DESIGN AND IMPLEMENTATION OF ARCHON'S COORDINATION MODULE

Nick R. Jennings and J. A. Pople
Dept. Electronic Engineering, Queen Mary & Westfield College, University of London, Mile End Road, London E1 4NS, UK.
{n.r.jennings,j.pople}@qmw.ac.uk

ABSTRACT

This paper describes the design and implementation of a domain-independent reusable coordination module. This module is at the heart of the ARCHON architecture and has been used in the development of cooperating multi-agent systems in a number of real-world industrial applications including: electricity distribution management, electricity transportation management, cement factory control, flexible assembly robotic cells and particle accelerator control. The module is based upon the philosophy of providing a corpus of extensible generic knowledge about cooperation and situation assessment. Special prominence is given to the problem of controlling the reasoning within the coordination module.

1. INTRODUCTION

Systems composed of multiple, interacting components (agents) are becoming an increasingly popular means of building complex industrial control applications. The majority of these systems are functionally distributed and have subcomponents which are ordered in some hierarchical fashion with clear, predefined communication links. Although this modular approach increases the maintainability of the system, it keeps the overall control at a central location (i.e. a global controller coordinates the activities of all the subcomponents). This centralisation of control has two particular drawbacks for industrial control applications (Jennings & Wittig, 1992). Firstly for large applications with a number of distinct supervisory and control subcomponents, the activation of tasks in the sub-systems and the decision of what data to exchange between them depends on the state of the entire process. In a centrally controlled system this assessment requires the controller to take into account the different views of all the relevant sub-systems and can, therefore, lead to severe delays while the relevant information is assembled and the appropriate decisions are taken. Secondly it is difficult (sometimes impossible!) to perform the modifications required to integrate the large number of pre-existing (legacy) systems which are often found in industrial applications into one unifying whole.

To alleviate the decision-making bottleneck, increase the flexibility of data exchange and task activation and facilitate software reuse, the next stage in system design is to decentralise the control and allow the components to interact directly with one another. This approach not only allocates more responsibility to the sub-systems, but also

requires them to coordinate their tasks if the whole system is to interact in a coherent manner. Such coordination can be hand-crafted for each and every application or it can be undertaken in a more structured manner by developing a framework which can be re-used in a number of different scenarios (the approach described in this paper). The ARCHON (ARchitecture for Cooperative Heterogeneous ON-line systems) framework (Wittig, 1992), which provides the context for this work, has been used to build cooperative, multiple agent applications in the domains of: electricity distribution management (Varga *et al.*, 1994), electricity transportation management (Wittig, 1992, ch. 8); cement factory control (Stassinopoulos and Lembesis, 1993); flexible assembly robotic cells (Oliveira *et al.*, 1991) and particle accelerator control (Jennings *et al.*, 1993).

Within the ARCHON framework each agent is composed of a number of functional components, one of which is responsible for coordination in a decentralised environment. During the design and development of this *Planning and Coordination Module* (PCM) a number of crucial issues needed to be addressed: (i) what are the requirements for coordination in large, real-world industrial applications? (ii) what types of facilities should a general-purpose framework provide to an application developer? (iii) how can the reasoning of the coordination module be controlled so that the agent's objectives are satisfied? (iv) how can the coordination module be designed so that it responds rapidly to important events but also deals with events in a fair manner avoiding resource starvation? (v) how can a generic coordination module be tailored to fit a particular application? (vi) how can such a coordination module be implemented so that it meets the aforementioned desiderata?

This paper describes how the above issues were tackled and solved within the ARCHON framework. These experiences and insights are important for a number of different reasons. From the perspective of Distributed Artificial Intelligence (DAI) this work represents one of the first serious attempts to build a generic cooperation framework for large scale, real-world industrial applications. From the perspective of industrial control applications, this work highlights the feasibility of employing a cooperating systems metaphor and enables the problems associated with building decentralised control systems to be clearly stated and evaluated.

Section two presents a brief overview of the ARCHON architecture so that the work on the PCM can be placed in context. Section three details the philosophy of re-usable generic knowledge which lies behind the PCM and section four describes its implementation as an object-oriented rule-based system.

2. STRUCTURE OF AN ARCHON AGENT

ARCHON agents have two distinct components; an *Intelligent System* (IS) and an *ARCHON Layer* (see figure 1). The former may be pre-existing or may be purpose built and solves domain-level problems such as detecting disturbances in electricity networks or controlling the blower of a cement factory kiln. In the majority of ARCHON's applications, the community contains a number of different types of IS, including expert systems, databases and conventional numerical software. From the ARCHON Layer perspective the IS is composed of a number of atomic *tasks*, although

in terms of their actual implementation the tasks may themselves be relatively sophisticated problem solving activities involving branching and decision making. The ARCHON Layer is a meta-level controller which operates on the IS to ensure that its activities are coordinated with those of the others within the community. The separation of the domain and cooperation know-how into the IS and the ARCHON Layer respectively, allows pre-existing systems to be incorporated into the multiple agent community with relatively few modifications and allows the cooperation know-how to be re-used in a number of applications. Without this demarcation, extensive changes would be required to the existing systems in order to provide them with the necessary knowledge to interact with, and benefit from, the other agents in the community.

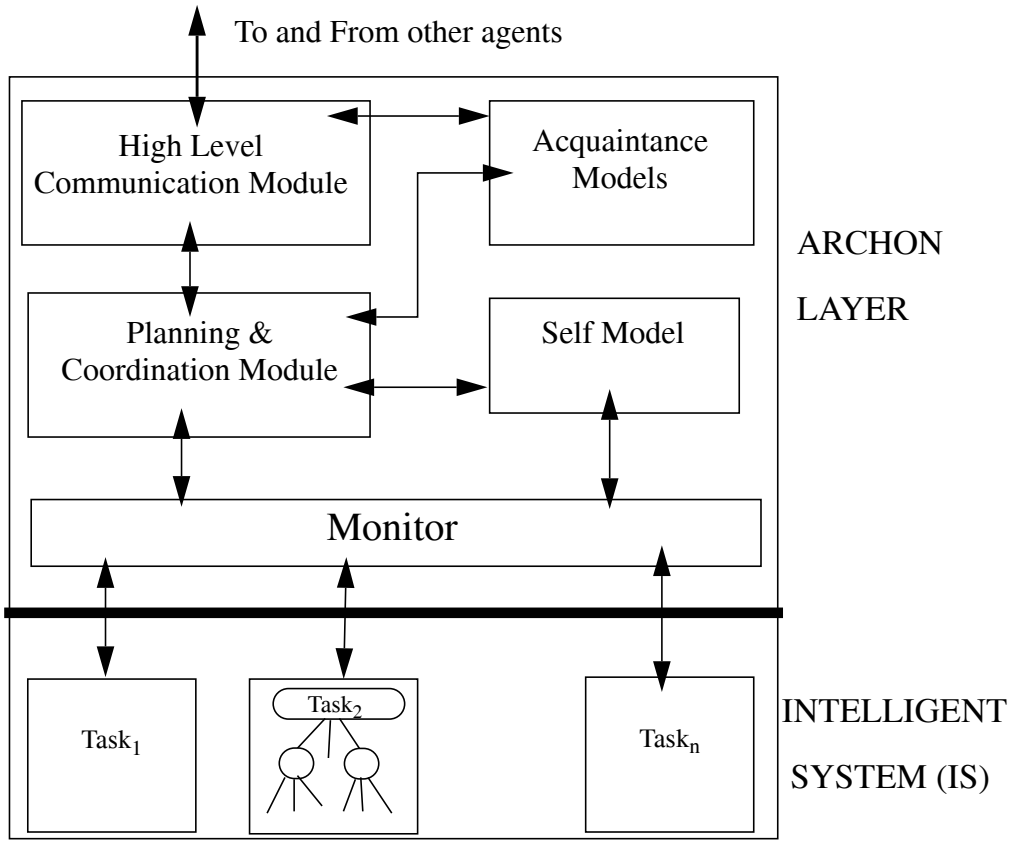

Figure 1: ARCHON Agent Architecture

Communication between agents is via message passing and is controlled through the *High Level Communication Module* (HLCM). This module is deemed High Level since it not only provides standard communication facilities (achieved through a Session Layer implementation) but also embodies services such as intelligent addressing and filtering. A message passing paradigm was chosen because of the physical distribution of the problem solving agents and the desire to conform to OSI standards.

The *Acquaintance Models* (AMs) are a representation of other agents in the community. Information maintained includes an acquaintance's skills, interests, current status,

workload and so on (Jennings *et al.*, 1992). These models are essential when coordinating activity because they provide a characterisation of the social problem solving context in which the agent has to operate. Much like the AMs represent other agents in the community, the *Self Model* (SM) is an abstract characterisation of the agent's underlying IS. It contains information about the current state of the IS and embodies a representation of the sequences of actions which can be executed by the ARCHON Layer in its underlying IS.

The *Monitor* organises locally executable activities and is responsible for passing information to and from the IS. *Skills* are the coarsest granularity at which these activities are described. Other ARCHON Layer components deal exclusively on the level of skills, but within the Monitor they are given a finer structure - corresponding to an OR-graph in which the named branches specify alternative solutions. The nodes of the graph are called *monitoring units* and they correspond to the invocation of individual tasks within the IS.

The PCM reasons about the agent's role in terms of the wider cooperating community. It has to assess the agent's current status and decide which actions should be taken in order to exploit interactions with others whilst ensuring that the agent contributes to the community's overall well being. Specific examples of the functionality supported include: deciding which skills should be executed locally and which should be delegated to others, directing requests for cooperation to appropriate agents, determining how to respond to requests from other agents and identifying when to disseminate timely information to acquaintances who would benefit from receiving it.

3. ARCHON'S PLANNING AND COORDINATION MODULE

3.1 Re-Usable Generic Cooperation Know-How

Analysis of a number of industrial control applications, highlighted a surprising degree of commonality in terms of their status and their characteristics. In the majority of cases studied, there were a number of automated components which were responsible for a well-defined portion of the overall process. Although the sub-systems made reference to the same environment, and hence decisions and actions by one component influenced those of another, they were not integrated. However when major events occurred (eg lightning storms in the electrical management domains) the operators of the individual components interacted verbally with one another to coordinate their problem solving activity (Jennings & Wittig, 1992).

In addition to this conceptual similarity at the operator level, the problem solving entities also had a number of broadly common characteristics. Most important from the PCM's point of view was the fact that the sub-systems were able to undertake significant amounts of processing in their own right - a consequence of the fact that most of them were originally intended to operate on their own or with minimal intervention from an operator. In terms of the cooperating system metaphor, this meant that agents would spend the majority of their time engaged in domain level computations and substantially less time on coordination activities and inter-agent communication. Also the number of cooperative interactions which would be needed were relatively small

in comparison to the number of activities undertaken within the domain level system. However interaction with other agents was needed to accomplish tasks that could not be performed locally and to supply information which was needed for problem solving but which could not be readily accessed. As well as these necessary interactions, there were a number of other new interactions, made possible by the sub-system integration, which could enhance the problem solving of the participating agents. Examples include: receiving relevant information which helps an agent prune its search space; agents cross-checking results by performing tasks which produce the same information using different data or a different approach; and the provision of more timely/accurate information for injection into the problem solving process. In general the mandatory interactions mirror those between the stand-alone system and its operator, whereas the new ones are similar to the types of interactions which took place between the operators when exceptional circumstances arose.

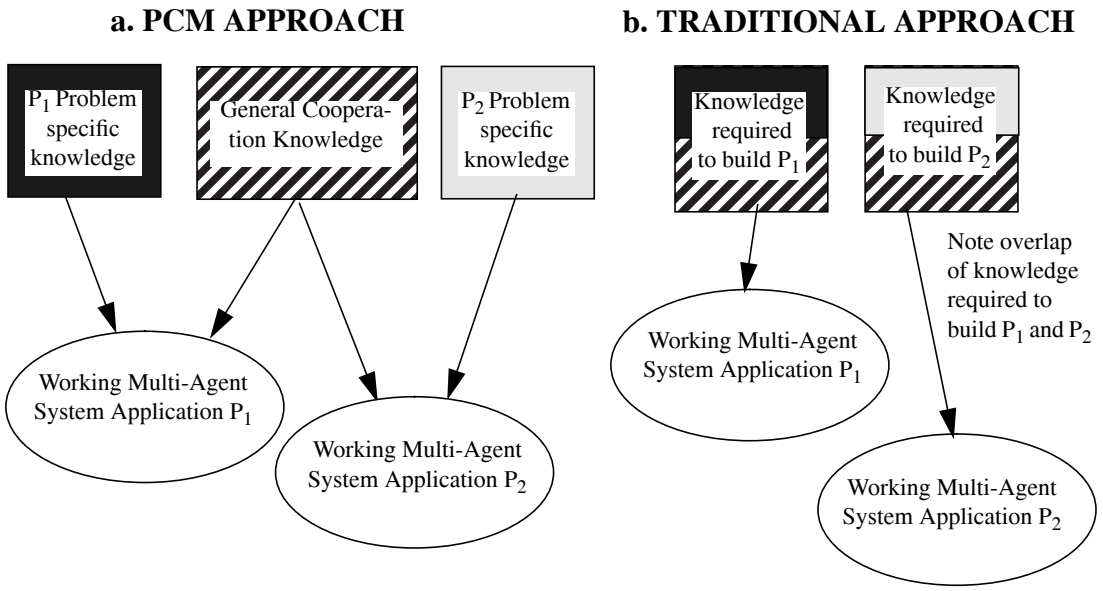

Figure 2: PCM Paradigm for Constructing Multi-Agent Systems

Within these well-defined constraints, it was decided that the greatest degree of support could be offered to the developers of ARCHON applications if a significant portion of the cooperative functionality could be provided as a core of inbuilt-knowledge. Thus rather than providing the developer with just programming features, he is presented with a library of knowledge about cooperation with which the application can be constructed[1]. This core can then be augmented, if necessary, with domain-specific cooperation knowledge in order to build the coordination mechanism for a particular multi-agent system (figure 2a). This approach contrasts with the conventional means of fabricating multi-agent systems in which the application developer is forced to con-

[1.] This approach has been advocated by a number of researchers concerned with the inherent difficulties and inefficiencies in the present software engineering development process (see for example (Blum, 1992; Cox, 1990; McDermott, 1990; Neches *et al.*, 1991; Stefik, 1986)).

tinually re-code a large proportion of essentially the same knowledge in each and every case (see figure 2b).

This re-usable knowledge approach means that each community member has the same core know-how about cooperation encoded in its PCM. The majority of the domain dependent data which is obviously needed to define individual behaviour is then located in the agent models. Examples of three such generic rules are as follows:

```
Rule1:   if   an agent has generated a piece of information i and
              it believes that i is of use to an acquaintance
         then send i to that acquaintance

Rule2:   if   an agent has a skill to perform and
              it is not able to perform it locally
         then seek assistance from another agent

Rule3:   if   an agent has finished executing skill s and
              s was undertaken because of a request from an
                    acquaintance
         then inform the acquaintance that s has finished and
                 return any results which have been produced
```

In the case of $Rule_1$, the acquaintance models contain a list of information that the other agents are interested in receiving and a condition under which they are interested. If the condition is met, then the second clause of the rule will be satisfied and the information will be sent. In the case of $Rule_2$, the self model is used to determine that the agent cannot complete the skill locally and the acquaintance models are used to identify those agents who are able to furnish the necessary skill. $Rule_3$ is triggered when the Monitor indicates that a skill has finished. At this point, the self model is examined to determine the reason for executing the skill. If this reason indicates that the skill was initiated as a result of a request from an acquaintance, then the information that the skill has finished and any relevant results which have been produced are returned to the originator. All of these rules are application independent and are tailored to a specific domain by the appropriate instantiation of the agent models.

3.2 Design Decisions

Being a key functional component of the ARCHON architecture, it is important that the PCM's design rationale and philosophy is made explicit and can be scrutinised. This allows the factors which influenced its internal structure, its representations and its control mechanism to be evaluated and assessed for appropriateness. Throughout the entire design process, the primary objective was to develop a domain-independent, re-usable mechanism whose operation would be as transparent and extensible as possible.

Given that the PCM is the overall director of, and broker between, the activity of the underlying IS and that of the agent's acquaintances, it has two obvious spheres of influence. Firstly to interact with other agents there must be an interface to the HLCM so that messages can be sent and received across the community. Likewise, an inter-

face to the Monitor is needed so that the PCM can influence the activities of the IS. This separation of concerns meant that the PCM's operations could be divided into two distinct groups; those related to managing the agent's local activity in a cooperative environment and those related to controlling the agent's social activities *per se*. For reasons of software modularity and clarity of design, these distinct functional roles were implemented as separate problem solving modules within the PCM - the former as the *Situation Assessment Module* (SAM) and the latter as the *Cooperation Module* (CM).

In more detail, the SAM is responsible for: deciding how data needed by the IS can be supplied (start activity locally or enlist the help of an acquaintance?); determining whether a request for the performance of a skill should be carried out locally; evaluating which skills should be started, in what order and with what data; deciding whether external requests should be met by starting a new skill or by exploiting an already active one; and evaluating whether new information should be passed onto the relevant active skills.

The CM has three primary objectives. Firstly it has to establish social interactions. This involves deciding how requests from the SAM can be best satisfied. Two forms of cooperation are presently supported: task and information sharing. In the former the agent asks an acquaintance to execute a skill or produce a specified piece of information, in the latter the agent spontaneously volunteers information to acquaintances who will benefit from receiving it (based on information specified in the acquaintance models). In both forms of interaction, the CM has to decide with which acquaintances the interaction should take place (i.e. which agents to request aid from and which agents to disseminate information to). With task sharing, the CM has to additionally decide between the *client-server* protocol and the *contract-net* protocol (Smith, 1980) as the means of determining how the task should be awarded to an acquaintance. With the client-server protocol the request is directed to just one acquaintance. With the contract-net protocol the agent advertises the activity it would like to be performed to all of those acquaintances who are capable of providing it. Upon receipt of the request, each acquaintance puts together a bid which specifies when and with what quality it could provide the service. When the originating agent receives all the bids, it evaluates them and establishes a contract for the activity with the most appropriate agent. The second primary objective of the CM is to maintain ongoing cooperative activities. So, for example, in the case in which an agent agrees to perform a skill because an acquaintance has asked it to, the social action's progress must be tracked to ensure that any relevant intermediate results are returned and that upon completion a final report describing the status and results of the requested activity is sent back to the originator. Finally the CM has to respond to cooperation initiations from other agents.

An early prototype of the PCM, called GRATE, which implemented the SAM and the CM as concurrent processes was built for evaluation purposes and applied to the domain of electricity transportation management (Jennings *et al.*, 1992) and particle accelerator control (Jennings *et al.*, 1993). As a consequence of this prototyping activity, three important points pertaining to the design of the PCM were highlighted (Jennings, 1992). Firstly the process of controlling the reasoning about cooperation and situation assessment needed to be significantly improved (GRATE just had a simple

looping structure and consequently did not respond quickly to important events). Secondly some organisational structure needed to be imposed on the knowledge embodied within the SAM and the CM if the application developer was to be able to add any domain specific cooperation know-how (in GRATE all the cooperation knowledge was just bundled together). Finally it must be possible to specify the objectives of the PCM so that important events can be more easily recognised. With respect to the final point, GRATE did not enable the application developer to introduce any bias into the reasoning process. So, for example, it was not possible to reflect the fact that the agent's main role in the community may be to provide services for the others (eg a database agent which contains large amounts of static information about the process being controlled). Nor was it possible to reflect the fact that another agent carries out such an important task that it should not be interrupted by low priority external requests (eg an expert system planning how the network can be repaired after a major fault should not be distracted by the receipt of unrequested information which is probably out of date). In the former case the developer needs the facilities to specify that external requests should be given a higher priority than locally generated ones and in the latter case that local activities should take precedence.

To rectify these problems it was decided that the PCM should be decomposed into smaller, more modular units and that some explicit reasoning about the invocation of situation assessment and cooperation functions needed to be introduced. Firstly rather than allowing the CM and the SAM to run as concurrent processes, and hence having no real control over their relative resource usage, an *overall controller* was introduced into the PCM (see figure 3). This controller maintains a high-level description of all the processing which the PCM has to undertake and decides whether situation assessment or cooperative functionality should be invoked next. As a second step the CM and the SAM were further divided into two *sub-modules* according to the interface which initiated their action. For the SAM this resulted in one sub-module for dealing with messages arriving from the Monitor and another for dealing with messages from the CM. Likewise for the CM, one group of operations were activated by messages arriving from the HLCM and a separate group were related to messages arriving from the SAM. These sub-modules act on the overall controller's instructions and use their more detailed knowledge of that sub-area of the PCM's operation to decide which types of functionality should be invoked and for what duration. As functionality invocation is now to occur as a result of reasoned activity, rather than being purely data driven, the messages arriving at a sub-module need to be stored in a buffer. Rather than having just one buffer, in which the structure of the activities to be performed would be lost, each sub-module maintains its own buffer for the messages that it has to process. Thirdly the individual functionalities of the PCM were represented as distinct blocks - called *operational rule blocks*. Thus the CM sub-module which processes messages from the HLCM is responsible for controlling the operational blocks which deal with the arrival of unrequested information, with requests to carry out problem solving activity for other agents and with the return of information which has been requested from other agents[2]. To facilitate the reasoning about invocation, each operational block is designated as having a particular orientation - *SERVES-SELF*

[2.] For reasons of clarity, only 12 of the PCM's operational blocks are shown in figure 3 - those not shown are related to: the rejection of cooperation requests, the resolution of conflicts and the contract-net protocol.

(SS) means that it progresses the agent's own local objectives; *SERVES-OTHERS* (SO) means that it progresses the processing of other community members and *MIXED* means that it has elements of both.

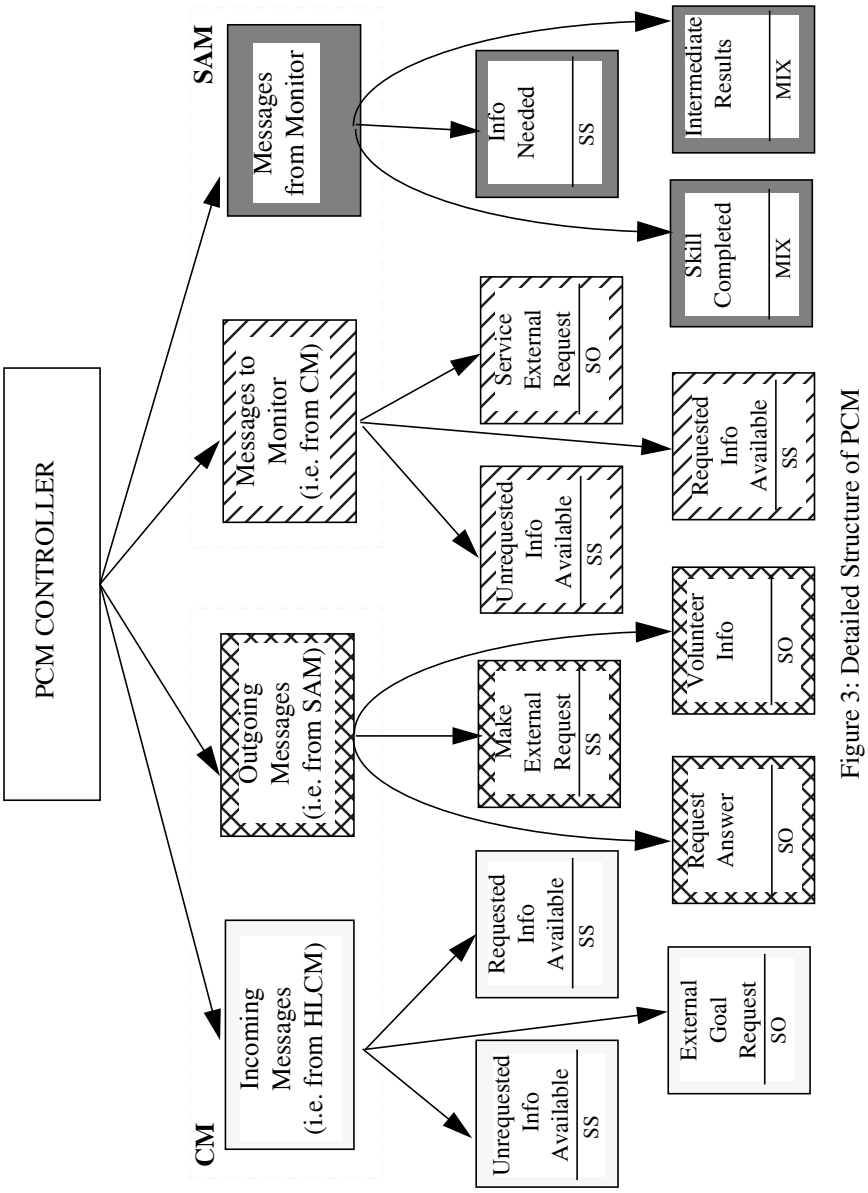

Figure 3: Detailed Structure of PCM

3.3 Meta-Level Control of the Coordination Process

Ensuring agents act coherently in an environment in which control decisions are decentralised is a difficult task which has resulted in the development of a variety of coordination mechanisms. In terms of the PCM, the major decision which affects the coherency of the system is the decision of what operational functionality to invoke at what time and for how long. In order to take this decision a number of factors needed

to be taken into consideration, ranging from long-term and relatively static information about the agent's objectives, to the immediate and constantly varying status information. The PCM's objectives are determined by examining the designated role of the agent in the community. Three alternatives are available, an agent's primary role may be: (i) SERVES-SELF in which case its main objective is to complete its own problem solving; (ii) SERVES-OTHERS in which case the agent is predominantly a server for the other community members; (iii) MIXED in which case the agent has a mixture of objectives, some of which are related to serving its own needs and some of which are related to helping others.

As well as invoking the appropriate operational functionality in order to fulfill the agent's role within the system, the control regime of the PCM has two other desiderata. Firstly it must avoid resource starvation and ensure that messages do not remain in the system for an unacceptable amount of time without being processed. Secondly because ARCHON is to be used in industrial applications, the decision making process which determines the functions to be invoked should not consume significant amounts of resource. This means a "satisficing" approach to control decisions is required in which relatively simple (and computationally cheap) criteria are applied to produce decisions which are "good enough". Optimal decisions, though desirable, may consume considerably more resources to make only marginally better decisions and may compromise ARCHON's time-criticality objectives.

The PCM's overall controller is responsible for selecting which of the four sub-modules should be processed at any one time and also for determining the amount of resource that should be consumed during this processing. The decision about sub-module activation is based on the policy set by the application developer:

`ROUND-ROBIN`: select the successor of the currently active sub-module until reach the end of the ordered list, in which case restart with the first element.

`SHORTEST-FIRST`: select the sub-module with the fewest messages to process.

`BUSIEST-FIRST`: select the sub-module with the most messages to process.

The amount of resource which should be consumed during a particular sub-module invocation depends on the loading of the PCM. If this load is high, then processing should be evenly spread between the sub-modules to ensure that all the important events are dealt with in a reasonable amount of time. If the PCM's load is relatively low, then some effort can be dedicated to processing less important messages and hence ensuring that long backlogs do not build up. The three choices which the controller can pass onto the chosen sub-module are as follows:

`CLEAR-BACKLOG`: clear up any backlogs which have built up.

`DEFAULT`: process important messages first but also ensure that no messages are waiting too long before being receiving attention.

`IMPORTANT-TASKS-ONLY`: only process message types which are important.

Within the constraints set by the Controller, the chosen sub-module has to decide which of its associated operational blocks will be invoked and how much processing each should undertake. So, for example, if the sub-module processing messages from the HLCM is chosen it may decide to process all of the messages corresponding to replies for information which have been made to acquaintances, one message providing unrequested information and no messages which are requests from other agents for the local agent's services. This selection will be based on the policy set by the controller, the priority of the individual operational blocks, the orientation of the operational blocks and the agent's orientation - see figure 4 for a more detailed description of the algorithm controlling this process.

```
CONST
 AgentOrientation ∈ {SERVES-SELF, SERVES-OTHERS, MIXED};
 SubModuleList ∈ {Incoming-Messages, Outgoing-Messages,
                  Messages-To-Monitor, Messages-From-Monitor};
 SelectionCriteria ∈ {ROUND-ROBIN, SHORTEST-FIRST, BUSIEST-FIRST};

LOOP FOREVER
     NextActive = select(SubModuleList, SelectionCriteria);
     IF NextActive ≠ nil THEN
       BEGIN
        PCMWorkloadStatus = EvaluateWorkload(SubModuleList);
        FORALL OperationalBlk(i) ∈ NextActive DO
           CASE PCMWorkloadStatus OF
             CLEAR-BACKLOG: Process all messages in buffer;
             NORMAL: IF orientation(OperationalBlk(i)) =
                              AgentOrientation
                        THEN process all associated messages
                        ELSE process first associated message;
             BUSY: IF orientation(OperationalBlk(i)) =
                              AgentOrientation OR
                        high-priority(OperationalBlk(i))
                     THEN process first associated message;
           ENDCASE
       ENDTHEN
ENDLOOP
```

Figure 4: PCM Control Algorithm

3.4 Instantiating the PCM for a Particular Application

The first step when instantiating the PCM is to analyse the inbuilt generic knowledge to determine whether it contains all the functionality and reasoning required to build the application. In all of the ARCHON applications which have been built so far, this generic knowledge has been sufficient and has not needed modification. However, in general the application builder may wish to augment this knowledge with cooperation know-how which is specific to the application being developed. In the present implementation this process is limited to the modification of existing functionality (i.e. the

developer can change the way in which unrequested information is processed, but a new message type cannot be added to the system, nor can the PCM structure be altered). Some examples of possible application specific cooperation knowledge which could be included for a given application are given below:

```
Rule1:   if   decide to seek assistance for skill s and
              more than one acquaintance can perform s and
              agent A1 can perform s
         then select agent A1 to perform s

Rule2:   if   executing skill s1 and
              receive request to do skill s2
         then reject request to perform s2

Rule3:   if   executing skill s and
              receive unrequested information i which is an
                     optional input for s and
              i is sent by agent A2
         then do not pass i onto s
```

$Rule_1$ could be added to the `make-external-request` operational block to indicate that acquaintance A1 is always the most appropriate (fastest, most accurate) agent to select to complete skill s. This rule may speed up the reasoning process in that it avoids the overhead associated with setting up a contract-net protocol to establish the most worthy agent. $Rule_2$ may be included in the `service-external-request` operational block if the designer knows that it is impossible to execute skills s1 and s2 in parallel and that after executing s1 it is no longer necessary (or appropriate) to execute s2. $Rule_3$ could be added to the `unrequested-info-available` operational block to encode the situation in which the application developer knows that acquaintance A2 sends unreliable data and hence it should not be used as an input for a skill which is particularly sensitive to noisy information.

This corpus of knowledge (generic plus application specific) together with its associated structure (as described in section 3.3) then forms the basis of the working PCM for a given application (see figure 5). Although the generic knowledge is widely applicable, the functionality which it supports will vary in usefulness between scenarios. So, for example, in some cases the main form of cooperation will be through the volunteering of unrequested information to relevant agents; whereas in others most cooperative interactions will be through explicit requests for services and information. To provide the necessary degree of flexibility, there are various parameters which the application builder is able to tweak in order to tailor the PCM to best fit the given problem. At the finest level, the developer may alter the relative priorities of the operational blocks. So, for example, the block which deals with unrequested information may be given a higher (lower) priority than that of explicit requests. Priority is specified as a number between 1 and 100 (the higher the number the greater the priority). The developer also has to specify what constitutes a high priority (eg high priorities are those greater than 80) - since it is important that these functions are processed in a timely and efficient manner.

The application designer then has to specify what constitutes a small number of mes-

sages for the PCM to process and what constitutes a large number. In between these two values, the PCM is operating in normal mode. These parameters are important because, as figure 4 indicates, the PCM behaves differently if it has a large number of messages to process, from if it has a normal amount from if it has a small number. Finally the policy for selecting the next sub-module needs to be fixed.

As figure 5 highlights, the process of tuning the parameters is iterative. The designer sets up a first approximation for each of the agents and then tests how they perform in their operational environment. As a result of this analysis, the parameters of one or more of the agents will be modified. This process continues until the community attains a satisfactory level of performance.

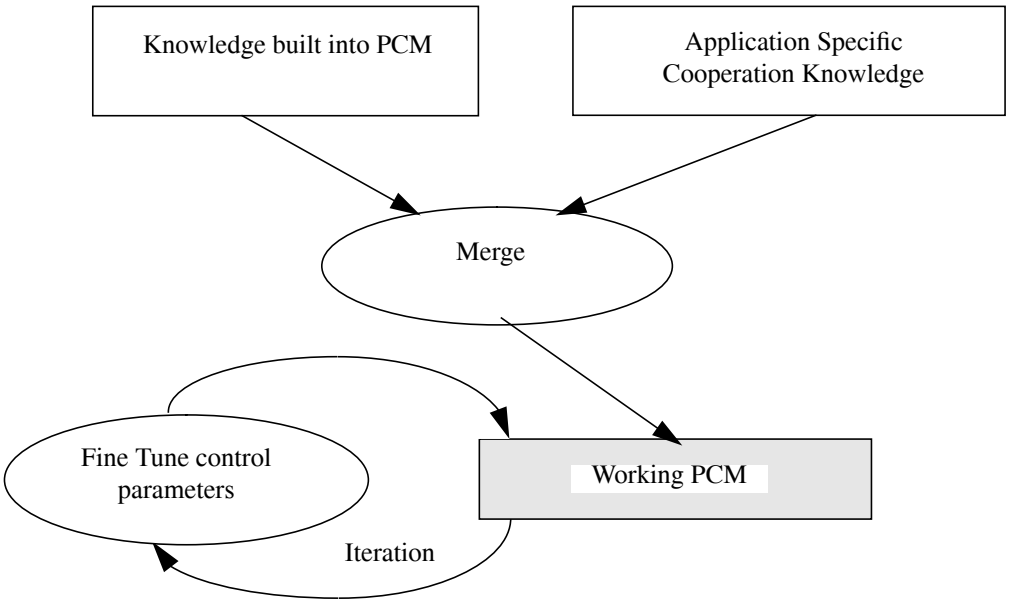

Figure 5: Instantiating the PCM

4. IMPLEMENTATION OF THE PLANNING & COORDINATION MODULE

4.1 Underlying Mechanisms

The PCM's modular and hierarchical design can be implemented using a number of different technologies. Alternatives which were considered include: writing simple procedures to perform each operational function and embedding the control regime into a fixed calling sequence; utilising a rule-based implementation to express the generic knowledge as declarative rules; using an object-oriented representation to encapsulate the various levels of the PCM in a class hierarchy. In order to comply with the overriding principles of clarity and extensibility, it was decided that both the operational and the control aspects should be implemented using a rule-based approach. However to retain the PCM's well-defined design structure in the imple-

mentation, it was decided that an object-oriented rule based system should be employed. Such an implementation platform allows the structure of the PCM to be represented in a unified manner - the control level objects reason about the sub-module objects, which in turn reason about operational level objects, which reason about objects external to the PCM (such as the acquaintance model objects, the self model objects, the PCM-Monitor interface objects). This underlying mechanism combines the ease of extensibility offered by the rule based approach, with the modelling capabilities of an explicit class hierarchy.

MIKIC-II, the particular object-oriented rule-based system which was chosen, allows both a structural and a functional decomposition of the class hierarchy to be devised (MIKIC, 1991). The structural elements act purely as a source of declarative knowledge (eg the agent models, the interface with the Monitor, etc.); whereas the functional entities, called *rulesets*, are associated with a particular group of rules and may be executed. The way in which rulesets are executed is specified by their *execution style* - `sequence` (pass through the rule list once), `iterate` (continually pass through the rule list until a halt flag is set) or `mutually-exclusive` (halt after the first successful rule firing) - and can be dynamically altered at run-time. Thus, for example, when a sub module controller is attempting to clear a backlog it will set the execution style of its operational rulesets to iterate until all of their associated messages have been processed. In contrast, when the workload is heavier and the sub-module controller is trying to pick out only the most important messages, it will set the execution style so that only a single instance of each message type is processed. The remaining attributes of rulesets can be divided into two distinct groups: (i) those inherited by all rulesets and which provide execution details; (ii) those specific to a particular class of rulesets and which contain static or dynamic declarative information about an instance. Further details of both types of attributes are provided in the following sub-section.

Within the object-oriented structure, individual production rules are created as separate instances and are associated with one particular ruleset. Individual rules can be applied to both specific instances of a class or to all elements of a class. For example, there are a number of operational rules which relate to the `BEHAVIOUR-INTERFACE-OBJECT` class[3]. One such rule has to identify new requests for information arising from the underlying IS and so must be applied to all instances of the class. In contrast, when a specific piece of requested information becomes available, the relevant rule only needs to be applied to the individual interface object instance which corresponds to the skill which initiated the request.

4.2 PCM Structures

The overall controller ruleset class has a single instance which is responsible for controlling the top-level operation of the PCM (see figure 6). As stated previously, this ruleset has to decide which sub-module to launch taking into account the prevailing conditions and the agent's long-term objectives. There are a number of attributes which are specific to this class and which provide information about the PCM's

[3]. This is a purely structural class which represents all the details of the interface between the PCM and the Monitor. Individual instances of the class represent a unique interface to a single skill

organisational framework. These attributes are given values by the application designer when the PCM is configured and include the following entries: *policy, agent orientation, large number of messages, small number of messages* and *high priority RP*. The meaning of these entries and their effect on the control of the PCM was discussed extensively in section 3.4. The other class specific attribute is *currently active submodule* which stores the name of the sub-module which is currently being processed - this is used when executing the round-robin policy of sub-module selection.

```
Ruleset-Top-Level-Control
 :name 'Ruleset-Top-Level-Control
 :fwd-rules '(tl-rule-0 tl-rule-1.....rule-n)
 :bwd-rules '(tl-rule-a.....)
 :sub-rules '(
              (Set-Active-Submodule (   (rules (tl-rule-0 tl-rule-1))
                                        (style mutually-exclusive)))
              (Set-Execution-Policy (   (rules (tl-rule-2 tl-rule-3 tl-rule-4))
                                        (style mutually-exclusive)))
              (Execute-Submodule (      (rules (tl-rule-a tl-rule-5))
                                         (style sequence))))
 :Execution-Style 'iterate
 :Currently-Active-Submodule :MESSAGES-TO-MONITOR
 :Policy :BUSIEST-FIRST
 :Agent-Orientation :SERVES-SELF
 :Large-Num-Messages 35
 :Small-Num-Messages 10
 :High-Priority-RP 75
 :Execution '(:evaluate-ruleset 'Ruleset-Top-Level-Control)
```

Figure 6: PCM's Top Level Control Ruleset

The remaining attributes depicted in figure 6 are inherited by all rulesets. *Execution* contains a calling function which, when it is evaluated, has the effect of executing the ruleset. As such, a rule belonging to one ruleset may invoke another ruleset by evaluating its execution attribute - this is in fact the mechanism by which the distribution of control spreads through the PCM's distinct control levels. Once a ruleset has had its execution attribute evaluated, the attributes *fwd-rules* and *bwd-rules* are referred to in conjunction with the execution style in order to actually execute it. The former specifies that the associated list of rules will be executed in a forward chaining mode, whereas the latter indicates that its associated rules should be executed in a backward chaining mode. In some cases it is desirable to decompose the operation of a ruleset into a number of separate functional blocks *(subrulesets)* each with their own execution strategy. This is especially useful when, within a single ruleset, some rules proceed directly from one another but others need to be repeated or are mutually exclusive. The top level controller has three subrulesets which are invoked sequentially each time a decision has to be made about the next sub-module to launch. Set-Active-Submodule has two mutually exclusive rules and identifies which submodule should be launched next (this corresponds to the select function of figure 4). Set-Execution-Policy updates the execution policy in response to the PCM's current workload

using knowledge of what is in the chosen submodule's input buffer and what constitutes a small and large number of messages to process (this corresponds to the EvaluateWorkload function of figure 4). Finally Execute-Submodule launches the chosen submodule (by invoking its execution attribute) and tries both tl-rule-a and tl-rule-5 (execution style sequence).

The sub-module controller ruleset class has four instances (one for each of the submodules). It inherits the attributes which are common to all rulesets and has the additional class specific attributes of *input buffer* and *execution policy*. The former maintains a pointer to the buffer where all the messages to be processed by this sub-module are stored. The latter indicates the overall state of the PCM (i.e. busy, normal, clear backlog). This ruleset is responsible for controlling the invocation of the individual operational rules (this corresponds to the body of the for loop of figure 4).

The final class of rulesets are the operational rule blocks - these have the class specific attributes of rule package orientation and rule package priority. As an illustration of this class, the ruleset for handling requests for information from locally executing skills is shown in figure 7 - as shown in figure 3, this ruleset is part of the SAM.

```
SAM-Info-Needed
 :name      ruleset-s-a-internal-info-request
 :fwd-rules  (rule-500 rule-501 rule-502a rule-502b rule-503 rule-504)
 :bwd-rules (rule-505)
 :sub-rules (  (launch-local-behaviour      (rules (rule-5010 rule-5011 rule-5012
                                                    rule-5013 rule-5014 rule-5015))
                                            (style mutually-exclusive)))
               (evaluate-cooperation-request      (rules (rule-5020 rule-5021
                                                          rule-1022 rule-1023))
                                                  (style mutually-exclusive))))
 :execution-style iterate          :orientation :SERVES-SELF          :priority 65
```

Figure 7: Internal Information Request Ruleset

As a means of obtaining a deeper insight into the type of generic knowledge encoded within the PCM, the rules associated with the `SAM-Info-Needed` ruleset are listed in full in figure 8. The first rule to be executed is rule 500 which examines all instances of the interface object class to determine whether any of them have an outstanding information request. If such a request is found, then a flag is set. Assuming that there is some processing to be done, the next step is to determine whether it should be done locally (rules 501, 502a and sub-ruleset launch-local-behaviour) or whether it should be handled via cooperative activity (rule 502b and sub-ruleset evaluate-cooperation-request)[4]. The remaining two forward rules (503 and 504) are con-

[4]. There is a basic assumption contained in the rules *502a* and *502b* which enforces the notion that whenever possible requests should be processed locally. This assumption may be reversed simply by modifying the rule antecedent or it may be weakened through the addition of some application specific rules. For example, it would be possible to add rules which ensured that requests were always handled locally unless the agent's workload was above a certain threshold.

cerned with controlling the ruleset's execution. If the sub-module controller has set the execution style to sequence, then neither 503 nor 504 will be fired and the ruleset will have finished executing. However if the execution style is set to iterate then one of 503 or 504 will be fired and the ruleset will be evaluated again (if there are more requests) or it will terminate (if there are no more requests).

:name rule-500
:rule IF there is a new request message,
THEN flag it as an outstanding request.
:for instances of behaviour interface class.

:name rule-501
:rule IF there is an outstanding request for information,
AND the information is locally available,
THEN provide the information and delete the outstanding request.
:for instances of behaviour interface class.

:name rule-502a
:rule IF there is an outstanding request for information,
AND it can be produced locally,
THEN call subruleset '**launch local behaviour**'.
:for instances of behaviour interface class.

:name rule-502b
:rule IF there is a an outstanding request for information,
AND it cannot be produced locally
THEN call subruleset '**evaluate cooperation request**'
:for instances of behaviour interface class.

:name rule-503
:rule IF the execution style for this ruleset is iterate
AND backward rule 505 is satisfied
THEN evaluate Execution attribute
:for this ruleset

:name rule-504
:rule IF the execution style for this ruleset is iterate
AND backward rule 505 is not satisfied
THEN stop iteration.
:for this ruleset

forward rules

:name rule-505
:rule IF there is a new request message
THEN return 'true'.
:for instances of behaviour interface class.

backward rule

Figure 8: PCM Situation Assessment Internal Request Handling Ruleset

Once it has been decided whether the information request will be dealt with locally or whether it will be dealt with by an acquaintance, the relevant sub-ruleset is invoked

(respectively, launch-local-behaviour and evaluate-cooperation-request).

The subruleset `launch-local-behaviour` (figure 9) contains the PCM's generic knowledge about how this processing should proceed if the information request is to be dealt with locally. It assumes that it is always better to utilise a skill which is already active, rather than start a new one (rules 5010 and 5011), but if no such skill is available it is preferable to launch a skill which can be started with the minimum of effort. Where assistance from other agents is unavoidable, the agent uses information about its acquaintances in order to make an informed choice about which skill to launch. For example, rule-5014 looks at the agent's previous, but as yet unsatisfied, information requests to see if they may fortuitously provide the inputs needed to launch a suitable behaviour.

Subruleset: **`Launch Local Behaviour`**
Execution Style: Mutually-Exclusive
:name rule-5010
:rule IF there is an active local skill that produces the requested info.
AND the request provides identical execution parameters
THEN append the request to this skill
:for instances of behaviour interface class.

:name rule-5011
:rule IF there is an active local skill that produces the requested info.
AND the request provides no execution parameters,
THEN append the request to the first of these skills
:for instances of behaviour interface class.

:name rule-5012
:rule IF there is a local skill capable of producing the requested info
AND the mandatory inputs are available for that skill,
THEN create a behaviour interface instance and launch the skill.

:name rule-5013
:rule IF there is a local skill capable of producing the requested info
AND the mandatory inputs to that skill can be produced locally
THEN create a behaviour interface instance and launch the skill.

:name rule-5014
:rule IF there is a local skill capable of producing the requested info.
AND the mandatory inputs to that skill have already been requested
THEN create a behaviour interface instance and launch the skill.

:name rule-5015
:rule IF there is a local skill capable of producing the requested info.
THEN create a behaviour interface instance and launch the skill.

Figure 9: Launch Local Behaviour Subruleset

As stated previously, the subruleset `evaluate-cooperation-request` (figure 10) is invoked if it is deemed desirable to satisfy an information request by interacting with an acquaintance. As with the launch local behaviour sub-ruleset, the objective is to minimise the amount of inter-agent activity. For example, if the agent has recently made a request for the same piece of information, then no new request is actually generated. Rather the agent notes in its acquaintance model that when the request is met, the information should be passed onto the new skill as well as the originating one (rule 5020). Rule 5022 demonstrates the pro-active nature of the PCM's situation assessment module. If it sees that the desired piece of information could potentially lead it to starting a new skill, as well as providing an input to an existing skill, then a request is made. This feature enables the agent to eagerly assemble relevant information which it will utilise in its subsequent problem solving activity.

Subruleset: **`Evaluate Cooperation Request`**
Execution Style: Mutually-Exclusive

:name rule-5020
:rule IF the request is identical in every respect to any outstanding request
THEN record the request as sent but do not send to the cooperation module
:for instances of behaviour interface class.

:name rule-5021
:rule IF the request is for mandatory input to the requesting skill.
THEN pass the request to the cooperation module
:for requesting instance of the behaviour interface class.

:name rule-5022
:rule IF the request is for information that may trigger a local skill
THEN pass the request to the cooperation module
:for instances of behaviour interface class.

:name rule-5023
:rule IF none of the above
THEN reject the request
:for requesting instance of the behaviour interface class.

Figure 10: Evaluate Cooperation Request Subruleset

4.3 PCM In Action

To provide an intuitive feel for the PCM's operation, an example of the interaction between the CM, the SAM and the other ARCHON layer modules will be given. Imagine that the Monitor has indicated, via the appropriate behaviour interface object, that it requires a particular piece of information to proceed. The first task is for the PCM's top level control ruleset to examine the status of each submodule and launch the most appropriate. For our purposes assume that the agent's orientation has been set to SERVES-SELF and that it is selecting sub-modules on a BUSIEST-FIRST basis. By examining each submodule's input buffer, the controller is able to identify the busiest

and hence the one which should be processed next. In this case assume that the situation assessment submodule MESSAGES-FROM-INTERFACE-OBJECT is the busiest and, because there are a large number of events to process, it is launched with an execution policy of IMPORTANT-TASKS-ONLY. As shown in figure 4, with this policy the submodule executes only those operational rulesets that have the same orientation as the agent and it uses a `sequence` execution style to ensure that only one instance of each message type is processed. As processing a request for information from a local skill is seen as a self serving function, the operational ruleset designed to handle such events (i.e. SAM-INFO-NEEDED) will be invoked by the submodule controller.

Once called, the SAM-INFO-NEEDED ruleset has to determine how the request should be met. Firstly it must decide whether it should be dealt with locally, through execution of a new skill or through an association with an already active one, or whether it is necessary to request the assistance of an acquaintance. For explanation purposes it is assumed that the desired information cannot be provided locally and that although it is not a mandatory input to the requesting skill it may trigger other local activity when it is received. Under the rules given in figure 10 this would result in an appropriate cooperation request being placed in the input buffer of the submodule designed to handle events coming from the SAM (i.e. MESSAGES-FROM-SAM). Once the currently active submodule has launched each of its correctly oriented operational rulesets, control returns to the top level where the process of choosing a new submodule and its execution policy is repeated. At some point, control will pass to the submodule which has the example cooperation request in its input buffer and the operational ruleset whose role is to determine which agent the cooperation request is best handled by (i.e. MAKE-EXTERNAL-REQUEST) will be launched.

The operational ruleset MAKE-EXTERNAL-REQUEST examines each of its acquaintance models and compiles a list of agents capable of satisfying the request. This ruleset must then select the most appropriate means of obtaining the information - this may be through the use of a contract-net protocol or it may be through the use of a simpler client-server protocol. Whichever mechanism is chosen, a message (or messages) conveying the need for information is sent out via the HLCM. When the information request is received by an acquaintance it goes through a number of phases. Assuming that the message is processed successfully it will take the following path: (i) from the HLCM to the cooperation module's INCOMING-MESSAGES submodule where it is processed by the EXTERNAL-GOAL-REQUEST operational ruleset; (ii) from the EXTERNAL-GOAL-REQUEST ruleset to the SAM's MESSAGES-TO-MONITOR submodule where it will be dealt with by the SERVICE-EXTERNAL-REQUEST operational ruleset; (iii) when the information has been produced it will be placed, by the appropriate skill, into the MESSAGES-FROM-MONITOR submodule where it will be processed by the SKILL-COMPLETED operational ruleset; (iv) from the SKILL-COMPLETED ruleset it will be placed into the OUTGOING-MESSAGES submodule where it will be processed by the REQUEST-ANSWER operational ruleset; (v) the REQUEST-ANSWER-RULESET will send the message back to the originating agent via the HLCM.

When returned to the originator, the desired information will flow from the HLCM to the INCOMING-MESSAGES submodule where it will be processed by the REQUESTED-INFO-AVAILABLE ruleset. From here it will pass to the MESSAGES-TO-MONITOR sub-

module where it will be passed to the appropriate interface object by the situation assessment's REQUESTED-INFO-AVAILABLE ruleset.

5. CONCLUSIONS

This paper has described the design and implementation of ARCHON's Planning and Coordination Module. This coordination module has been used to instantiate cooperation in a number of real world, industrial applications. The philosophy of utilising a corpus of inbuilt generic knowledge has been explained and its implementation in an object-oriented rule-based system detailed. There are still a number of issues associated with this approach which require further investigation: (i) the types of cooperative interactions are relatively straightforward, how will this approach cope with more complex scenarios?; (ii) will the corpus of general knowledge be appropriate for applications other than industrial control?; and (iii) the empirical effect on local and social problem solving of varying the PCM's control parameters needs to be analysed in a systematic way.

REFERENCES

Blum, B. I., (1992). The Fourth Decade of Software Engineering: Some Issues in Knowledge Management, *Journal of Intelligent and Cooperative Information Systems* 1 (3&4) 475-514.

Cox, B. J., (1990). Planning the Software Industrial Revolution. *IEEE Software*, Nov., 25-33.

Jennings, N. R., (1992). Using GRATE to Build Cooperating Agents for Industrial Control. *Proc. IFAC/IFIP/IMACS International Symposium on Artificial Intelligence in Real Time Control*, Delft, The Netherlands, 691-696.

Jennings, N. R., Mamdani, E. H., Laresgoiti, I., Perez, J., & Corera, J., (1992). GRATE: A General Framework for Cooperative Problem Solving. *Journal of Intelligent Systems Engineering* 1 (2) 102-114.

Jennings, N. R. & Wittig, T., (1992). ARCHON: Theory and Practice", In N. M. Avouris & L. Gasser (eds.), *Distributed Artificial Intelligence: Theory and Praxis*, Kluwer 179-196

Jennings, N. R., Varga, L. Z., Aarnts, R. P., Fuchs, J., & Skarek,P., (1993) Transforming Standalone Expert Systems into a Community of Cooperating Agents. *International Journal of Engineering Applications of Artificial Intelligence* 6 (4) 317-331.

McDermott, J., (1990),. Developing Software is Like Talking to Eskimos about Snow. *Proc. 9th National Conference on AI*, Boston, USA, 1130-1133.

MIKIC-II, (1991), A Multi-Purpose Inference Kernel for Industrial Control, User Manual.

Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., & Swartout, T.,

(1991). Enabling Technology for Knowledge Sharing. *AI Magazine* 36-56.

Oliveira, E., Camacho, R. & Ramos, C., (1991). A Multi-Agent Environment in Robotics. *Robotica* 4 (9).

Smith, R. G., (1980). The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver. *IEEE Trans. on Computers* 29 (12) 1104-1113.

Stassinopoulos, G. & Lembessis, E., (1993). Application of a Multi-Agent Cooperative Architecture to Process Control in the Cement Factory, ARCHON Technical Report 43/ 3-93.

Stefik, M., (1986). The Next Knowledge Medium. *AI Magazine* 7 (1) 34-46.

Varga, L. Z., Jennings, N. R., & Cockburn, D., (1994). Integrating Intelligent Systems into a Cooperating Community for Electricity Distribution Management. *Expert Systems with Applications 7* (4).

Wittig, T. (ed.) (1992). *ARCHON: An Architecture for Multi-Agent Systems.* Ellis Horwood.