messages (possible from different respondents) will evaluate them according to the following rule belonging to the correspondent knowledge source:

- From all bids whose expected execution time is lesser than the deadline, choose the agent that produces better quality of results.

5.3 Functionalities for conflict resolution

For each task that can be executed by more than one agent, it should exist at least one agent Ag_{cr} able to analyze all available results (answers) for that task. This agent, expert on resolving conflicts for that specific task, is the one in charge of sending the final agreed result to the organizer (the agent which had made the request).

An agent is able to sense (by consulting the dynamic part of its acquaintance model) that another agent is also working to answer the same request (either because they have been both contracted or they are voluntarily willing to help). When this happens, the results have to be sent to Ag_{cr}. This agent will consult in its own self model the appropriate slot with the procedure to deal with that kind of conflict, which may be based on domain dependent knowledge.

Agent Ag_{cr} will be the responsible for issuing a result, meeting task deadline calculated from the value of the missing time received.

A different kind of conflict also occurs whenever the organizer gets different inputs to the task it wants to execute, and these different inputs (which are all needed) are in some way inconsistent. This inconsistency may be due to the fact that interrelated inputs do not belong to the same context, what can be detected. In more complex situations the conflict detection should be of the Intelligent System responsibility.

6. Example

[2] There are two kinds of requests for negotiation: "task_announcement" and "slave".

The first kind of request is used by an organizer, that finds in its acquaintance models, more than one agent able to execute that task. If the organizer only finds one acquaintance that might execute the task, it will send it a "slave" request for negotiation.

In this section, we present an example of cooperation, including conflict resolution, in a realistic scenario which have been proposed by Iberdrola, a Spanish company which is responsible for the management of large networks for distribution of electrical energy. This scenario relates with a situation involving mainly two different agents, AAA (Alarm Analysis Agent) and BRS (Breakers and Relays Supervisor), when executing a task ("Hypothesis Generation Task") for generating a list of possible faulty elements in the network.

Agents AAA and BRS are both able to perform, differently, the same tasks. Their main task is to find out the elements of the network at fault. They receive, from a third agent, CSI (Control System Interface), blocks of alarms with indication of a disturbance. BRS also receives chronological information about the alarms, if available. If BRS has the complete information about the chronological alarms, its diagnosis of the faulty elements is more reliable than the one made by AAA, although it takes longer.

But when CSI does not get all of the alarms with chronological information they will be sent without time stamps. BRS is able to detect missing chronological alarms because they are replaced by the non-chronological messages. If there are missing chronological alarms, the quality of the results produced by BRS decreases.

Also it is possible that the alarms take longer to arrive to BRS than to AAA, because there is more data to transfer. Typically both agents will not complete the Hypothesis Generation Task at the same time.

To diagnose where the disturbance took place, both AAA and BRS generate a list of elements possibly at fault (Hypothesis Generation Task) and then analyze one by one each hypothesis for validation (Hypothesis Validation Task). This last task is very time consuming and would benefit if it is possible to shorten the list of hypothesis. Another agent, BAI (Black out Area Identifier) calculates the initial black out area (Initial Black out Area Task) based on the first set of alarms it receives from the CSI when there is a disturbance. If this information arrives to AAA or BRS, before or during their Hypothesis Validation Task, these agents may filter the list of hypothesis generated, eliminating any element that does not belong to the initial black out area (Hypothesis Refinement Task) hence reducing the length of the hypothesis list. Hypothesis Validation Task may need to be restarted but then, it will execute faster.

Several different interactions are possible to happen, that lead to conflict situations, but we will consider the following one:

Both BRS and AAA will calculate a list of possible faulty elements, with a certainty factor attached to every element. These certainty factors are computed by the rules through which they were deduced and they reflect not only the relative reliability of the diagnosis of each agent but also the reliability of the received information (inputs for the task), such as the percentage of non-chronological information received by the BRS.

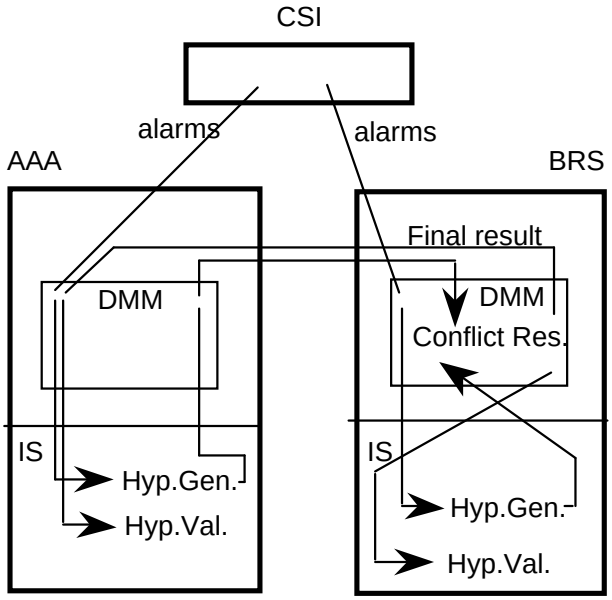

Fig.: Conflict resolution on the Iberdrola scenario

The output of this task will be:

For BRS: [E1brs-CF1brs, E2brs-CF2brs, ...]

For AAA: [E1aaa-CF1aaa, E2aaa-CF2aaa, ...]

Let us suppose that Agent BRS is in charge of resolving the conflicts associated with the results of this task. AAA and BRS would sense that both were running the same task. This will cause that instead of directing the results of these tasks to the agents that requested them or volunteering them to any agent, they will send their results to the agent appointed to resolve conflicts for this specific task (BRS).

These messages have the following syntax:

my_answer(AgSender, Task, Result,
AgDestination, MissingTime)

The agent which is in charge of solving conflicts for a specific task being handled, will be always the responsible to issue a result meeting task deadlines (deduced from the missing time received).

In this case, BRS will wait for two messages of "my_answer" type (from AAA and itself) while the deadline is not reached. Once one message has been

received, if a second message does not arrive in the deadline time specified by the received one, the received message will be forward to the destination agent.

If both messages did arrive in due time, the "conflict_resolution" procedure for this task will be invoked to produce a unique result.

The "conflict_resolution" procedure is situation dependent and, for this particular case, creates a list with all the elements contained in any "Result" list from all "my_answer" messages.

The elements that belong to both "Result" lists see their certainty factor increased (CF=CF1+(1-CF1)*CF2), while the elements contained only in one "Result" list keep their certainty factors. The new "Result" list will have the same syntax and will be sorted by decreasing order of the computed certainty factors.

This new list will be now considered as the final result of this task and will be sent to the destination agent.

7. Conclusion

We have here presented what we believe to be the basic knowledge for enabling coarse grain, semi-autonomous expert systems, to behave as cooperative agents in a multi-agent community.

UPShell is intended to be used for very different applications like electrical distribution network management or a flexible robotic manufacturing cell.

We briefly presented a tool for generating cooperative expert systems which include facilities for local task scheduling, negotiation, task and result sharing, as well as conflict resolution. Such high level features also rely on more basic functionalities like management of deadlines and priorities.

A tentative classification of cooperative and conflict cases whenever multiple agents interact is also presented.

Acknowledgements

This work has been developed under the support of Esprit project P2256: Archon. We thank all other Archon partners: Atlas Elektronik GMBH, Amber, CERN, CNRG-NTUA Athens, EA Technology, Framentec-Cognitec, FWI University of Amsterdam, Iberdrola, Iridia-ULB, JRC ISPRA, Labein, Volmac, Queen Mary and Westfield College.

We give special thanks to our colleague Rui Camacho Ferreira da Silva for his valuable contribution to the initial development of UPShell.

References

[1] - Eugénio Oliveira, Rui Camacho
"A Tool For Cooperating Expert Systems"
Proceedings of the 1st Expert Systems World Congress, Pergamon Press, 1991

[2] - Thies Wittig (Editor)
"ARCHON: Architecture for Cooperative Multi-Agent Systems"
Ellis Horwood 1992

[3] - R. Smith; R. Davis
"Frameworks for Cooperation in Distributed Problem Solving" in Readings in Distributed A.I.
Edited by Alan H. Bond and Les Gasser
Morgan Kaufmann Publishers, 1988

[4] - M. Klein & A. Baskin
"A Computational Model for Conflict Resolution in Cooperative Design" in Cooperating KBS
Springer-Verlag, 1990