An Agent Communication Platform Developed in Object Oriented Prolog

Torbjørn S. Dahl†, Siani Pearson††, Chris W. Preist††

†Department of Computing
Imperial College
London SW7 2BZ, UK
tsd@doc.ic.ac.uk

††Intelligent Networked Computing Laboratory
Mediated Communication Systems department
Hewlett-Packard Laboratories Bristol
Filton Road, Stoke Gifford
Bristol BS12 6QZ, UK
cwp@hplb.hpl.hp.com, siani@hplb.hpl.hp.com

1. Introduction

In this paper, we present the design and implementation of a platform for the development of multi-agent systems. The platform is implemented in Prolog++, an object oriented extension of Prolog.

The motivation behind this work is to allow the exploration of various macro-architectures for organising communication between, and coordination of, agents. We are interested in studying the different possibilities, and understanding their advantages and disadvantages.

We have chosen a reference application which can be used to compare these alternatives; namely the allocation of tasks in a team of computer support engineers. The computer support team (CST) is contacted by its clients with requests to perform tasks, such as installing new software, fixing a printer, etc. They also give preferences as to how the task is to be handled - such as a time they would like it carried out, what priority they consider it to have, and possibly which team members they would prefer to do it. The team must negotiate with the clients to agree who will perform each task, and when they will do it. The agreements must satisfy the needs of each client, and fall within their preferences when possible, must use the team effectively, and must allow the team members some degree of influence over the decisions which are made.

Full details of this reference application, together with the design decisions taken in the first agent-based system applied to it, are given in [Pearson *et al*, 96]. In this paper, we concentrate primarily on the design and implementation of the underlying platform. To illustrate the use of the platform, we use examples from this first system.

2. Design Overview

The agent platform consists of three parts;

- It provides an agent shell, which can be replicated and instantiated with information and plans specific to a particular agent.

- It provides a simulation of network communications to allow messages to be passed back and forth between agents.

- It provides a set of communications messages which agents can use to send information to each other, and to negotiate over the provision of services.

We have chosen to adopt the Beliefs-Desires-Intentions (BDI) architecture [Georgeff *et al*, 86], [Rao *et al*, 95] for the internals of the agent shell. A BDI agent consists of a message handler, a beliefs database, a planner and a goal queue. We have also chosen to have a database storing a record of all messages that an agent has received and sent (Figure 1).

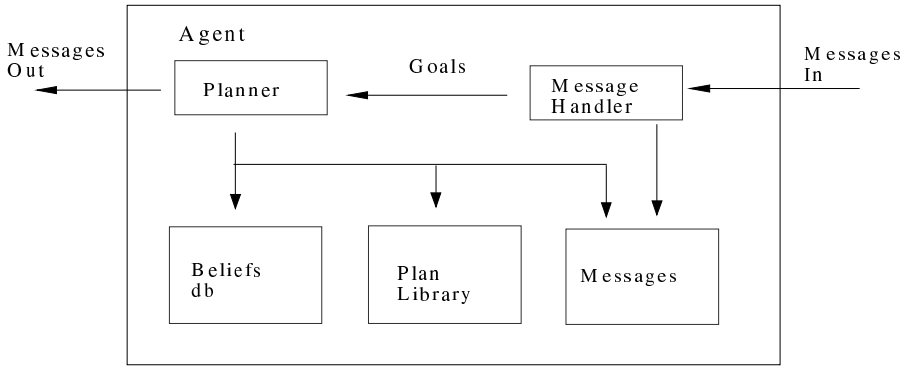


Figure 1: The internal agent architecture

The message handler processes incoming messages, altering the goal queue (corresponding to desires) and beliefs database as appropriate. It can add new goals to the goal queue, and unsuspend partly solved goals which are awaiting an incoming message before continuing. The planner uses plans from the plan library in an attempt to satisfy the goals. Plans are written in Horn clause logic, and can add messages to the outgoing message queue. They can suspend themselves, waiting for certain incoming messages and message types. A plan can also access the beliefs database and message database. The beliefs database contains beliefs about the world, such as information about what expertise this agent has, or current tasks it has committed to do. The message history database contains a record of all messages it has received. The agent shell provides the planner, beliefs database access predicates, message handler, etc., and is instantiated with specific plans, beliefs, belief queries and message handling rules to generate a particular agent.

The network simulator simulates, to a certain extent, parallel execution of agents as they pass messages between each other. It does this by allocating timeslices to each agent, but also maintaining the concept of each agent being at a given time in the system. The agent which is at the earliest time is given permission to execute next, until it wishes to send a message, at which time it suspends and another agent executes. When agents send messages, the network notes the time of the sending agent, and adds a transit time to give the arrival time of the message. It will only hand this message to its recipient when the receiving agent reaches the message arrival time and gets a timeslice.. Further details of this approach will be given in section 4.

The communications messages are based on the standard agent communications language, KQML [Genesereth *et al*, 92]. However, as KQML is focused primarily on the problem of information provision and exchange, rather than the more complex problem of service provision, it is necessary to add performatives to handle negotiation of services. These are taken from COOL [Barbuceanu *et al*, 96], an extension of KQML, and ADEPT [Alty *et al*, 96] where possible. For an overview of other approaches see [Mueller, 96]

The performatives we are using are:

- *advertise* - for an agent to register its capabilities with another. (From KQML)

- *ask, tell* - for exchanging information between agents. (From KQML)

- *can_you_do, can_do, cant_do* - to allow one agent to explore what another agent is capable of doing, with no commitment on either side. (From ADEPT)

- *propose, counterpropose, accept, reject* - for two agents to negotiate about a contract to perform a given task. (From ADEPT and COOL)

- *commit, cancel* - when negotiation is complete, for an agent to commit to the contract the other agent has accepted, or to cancel the contract before commitment is made. (From COOL)

- *request-cancellation, renegotiate* - to allow an agent to request another agent to drop an existing contract they have agreed on; either to renegotiate the parameters of the contract, or to cancel it altogether. (New performatives.)

The initial macro-architecture we use to solve the task allocation problem is based on the facilitator architecture, where one agent acts as a central point of contact with a team of service providers [Genesereth *et al* 94],[Finn *et al* 94](Figure 2). In our system, we have agents representing clients of the CST, agents representing members of CST, and a facilitator agent which takes responsibility for ensuring that a task requested is provided, when reasonable. The CST member agents are independent, and do not have to obey the facilitator blindly. However, they are expected to cooperate with the facilitator.

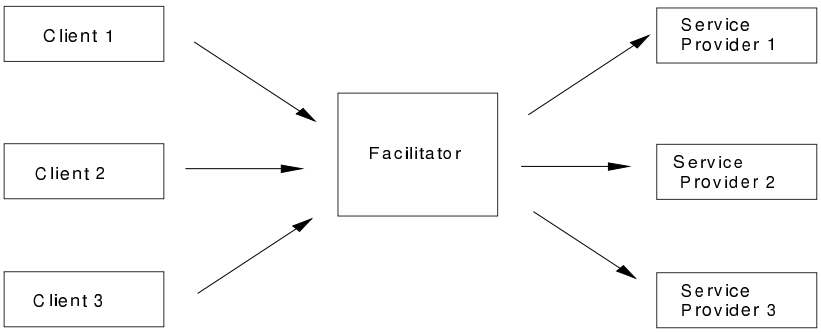


Figure 2: The initial architecture adopted

A client agent acts as an interface to the client, who is responsible for making their own decisions. The client agent uses plans which query the user for information. Its beliefs database contains information about the client (name, location, etc.) which can be queried by other agents.

A CST agent has information about a CST member, such as their skills and responsibilities, stored in its beliefs database. The database also contains a representation of the person's diary, showing when they are available, when they have committed to doing a task, and when they are busy for other reasons. In the current implementation, CST agents are considered cooperative, and will agree to take on any task they can do unless they have a high workload (i.e. < 20% time free).

The hotline agent holds basic information about the abilities and responsibilities of CST members who have registered with it, in its beliefs database. It receives incoming tasks from clients, and negotiates with CST agents to assign the task. If the task cannot be assigned straightforwardly within the constraints requested by the client, it will negotiate with the client to relax these constraints and/or with CST members to transfer other tasks, or to delay less urgent tasks, in an effort to accommodate the new task.

We base our approach to the modeling task allocation problem on that used in [Jennings *et al*, 96] to model business processes. We adopt the concept of contract, a simplified version of their service level agreement, to capture the notion of an agreement as to how a service is to be provided to the client. In our domain, this will contain information about who will perform the task requested, when they will do it, and what priority it is given.

3. Choice of Language

When selecting the language to implement the system in, the four main alternatives we considered were; Smalltalk, Java, Prolog and Prolog++.

Certain aspects of the system specification suggested that an object-oriented paradigm would be appropriate;

- A system that consists of a collection of agents has a strong modular structure. An object oriented language would be a very natural choice to implement this.

- Agent communication through messages can be naturally implemented by method calls in an object oriented paradigm.

Certain aspects also suggested that a logic programming approach would be useful;

- The planner and message handler would require rulebases. A logic programming language allows such rulebases to be implemented straightforwardly.

- A logic programming language provides a unification mechanism in its interpreter, saving the need to write one.

- A logic programming language can express the beliefs database as a deductive database very easily, and can be used to implement queries to it straightforwardly.

- The planner required an interpreter of the rulebase with certain special abilities, such as the ability to suspend itself and continue later. A logic programming language can be used to write such specialised interpreters (meta interpreters) in a straightforward way.

- Because the logic programming approach provides a lot of the building blocks we need, it can be used to produce and modify prototypes rapidly.

Because of these factors, we chose to use Prolog++, which provides both logic programming and object oriented features. On the down side, it is a relatively untested tool (being an add-on to LPA Prolog, rather than a product in its own right), and provides less sophisticated networking interfaces than Java or Smalltalk. As our system was developed for experimentation rather than deployment, this was considered an acceptable tradeoff.
As is recommended in the literature on objects in logic programming [Moss 94] we used object orientation at a relatively high level of granularity. All basic datastructures were kept as Prolog structures. The different subparts of the agents' architecture like the message handler, the plan library and the beliefs database were the finest level of objects. From this level the subparts were collected in an agent object and the agent objects communicated with each other through a network object. Objects were therefore used to modularise the code, create instances of modules, and to provide sophisticated message passing between modules, rather than as a fully-fledged programming paradigm as in Smalltalk. We sometimes also used objects as data structures in the beliefs database.

4. The Network Simulator

4.1 Simulation vs. True Distribution

An agent-based system should allow agents to be independent of each other - they should run as separate processes, possibly on separate machines, and communicate asynchronously. We had a choice between implementing this for real, using OLE, or implementing a simulation. We opted to go for the simulation, for the following reasons;

- A simulated network could be controlled easily. Parameters such as machine load and communication load would be under the control of the users of the system to a higher degree than would be possible even with a network dedicated to this experiment.

- A simulation requires fewer resources. Several experiments could be performed in parallel on different machines or even on the same machine.

- A simulation is easier to implement. The C interface in the Prolog used was only able to handle 16bit C code. Running a 32bit operating system, this would demand a translation layer between 16 and 32bit C code as well as a C network library of Microsoft windows system calls.

The main argument against a simulated network was that we might not simulate appropriately all the possible scenarios that could be found on a real network. To reduce this risk, advice from network programming experts internal to Hewlett-Packard was taken to try to make the simulation as accurate as possible. We hope, in the future, to implement the system in a truly distributed environment.

4.2 Factors to consider in the simulation

It is important that problems that can occur on real networks can also occur on the simulated network, to ensure that the agents are able to handle such situations. Handling communications problems on a network is very important if an agent is to be truly autonomous, and the effects of this on the message load and quality of solution cannot be ignored if the goal is to do experiments with results that will apply to truly autonomous agent systems.

Two major factors were taken into consideration when designing the network simulator.

- Machine speeds and availability; Every machine on a network will have a different speed according to hardware and load. In the simulation every agent is simulated to be on an individual machine. Any of these machines could go down in time.

- Network load and quality; Different parts of a network can have different loads affecting the times spent by messages in transfer between machines. Networks can also lose messages completely.

These factors create a set of classical problems which an agent system would have to deal with. The problems considered are;

- Deadlock: Two processes suspended each waiting for a message from the other process.

- Livelock: A number of processors performing actions interleaved in such a way that it stops any of them from making any progress.

- Racing conditions: Messages arriving in an order different from the order they were sent in.

- Lost messages: Sent messages not reaching the intended receiver, or machines going down and not sending or responding to messages as expected.

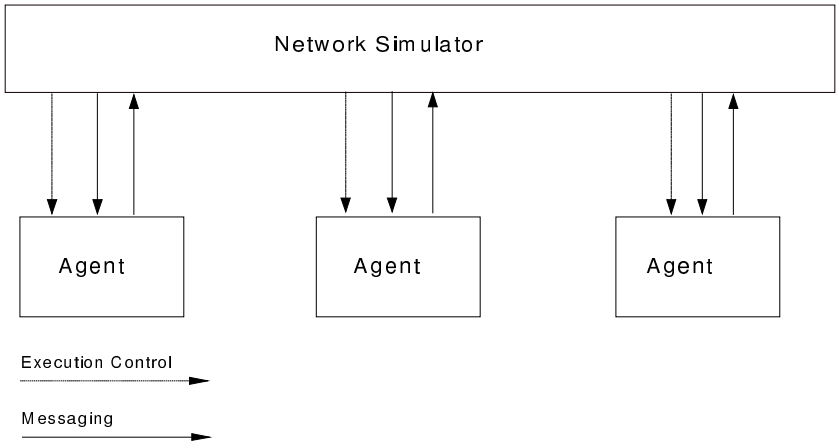


Figure 3: Communication between the network simulator and agents.

4.3 Design of the Network Simulator

The network simulator consists of a single object, and has two major functions. It simulates concurrent execution of the agents, and handles message passing between them. To simulate concurrent execution of agents, it allocates permission to run to one agent at a time, until the agent halts or sends a message. Each agent has a local concept of the time it is at, and the network simulator will ask the agent which is at the earliest time to execute itself. To pass messages between agents, it notes the time of the sending agent, adds a delay to simulate the time in transit, and places it in the incoming message queue of the receiving agent together with a record of its arrival time. The receiving agent will receive it when it listens to its incoming message queue, and the receiving agents time is equal to or later than the arrival time of the message.

4.3.1 Simulated Concurrency

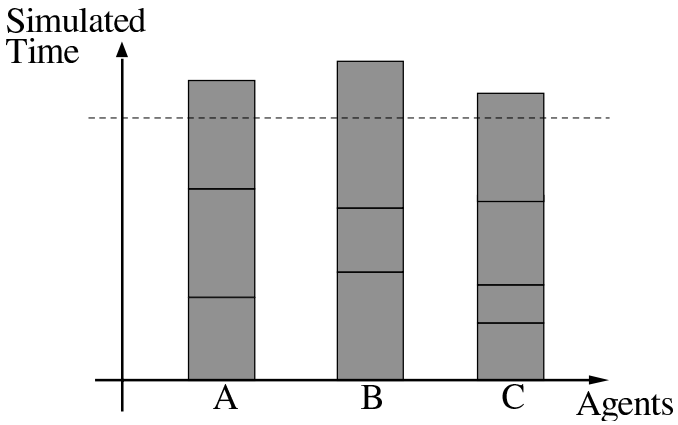
The central execution loop of the network simulator is the *run_agents* method. This is a simplified version of this loop:

```
run_agents(Limit,Max_slot_length):-
      find_lowest(@agents_times,(Agent,
                  Agent_start_time,Agent_current_time)),
            Agent_current_time =< Limit,
            Agent<-run_agent,
            slot_length(Max_slot_length,Slot_length),
            New_current_time is Agent_current_time + Slot_length,
            change_agent_time(Agent,New_current_time),,
            run_agents_limit(Limit,Max_slot_length).
```

The routine firstly determines which agent is at the lowest time (i.e. the one which needs to 'catch up' with the others) and, if it is less than the limit time for the length of the simulation, allows it to execute by sending it the method *run_agent*. When it has finished, we add a random amount - Slot_length - to its time, to simulate the length of time its run took. This amount of time represents the time between when the agent listens for its incoming messages to the moment it finishes processing these, possibly sending outgoing messages. To ease debugging, the seed used to generate the random numbers can be entered explicitly, so allowing repeated runs to behave identically if needed.

A limitation of this approach is that it does not allow messages to arrive when an agent is in mid-execution, interrupting it and forcing it to change its behavior. An agent can only receive messages at the start of its execution cycle, and send messages at the end of its execution cycle. However, we feel this is reasonable, as the execution cycle is intended only to do a small amount each time, before again checking for new messages.

The diagram on the next page shows how the agents execution is built up of blocks of random time length. The agent with the lowest time value will always be chosen to execute its next block and that block is then added to its bar. When all agents are above the end time for the simulation, it will stop.



4.3.2 Message handling

The modes of messaging that are allowed by the network is single agent to single agent messaging through the method *send* and single agent to all other agents through the method *broadcast*. An agent can check for incoming messages by using the *listen* method.

The network stores messages sent, together with their arrival time. When an agent listens for messages, the network sees to it that the listening agent only hears the messages that have arrived before the agent's current timestamp.

Racing conditions are not completely implemented in the network. If two agents send messages one after the other, then the random transit time can result in them swapping in transit , with the second one sent arriving first. However, two messages that are sent by the same agent will always arrive in the order they were sent. The simulation could be expanded to separate the random execution time from the random transfer time. This would give all the possible conflicts that have to be considered when one tries to avoid racing condition related problems.

We have also chosen not to implement the loss of messages in the network. Handling lost messages would require a lot of error handling within an agent which, due to time constraints, we did not want to address in the first phase of the project. [Barbuceanu *et al* 95] have shown how such error handling can be incorporated into the message handling protocol of an agent.

4.3.3 Assessment of this approach

By choosing to simulate the network, we simplified the task of writing the agent platform. As a result, we were able to put more effort into the areas we wished to explore, rather than being forced to first deal with writing a working distributed net of agents.

Care must be taken at a later point to make sure the simulation is as realistic as possible to ensure as high a level of correctness in the data gathered from the experiments. As mentioned above this could include separating agent execution time and message transfer time, making the network lose messages and making machines crash.

The most serious disadvantage is that there is no way at the moment to get an interrupt signal to an executing agent. If we decided to include this in the network simulation, the underlying design would require significant modification.

5. Design of the Agent

The architecture

The internal agent architecture is based on that described in [Rao *et al*, 95]. There is a message handler object, a planner object with a plan library and a beliefs database object. These subparts are coordinated through an execution cycle in the agent object.

This cycle executes once every time the agent is sent a *run_agent* message by the network simulator.
A simplified version of the agent execution cycle is given below:

```
run_agent:-!,
      Hndlr = @local_msg_handler,
      Hndlr<-run_msg_handler,
      local_msg_handler := Hndlr,
      Plnr = @local_planner,
      Plnr<-run_planner(Actions),
      local_planner := Plnr,
      take_actions(Actions).
```

- The cycle first runs the message handler object that gets all the incoming messages from the network, and updates the agent as appropriate. Based on the incoming messages, it adds goals to the goal queue, alters beliefs in the beliefs database, and unsuspends goals in the goal queue which are waiting for incoming messages.

- Next, the planner is run, executing the first goal in the goal queue which is not suspended. The actions the planner concludes should be taken are returned in the *Actions* variable. A plan may suspend, awaiting responses to its actions. In such a case, the goal is returned to the goal queue as a suspended goal.

- Lastly the cycle calls for the actions returned in the *Action* variable to be done. So far the only external action available is the *send* action, to send messages to other agents.

Message handler

The message handler acts as the interface between the network and the agent. The message handler invokes the *listen* method of the network simulator, and receives all messages which are waiting for the agent at this time. The message handler can act on an incoming message as follows;

- Add a new goal to the goal queue, based on what type of message it is, and how this particular agent treats such messages.

- Update the beliefs database.

- Unsuspending a plan which is awaiting this message before continuing.

The message handler also stores all incoming messages in the message store, and is responsible for sending messages from the agent, at the request of the planner.

Planner

The planner is responsible for attempting to achieve goals on the goal queue. It does this by executing plans in its plan library.
When the planner is called by the agent it takes the first goal on the goal queue which is not suspended. It then executes a plan corresponding to that goal, attempting to satisfy it. If the goal selected was previously suspended, then the plan continues to execute from the suspension point.

The planner is a Prolog meta-interpreter, which executes plans in a depth first fashion. It keeps track of each choicepoint, and stores this with a suspended plan. When a suspended plan is continued, the meta-interpreter firstly 'unwinds' the plan structure using this list of choicepoints to recreate the search tree corresponding to the plan.

Plans are currently 'pure' Prolog (i.e. no negation or non-logical features) augmented by certain special predicates. These allow the following functions;

- Querying the agent's beliefs database.
- Adding an action to the action variable
- Suspending the plan, and giving a condition on which to unsuspend the plan in the future. Currently, this condition is a logical expression (using ands and ors only) containing message templates.

To illustrate the structure of plans, we use an example taken from the task allocation system. This plan is a simplified version of part of that used by the facilitator agent to arrange for a given task request to be satisfied.

```
arrange (Client, Task Contact Request, message reference) :-

      % Arrange to prepare a contact within the requested
      % parameters of a client.

      belief( expected_time(Task,Duration) ),
      belief( responsible(Task,Agent_list) ),
      ppp( unique id( New_msg_reference) ),
      action( multicast(Agent_list,
            can_you_do(Task, Contract_request),New_msg_reference) ),
      ppp( create_suspend_conds(AgentList,
            New_msg_reference, Suspend_conds) ),
      suspend( await(Suspend_conds) ),
      choose_best_reply( New_msg_reference,
            Contract_request, Agent, Reply ),
      assign_task( Task, Agent, Reply ).
```

This plan is invoked in response to a goal to arrange a task for a client. It firstly calls the beliefs database, to determine how long the task will take, and which agents are responsible for doing this kind of task. It then sends a message to all responsible agents, asking them if they are able to carry out the task within the constraints proposed by the client. These messages are given a unique id, by a call out to prolog via 'ppp'. After the messages are sent, the plan suspends, waiting for replies to all the messages. When they have all arrived, it continues execution, firstly calling a sub plan to choose the best reply, and then calling a subplan to assign the task to the agent who gave the best reply.

If no adequate reply is received, this plan will fail, and another will be invoked, which will try other strategies for assigning the task.

Beliefs Database

The beliefs database is implemented as a Prolog deductive database, and information retrieval routines access this. Routines in the beliefs database should not make decisions, rather they should only provide information. For example, a routine which, given a task, returns all possible timeslots in which an agent could carry out this task, would live in the beliefs database - it returns beliefs about what is possible. However, a routine which makes a decision about in which timeslot to place the task should live in the planner.

As an example we give some of the facts and routines contained in the hotline agent from the task allocation system. These are used to return a list of agents responsible for a given task

```
%====================================
%
% responsible/2
%
% ouputs a list of agent ids whose agents are
% responsible for Task/Domain
%
% responsible(+Task,-AgentList)
% responsible(+Domain,-AgentList)
%
%====================================

      responsible(task(Name,Object),List):-
            Object@domain = Domain,
            collect_agents_resp(Domain,[],List).
      responsible(Domain,List):-
            collect_agents_resp(Domain,[],List).

      % collect_agents(+Domain,+InputList,-OutputList)
      % returns the list of agent ids responsible for Domain
      % ---------------------------------------------------
      collect_agents_resp(Domain,L,P):-
            is_responsible(Agent,domain(Domain)),
            collect_agents_resp(Domain,[Agent|L],P).
      collect_agents_resp(_,P,P).


is_responsible(kirsty,domain(unix_software)).
is_responsible(andy,domain(pc_hardware)).
is_responsible(matt,domain(pc_software)).
is_responsible(matt,domain(site_server)).
is_responsible(mark,domain(pc_software)).
is_responsible(torkwase,domain(pc_hardware)).
```

Note the use of an object to store information about the task. Object@domain is a Prolog++ construct to access the domain attribute of the task object.

The beliefs database is separated into access predicates and assertions. The access predicates are loaded into any agent of a given 'type' (i.e. all CST agents will have the same access predicates), and do not change. The assertions are dynamic, and are altered as new information comes into the agent. Different agents are initialized with different sets of assertions, depending on the current situation, or the situation being simulated.

6. The Task Allocation System

As an illustration of the capabilities of the platform, we describe the behavior of the task allocation system developed using it.

The current implementation of the system can demonstrate a variety of different negotiation scenarios corresponding to different situations which arise in task assignment. These scenarios currently involve about five client agents and five CST agents. The system presents a graphical display of messages passing between the different agents (see Figure 3). It is also possible to access information about an agent's internals such as its current goals and incoming messages. Client agents allow users to enter new tasks into the system; it is also possible to access and alter the diary of a CST agent. The hotline agent acts in the role of facilitator.

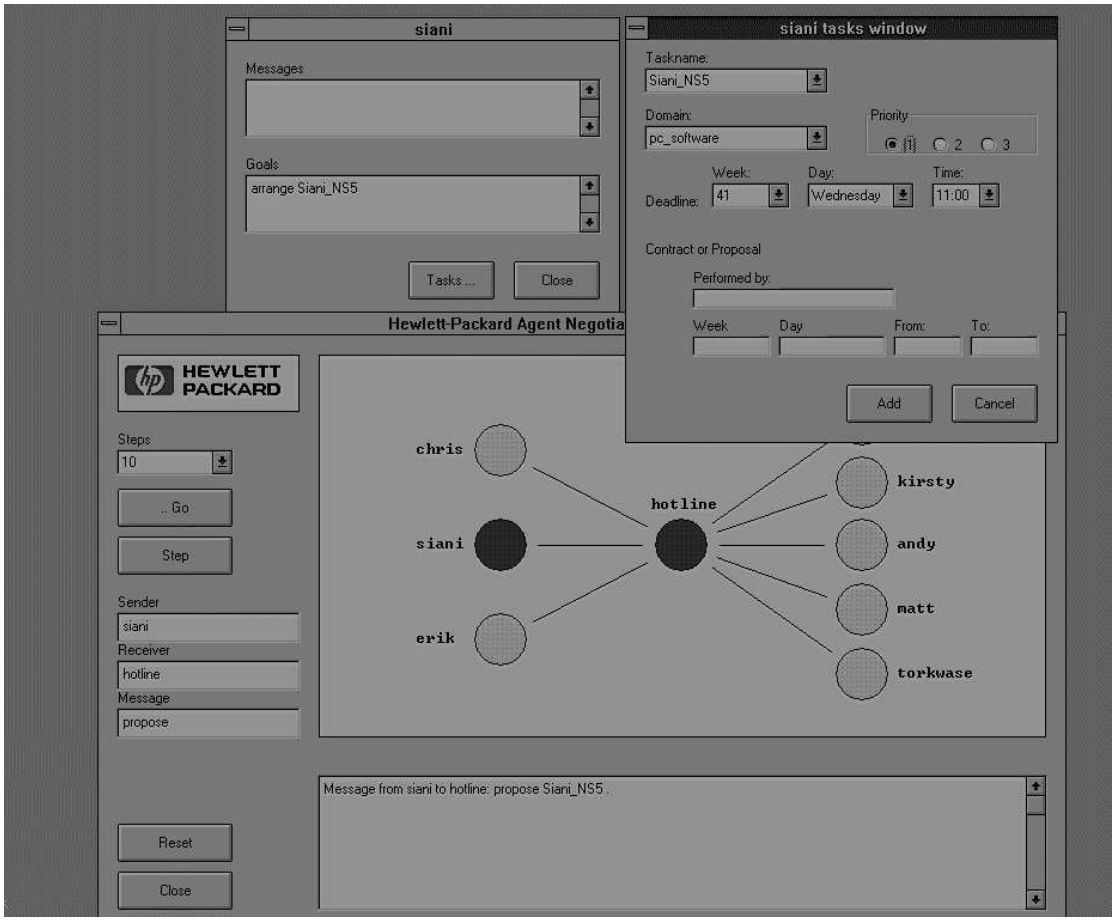


Figure 3: The system's user interface

The following short scenario will give a feel for the way in which our agents negotiate to get a particular task scheduled:

1. Client agent Siani is given the task of installing new PC software within the next three days by the user.
2. Siani sends a *propose* message to the hotline agent making this request.
3. Hotline multicasts a *can_you_do* message to all CST members responsible for PC software.

4. Two of the responsible CST agents, Mark and Kirsty, reply *can_do*, the others reply *cant_do*. Mark specifies that he could do the task tomorrow, while Kirsty says that she could do it in two days' time.
5. Hotline sends a *propose* to Mark asking him to carry out the task tomorrow.
6. Mark sends a *reject* message to hotline, because he has settled on another contract in the meantime, and so now his schedule is over 80% full.
7. Hotline sends a *propose* to Kirsty asking her to carry out the task in two days' time.
8. Kirsty sends an *accept* message to Facilitator.
9. Hotline sends an *accept* message to Siani, specifying that Kirsty will carry out the task in two days' time.
10. Siani sends a *commit* message to hotline.
11. Hotline sends a *commit* message to Kirsty, who then updates her schedule.

This is a very simple scenario in which one of the CST agents responsible for PC software is able to perform the task within the constraints formulated by the user. Other scenarios will arise when the situation is not that simple:

- When none of the responsible CST agents can do a task, the CST agents that are able to do it, but do not have responsibility are sent *can_do*'s.

- When they cannot do it, hotline relaxes the constraints specified by the user and sends new *can_do*s to the responsible CST agents. If a CST agent now *can_do* the contract, it is send a *propose*, and if it *accept*s, hotline sends a *counter_propose* to the client agent, in order to get its acceptance.

- If that does not work out, hotline sends a *ask(resolve_contract)* to the responsible CST agents, which in turn have to *tell* hotline what they would have to do to accommodate the contract. They may say that they can accept it (i.e. they are over 80% busy but still have enough space); they may suggest that they delay one of their less urgent contracts or that a previously arranged contract is transferred to another member of the CST. Hotline then selects one of these agents and sends a *propose* message with a high pressure measure to it. It will reply either by accepting or by requesting that an existing contract is delayed (using *renegotiate*) or transferred (using *request_cancellation*) to allow it to accommodate this new task. Hotline then negotiates with other CST agents, and possibly the client of the task being transferred, in an attempt to satisfy the agents' request.

- If all else fails, hotline sends a *reject* message to the client agent.

The resulting system ensures that tasks are roughly evenly distributed, in that if a CST member is 80% busy, they will only be given more tasks if no-one else is able to do them. If more skilled members of the CST are not given responsibility for less skilled tasks, it ensures that they are kept free of such tasks unless the less skilled members are overloaded (i.e. more than 80% busy). The system also ensures that new incoming urgent tasks are given space by moving less urgent tasks. It will move these less urgent tasks within the constraints requested by the client, when possible; if this is not possible, it will renegotiate with the client.

The system allows several such processes to take place simultaneously, by allowing agents to be running more than one plan at once. The *in_reply_to* message parameter ensures that an incoming message is used by the relevant plan only.

Because a contract will only be moved to make space for a more urgent contract, simultaneous processing or rescheduling will not result in recursive looping.

Conclusions

In this paper, we have shown that Prolog++, an object-oriented Prolog, can be used as a tool for the development of agent systems. The object-oriented programming capabilities give modularity and message passing. The Logic programming capabilities allow the rapid development of sophisticated rule bases and meta interpreters. The only disadvantage is that object-oriented Prologs are not used widely, so inevitably Prolog++ is not currently as robust or as well integrated as more traditional languages.

References

[Alty *et al*, 94] J. L. Alty, D. Griffiths, N.R. Jennings, E. H. Mamdani, A. Struthers, and M. E. Wiegand. ADEPT - Advanced Decision Environment for Process Tasks: Overview & Architecture. In *Proc. BCS Expert Systems 94 Conference* (Applications Track), Cambridge, UK, 359-371, 1994.

[Barbuceanu *et al*, 95] M. Barbuceanu and M.S. Fox. COOL: A language for describing coordination in multi-agent systems. In *Proc. First International Conference on Multi-Agent Systems*, MIT Press, 1995.

[Barbuceanu *et al*, 96] M. Barbuceanu and M.S. Fox. The Design of a Coordination Language for Multi-Agent Systems. In J. Muller, M. Wooldridge and N. Jennings, editors, *Working Notes of the Third International Workshop on Agent Theories, Architectures, and Languages*, ECAI-96, Budapest, 263-278, August 1996.

[Finin *et al*, 94] T. Finin and R. Fritzson. KQML as an Agent Communication Language. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94),* ACM Press, November 1994.

[Genesereth *et al*, 92] M.R. Genesereth and R.E. Fikes (editors). "Knowledge interchange format, version 3 reference manual", Computer Science Department, Stanford University, Technology Report Logiv-91-1, 1992. (http://www-ksl.stanford.edu/knowledge-sharing/papers/index.html).

[Genesereth *et al*, 1994] M.R. Genesereth and S.P. Ketchpel. Software Agents. *Communications of the ACM*, **37:7**, 48-53, 1994.

[Georgeff *et al*, 86] M.P. Georgeff and A.L. Lansky. Procedural knowledge. In *Proceedings of the IEEE Special Issue on Knowledge Representation*, volume 74, 1383-1398, 1986.

[Jennings *et al*, 96] N.R. Jennings, P. Faratin, M.J. Johnson, P.O. O'Brien and M.E. Wiegand. Using Intelligent Agents to Manage Business Processes. In *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM-96)*, 345-360, April 1996.

[Moss 94] Prolog++. The Power of Object Oriented and Logic Programming. Addison-Wesley' 1994.

[Mueller 96] H.J. Mueller. Negotiation Principles. In G. M. P. O'Hare and N. R. Jennings, editors, "Foundations of Distributed Artificial Intelligence", Wiley Interscience, 1996.

[Pearson et al 96] S.Pearson, C. Priest, T.Dahl and E. de Kroon.

An Agent-Based approach to task allocation in a Computer Support Team. Hewlett-Packard Technical report. Submitted to Practical Applications of Multi Agent Systems, 1997

[Rao *et al*, 95] A.S. Rao and M.P. Georgeff. BDI Agents: From Theory to Practice. In V., Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95),* San Francisco, USA, AAI Press, Menlo Park, 312-319, June 1995.

[Smith, 80] R.G. Smith. The contract net protocol: high-level communication and control in a distributed problem solver. *IEEE Trans. Comput.*, **29**, 1104-1113, 1980.