Scientia Iranica, Vol. 8, No. 4, pp 229–240
© Sharif University of Technology, October 2001

SCIENTIA
IRANICA

Ensuring the Localization of Responsibilities in the Adaptive Agent Oriented Software Architecture (AAOSA)

B. Hodjat* and M. Amamiya[1]

Distribution in dynamic multi-agent systems is only justifiable if there is a level of localization guaranteed by the system for each domain the agents represent. Adaptive Agent Oriented Software Architecture (AAOSA) is a new dynamic approach to software design based on agent-oriented architecture. In this approach, agents are considered as adaptively communicating concurrent modules which are divided into white box modules, responsible for the communications as well as learning, and black box modules, responsible for the independent specialized processes of the agent. A distributed learning policy that takes advantage of this architecture is used for the purpose of system adaptability. Then, a method is proposed to ensure the localization of responsibilities in this multi-agent methodology.

INTRODUCTION

In the classical view of agent oriented systems, each agent is considered as an autonomous individual, the internals of which are not known, that conforms to a certain standard of communications and/or social laws with regard to other agents [1]. Architectures viewing such agents must introduce special purpose agents (e.g., broker agents, planner agents, interface agents...) to shape the structure into a unified entity desirable to the user [2,3]. The intelligent behavior of these key agents, with all their complexities, would be vital to the performance of the whole system.

On the other hand, methodologies dealing with the internal design of agents tend to view them primarily as intelligent decision-making beings. In these methodologies, techniques in artificial intelligence, natural language processing and machine learning seem to overshadow the agent's architecture, in many cases undermining the main purpose of the agent [4,5].

*. *Corresponding Author, Dejima, INC., 160 W. Santa Clara St., Ste. 102, San Jose, CA 95113, USA.*

1. *Department of Intelligent Systems, Graduate School of Information Science and Electrical Engineering, Kyushu University, 6-1 Kasugakoen, Kasuga-shi, Fukuoka 816, Japan.*

In Adaptive Agent Oriented Software Architecture (AAOSA), instead of using assisted coordination, in which agents rely on special system programs (facilitators) to achieve coordination [2], new agents supply other agents with information about their capabilities and needs. In order to obtain a working system from the beginning, the designers preprogram this information at startup. This approach is more efficient because it decreases the amount of communication that must take place and does not rely on the existence, capabilities, or biases of any other program [6].

One of the aspects that makes agents more attractive to be used in software than objects is their quality of volition. Using AI techniques, adaptive agents are able to judge their results, then modify their behavior (and thus their internal structure) to improve their perceived fitness. This modification may even effect and correct the domain of responsibility for that agent. On the other hand, because each agent is considered to be adaptive, there has to be a way to restrain the agents from intruding into other agents' domains.

In this paper, a method is proposed to limit each AAOSA agent's domain and present an example in interactive systems. First, a description of the Adaptive Agent Oriented Software Architecture [7] is presented. Then, an implementation of AAOSA for

interactive systems using a simple example is described. Finally, a simple distributed learning algorithm and the overall evaluation of the proposed system are presented. Moreover, some suggestions are made for future work in this area.

ADAPTIVE AGENT ORIENTED SOFTWARE ARCHITECTURE

Agents in AAOSA are adaptively communicating concurrent modules. The modules, therefore, consist of three main parts: a communications unit, a reward unit and a specialized processing unit. The first two units are called the white box and the third unit is the black box of an agent (Figure 1). The white box part of an agent is common in all AAOSA agents although some functions may be left unused in certain cases. Here, the black box is regarded simply unknown and completely left to the designer. The main responsibilities of each unit are as follow:

Communication Unit

This unit facilitates the communicative functions of the agent and has the following sub-systems:

- Input of received communication items: These items may be in a standard agent communication language such as KQML;
- Interpreting the input: Decides whether the process unit is capable of processing certain input, or it should be forwarded to another agent (or agents). Note that it is possible to send one request to more than one agent, thus creating competition among agents;
- Interpretation policy (e.g., a table): Determines what has to be done about the input. This policy could be improved with respect to the feedback received for each interpretation from the reward unit. Some preset policy is always desirable to make the system functional from the beginning. In the case of a system reset, the agent will revert to the basic hard-

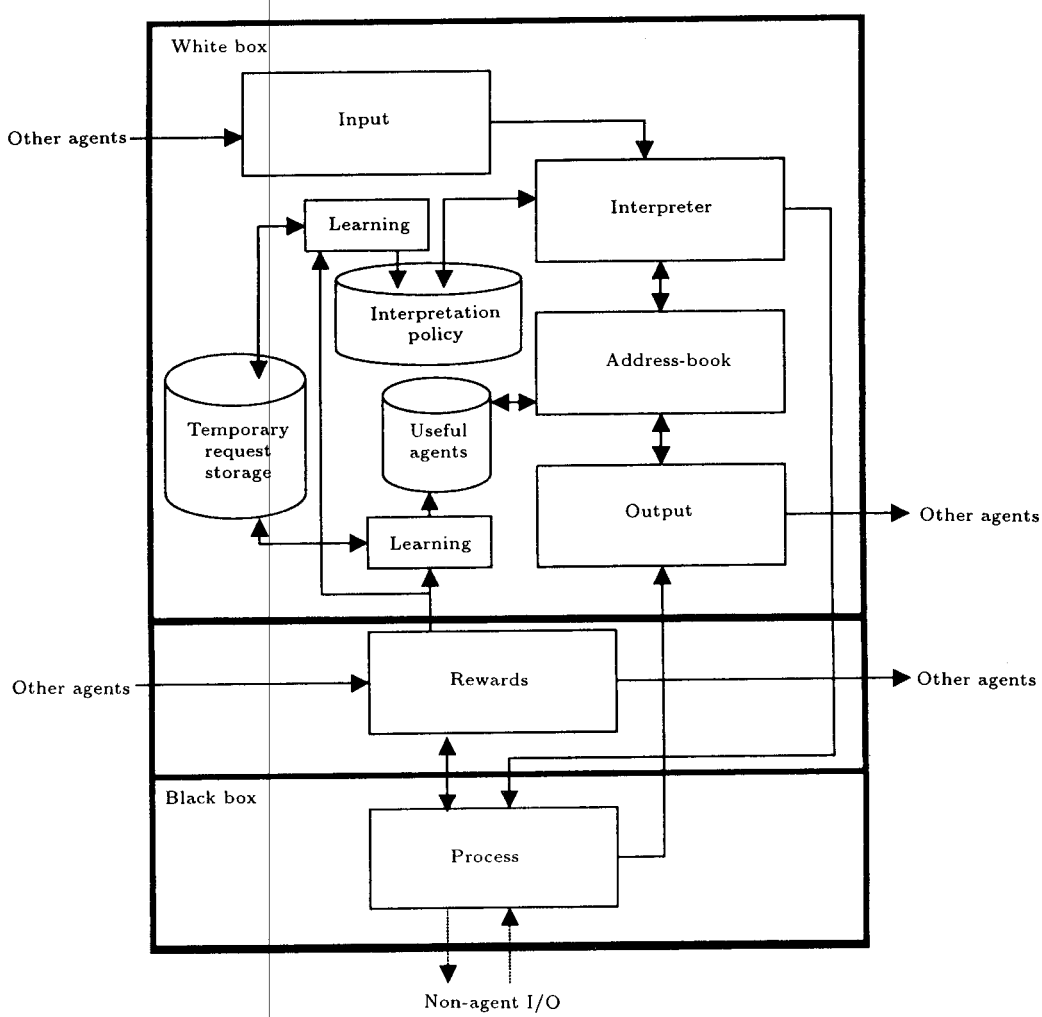

Figure 1. Each agent is comprised of a black box section (specialties) and a white box section (communications).

coded startup information. The interpretation policy is, therefore, comprised of a preset knowledge base and a number of learned knowledge bases acquired on a per-user basis. A 'learning' module is responsible for conflict resolutions in knowledge-based entries with regard to feedback received on the process of past requests. Past requests and the processes conducted on them are also stored in anticipation of their feedback;

- Address-book: keeps an address list of other agents known to be useful to this agent, or agents known as being able to process input that cannot be processed by this agent. Requests to other agents may occur when:
  1. The agent has received a request it does not know how to handle;
  2. The agent has processed a request and a number of new requests have been generated as a result.

  This implies that every agent has an address and there is a special name server unit present in every system to provide agents with their unique addresses (so that new agents can be introduced to the system at run time). This address list should be dynamic and, therefore, adaptive. This list may be limited, it also may contain information on agents that normally send their requests to this agent. In many cases, the address-book can be considered as an extension of the Interpretation Policy and, therefore, implemented as a single module.
- Output: Responsible for sending requests or outputs to appropriate agents, using the address-book. A confidence factor could be added to the output based on the interpretations made to resolve the input request or to redirect it. It will be shown later that this could be used when choosing from suggestions made by competing agents or output agents.

Reward Unit

Two kinds of rewards are processed by this module: outgoing and incoming. An agent is responsible for distributing and propagating rewards that are being fed back to it. (A special purpose agent is responsible for the interpretation of user input as feedback to individual user requests, which will then initiate the reward propagation process.) This unit will determine what portion of the incoming reward is deserved and how much should be propagated to requesting agents. The interpreter will update its interpretation policy using this feedback. The rewards will also serve as feedback to the address-book unit, helping it in adapting to the needs and specifications of other agents. The process unit could also make use of this feedback.

The rewards may not be the direct quantification of user states and in most cases will be interpretations of user actions made by an agent responsible for it. This point is further clarified later in this paper.

Processing Unit

This unit is considered as a black box by the methodology employed in this paper. The designer can use other methods that seem more suitable for implementing the processes that are unique to the requirements of this agent. The only constraint is that the process unit is limited to the facilities provided by the communication unit for its communications with other agents. The process unit may also use the reward unit to adapt its behavior with regard to the system. Note that each agent may have interactions outside of the agent community. Agents responsible for user I/O are an example of such interactions. These agents generally generate requests or initiate reward propagation in the community, or simply output results.

The white box module can easily be added to each program module as a transducer. According to its definition [6], the transducer mediates between the existing program (the process unit) and other agents. The advantage of using a transducer is that it requires no knowledge of the program other than its communication behavior.

The process unit has been mentioned as being able to conduct non-agent I/O. It is easy to consider I/O recipients (e.g., files or humans) as agents and make the program redirect its non-agent I/O through its transducer. Other approaches regarding agentification (wrapper and rewriting) are discussed in [6].

Design Issues

The AAOSA design methodology is essentially a bottom-up approach. The tasks necessary to achieve overall goals are identified and suitably decomposed [8]. Then, the data-flow between these tasks is determined. Through this way, pre-existing codes can also be incorporated in the design as non-decomposable tasks by wrapping them into the black-box of AAOSA agents.

The break up of software into sub-domains is the responsibility of the designer who should also define the interpretation policies. This is done by looking at the system input from each agent's point of view. It is important not to over-generalize in order to avoid claiming an input that really belongs to other agents. However, there is no need to be too conservative either. Designers should keep in mind that interpretations are done in the context of the communication path by which the input has arrived at the agent and resolving ambiguities that arise as a result of overlapping interpretations are the responsibility of up-chain agents.

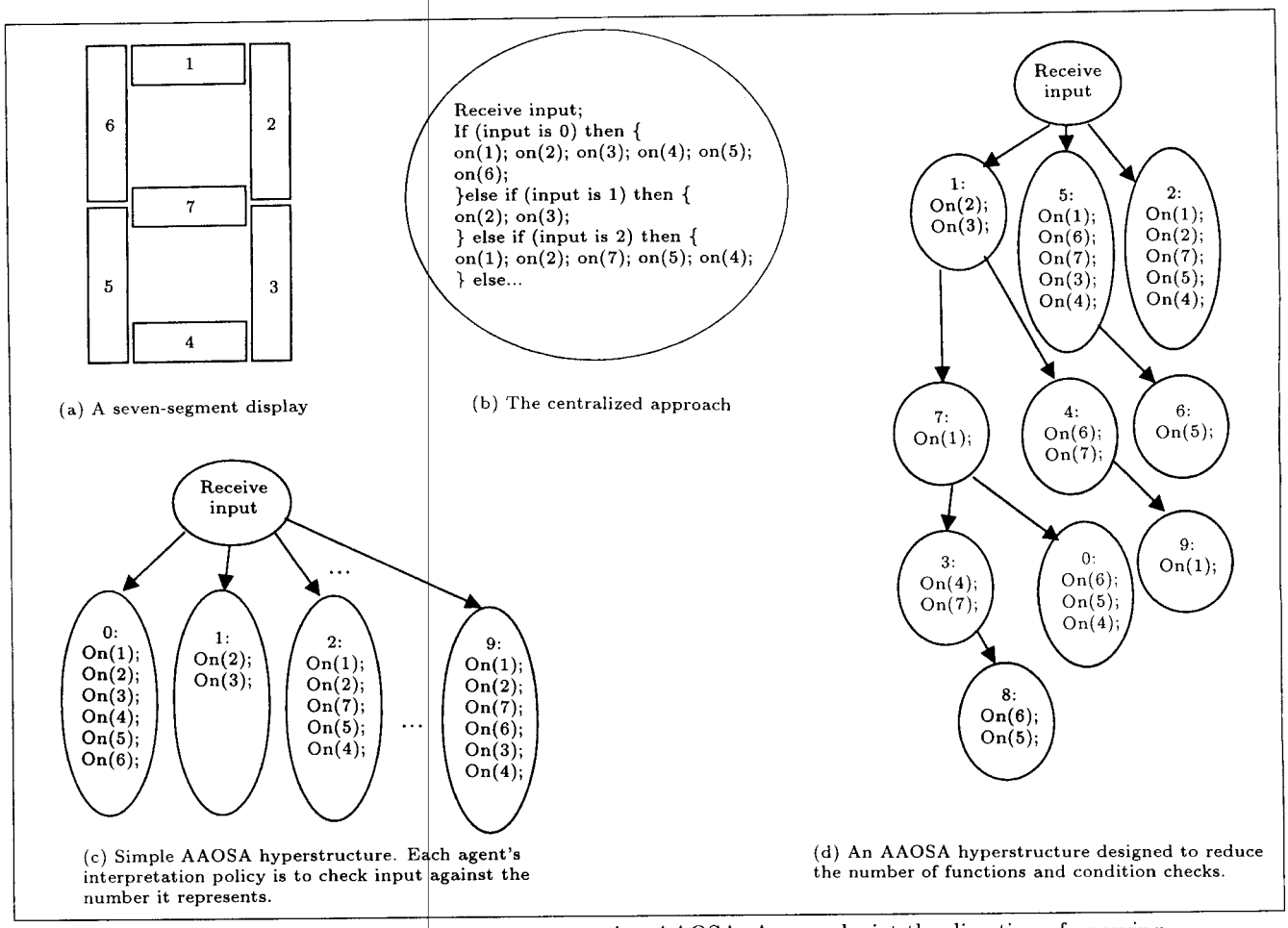

Figure 2. Designing a seven-segment system using AAOSA. Arrows depict the direction of querying.

The designer determines the level of localization of each agent. In other words, it is advisable that each agent be kept simple in its responsibilities and limited to the decisions that it must make to reap the benefits of distribution and enhance its learning abilities. The overhead of the required units (the white-box) should also be taken into consideration.

Agents can be replaced at run-time with other more complete agents. The replacement can even be a hierarchy or network of new agents breaking down the responsibilities of their predecessor. This feature provides for the incremental design and evaluation of software.

In AAOSA, the emphasis is on the distribution of capabilities. Therefore, if a capability is general enough to be coded into the white-box and distributed over all agents, it is much more desirable than assigning a specific agent to be responsible for it. For instance, using the learning module in the white-box is more desirable than creating a separate learning meta-agent.

In the following example, it is shown that the manner by which a system is agentified depends on the various objectives the designer has in mind.

Seven-Segment Example

The design of a simple application is, now, followed to observe the various advantages AAOSA may result through applying different levels of localization to agents. The system to be designed takes a number between 0 and 9 and switches on the appropriate LEDs in a seven-segment display (Figure 2a). There are, of course, tried and tested algorithms for designing this system that provide optimal results. This is mainly because the problem is a limited one and all the possible inputs and desired outputs are known.

The first step in the design of this system is identifying the range of possible input to the system and the set of output functions available. In this case, there are 10 possible inputs, namely the numbers 0 to 9. There are 7 functions which should be used to produce the overall desired output: Switch LED 1 on (or On(1) for short), On(2), On(3)... On(7). A non-modular centralized solution (Figure 2b) would involve 48 functions and 5.5 condition checks on average, assuming each number is inputted with equal probability (1/10).

An alternative to this approach would be a sys-

tem in which one agent represents each function and an input agent receives the input and distributes it (Figure 2c). If this agent (i.e., the input-agent) would have any interpretation of its own, they would be of the transitive kind, declaring an input to belong to one of the down-chain agents. However, in this example, transitive interpretations are not necessary, since the fact that the input has been handed down through the input-agent does not effect the route or process that it might be taking later. It is always preferable not to use transitive interpretations as it prevents the agents from being self-sufficient and makes the problem of maintaining the localization of agents even more difficult (Figure 3). Therefore, in the case of hyperstructure in Figure 2c, each agent will have its own interpretation policy, namely, checking its input against the number it represents.

Although the number of functions in this system is the same as the centralized system in Figure 2b, certain useful features appear due to the way modularization is conducted. Each agent is reusable in other systems and in the case of using a parallel platform, the number of conditions that may be checked on average would be much less (in a fully parallel system it would be one condition on average).

As stated previously, a system can be modeled using many different hyperstructures and the choice of the hyperstructure to be used depends on the requirements of the application. Now, the hyperstructure in Figure 2d is considered. This system is modularized based on the optimization of the number of functions, while maintaining a relatively low number of average condition checks. The total number of functions implemented here is 24 (half of the last two designs). The average condition check, if the system is taken as running on a fully parallel platform, can be calculated as follows:

1. Each possible input between 0 to 9 would occur 1/10 of the time;
2. If input were 1, 2, or 3, one condition must be checked;
3. For inputs 4, 6 and 7, two conditions would have to be checked;
4. For inputs 0, 3 and 9, the number of conditions checked would be 3;
5. For input 8, four conditions would have to be checked.

Thus, the average conditions checked would be 2.2. It is evident that for calculating this number, the conditions checked in the white-box of the agents during the query and delegation phase are disregarded. However, in general, unlike this example, the complexity of the interpretation process for each agent usually outweighs the complexity of the processes involved in these two phases. In comparison to the hyperstructure in Figure 2c, the reusability has been reduced and the average condition checks have been increased, in order to minimize the number of functions.

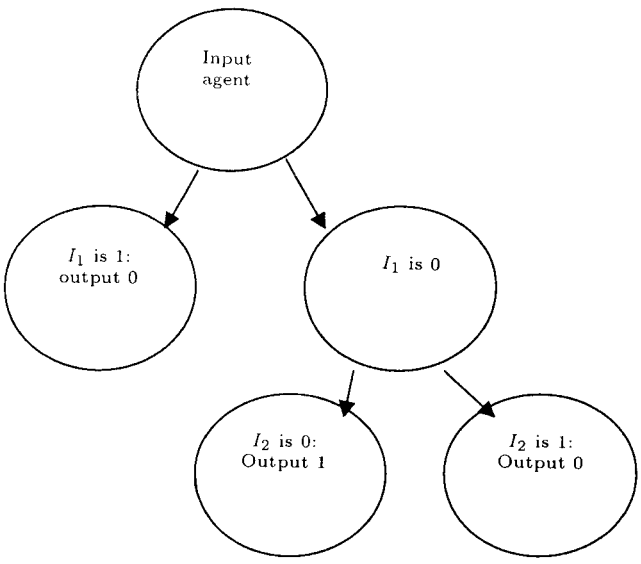

Figure 3. The NOR function using AAOSA. The input agent receives $I_1 I_2$ as input. This proves that there exists an AAOSA hyperstructure with no transitive interpretations for any computable function.

MODELING AN INTERACTION SYSTEM USING AAOSA: A MULTIMODAL MAP

Multiple input modalities may be combined to produce more natural user interfaces. To illustrate this technique, Cheyer and Julia [9] have presented a prototype map-based application for a travel-planning domain. The application is distinguished by a synergistic combination of handwriting, gesture and speech modalities; access to existing data sources including the World Wide Web and a mobile handheld interface. To implement the described application, a distributed network of heterogeneous software agents was augmented by appropriate functionality for developing synergistic multimodal applications.

A simplified subset of this example is considered here to show the differences of the two approaches. A map of an area is presented to the user and she is expected to give view port requests (e.g., shifting the map or magnification), or request information on different locations on the map. For example, a user drawing an arrow on the map may want the map to shift to one side. On the other hand, the same arrow followed by a natural language request such as: "Tell me about this hotel." may have to be interpreted differently.

Cheyer and Julia [9] have used Open Agent Architecture (OAA) [2] as a basis for their design.

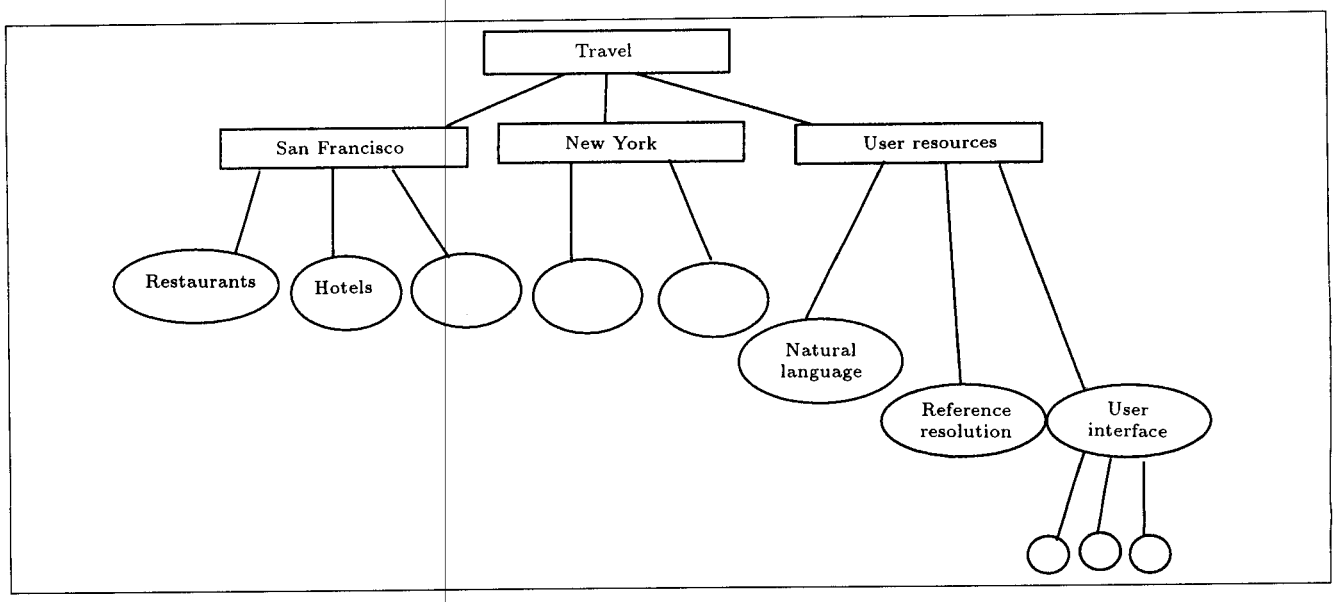

Figure 4. A structural view of the multimodal map example as designed using OAA in [9]. Boxes represent facilitators, ellipses represent macro agents and circles stand for modality agents.

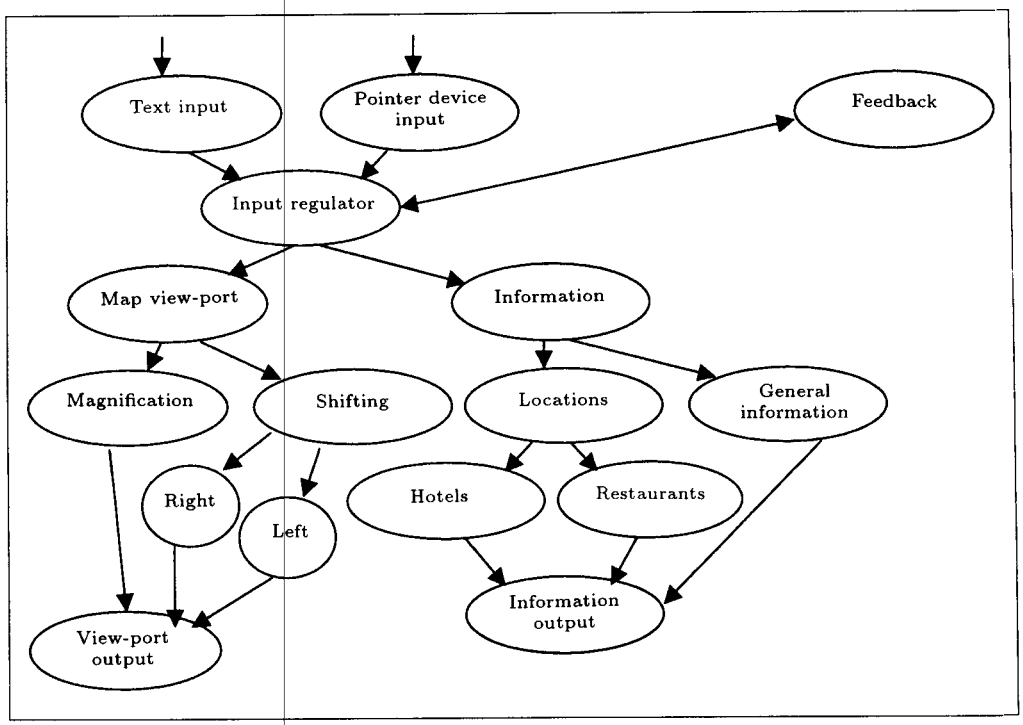

Figure 5. The multimodal map example designed based on AAOSA.

In that approach, based on a "federation architecture" [10], the software is comprised of a hierarchy of facilitators and agents. The facilitators are responsible for the coordination of the agents under them so that any agent wanting to communicate with any other agent in the system must go through a hierarchy of facilitators (starting from the one directly responsible for it). Each agent, upon introduction to the system, provides the facilitator above it with information on its capabilities (Figure 4). No explicit provision is given for learning.

An example design based on AAOSA is shown in Figure 5. It must be noted here that the design shown here is not rigid and communication paths may change through time with the agents adapting to different input requests.

The text and pointer input agents determine the end of their respective inputs and pass them on to the

input regulator. This agent in turn determines whether these requests are related or not. It then passes them down to the agent it considers more relevant. The output agents simply actuate suggestions made by shifting, magnification, hotels, restaurants and general information agents. Note that the combination of these output suggestions could also be chosen for actuation. The feedback agent provides the system with rewards interpreted from user input.

Some of the differences in the two designs are given below:

- The AAOSA design is much more distributed and modular by nature and many of the processes concentrated in the facilitator agents in Figure 4 are partitioned and simplified in Figure 5;
- AAOSA is more of a network or hyperstructure [11] of process modules as opposed to the hierarchical tree-like architecture in the OAA design;
- AI behavior such as input interpretation (e.g., natural language processing) and machine learning are incorporated on the architecture and distributed over the multi-agent structure rather than introduced as single new agents (as is the case with the natural language macro agent in Figure 4).

It must be stressed that AAOSA-like architectures could be obtained with an OAA if each OAA facilitator and its macro agents be considered as one agent and add learning capabilities to each facilitator. Another point worth mentioning is that agents in OAA are usually pre-programmed applications linked together through facilitators. The designers have a fewer options regarding the software architecture as a whole because they are forced to use what has already been designed, possibly without the new higher-level framework in mind.

In designing an interactive system using AAOSA, an agent has been assigned to each individual function of the system (e.g., Magnification, Shifting, Hotel information, Restaurant Information, General Information). The functionality of these agents is implemented in the black box of each agent. These agents are also responsible for maintaining a representation of their respective domain. For instance, the magnification agent maintains a variable representing the current degree of magnification.

The structure of agents that lead to these leaf agents represents the designer's view of the system hierarchy. These agents usually have a much simpler black box. Their role is mainly to direct requests to the appropriate agents and learn or resolve contradictions that may occur at their juncture.

AAOSA Interaction System's Communication Performatives

In this section, the common performatives used by the agents to communicate are described. These performatives are all general and, therefore, pre-implemented in the white box modules of these AAOSA agents.

Register

Agents need to register themselves with each other to be able to send messages to one another. Unlike similar systems, in AAOSA, it is not necessary that all agents be aware of all other agents and registration may be much more distributed and localized. For instance, the information agent only needs to register with the location, general information and input regulator agents. Each agent, upon receiving a register message, adds the registering agent's name and address in its address book. Registration may take place at run time.

Advertise and Un-Advertise

Agents advertise their responsibilities in order to draw requests from other agents. When an agent receives an advertise message, it updates its interpretation policy so as to redirect certain requests to the advertising agent. This information is removed from the interpretation policy once an un-advertise message is received. An advertise message specifies a community to which the advertising agent belongs. If the agent receiving this message does not recognize such a community in its interpretation policy, it may add it as a new community. In the multimodal map example, shifting agent advertises "shifting" to map view-port agent which in turn creates a new community by that name in its interpretation policy. This allows for more than one agent being member of an interpretation community of another agent.

Now, a sample run of the multimodal map example is followed.

THIS-IS-YOURS

When an agent is successful in interpreting a request, it must pass it over to an agent from the interpreted community. The performative under which this request is forwarded is called THIS-IS-YOURS. The receiving agent knows that if it cannot interpret this request, then the point of contradiction is itself. For example, consider that the input regulator agent receives "Map to the right" from the text input agent. If the interpretation policy for routing requests to the map view-port community is simply the presence of the word "map" in the requests, this request is sent to an agent in that community. In the example presented here, only one agent exists per community so THIS-IS-YOURS

message with the content request will be sent to the map view-port agent.

IS-THIS-YOURS?, IT-IS-MINE and NOT-MINE

For an agent-based interactive system, adaptation occurs to accommodate user preferences, whilst the conditions that the task must satisfy are poorly defined; thus, necessitating a search to find a suitable solution [12]. As mentioned earlier, the interpretation of input is distributed over the structure of agents so there might be situations in which an agent cannot directly interpret a request and will have to query the communities it knows and wait for their response. Each agent itself may query agents belonging to its communities. In the example here, using the same simplistic interpretation method (i.e., presence or absence of key words), when the map view-port agent receives "Map to the right", it cannot interpret it and so it will query the communities it knows by sending them IS-THIS-YOURS? messages. It will, then, wait until it receives responses from all agents it has queried. The magnification agent, not having any communities, will respond with NOT-MINE. The shifting agent, having interpreted the request successfully (because of the presence of "right" in the request string), responds with IT-IS-MINE. At this point, the map view-port agent sends the query down to the shifting agent using the THIS-IS-YOURS performative.

CONTRADICTIONS OR AMBIGUITIES

An important issue is how to combine the opinions of different agents. Generally, certain confidence factors are associated with each decision and, then, some method is used to generate the final decision on the basis of the individual decisions. If an agent is capable of dealing with a subset of given problems, and if this agent can be considered "sufficiently reliable", there is no need to worry about redundancy. In this case, the answer of that one agent is sufficient [13].

Contradictions occur in the following cases:

- When an agent that has received a THIS-IS-YOURS message cannot interpret it and all of its communities upon being queried respond with NOT-MINE messages,
- When an agent that has received a THIS-IS-YOURS message cannot interpret it and more than one of its communities upon being queried responds with an IT-IS-MINE message,
- When the user expresses dissatisfaction with a response from the system.

In the implementation of the interpretation system, a combination of the following methods have been used to resolve such contradictions:

- Using priorities or weights for agents of different levels in the hyperstructure. For instance in a contradiction that has occurred in the input regulator agent, the IT-IS-MINE that originates in the locations agent will have a higher priority than the one originating in the left agent;
- Using the recency of agent invocations can cause a context switching capability in the system;
- Determining the regions of input that have triggered agents can be a good contradiction resolution policy. If these agents are mutually exclusive, multiple agents may be required to handle a single request;
- Querying the user directly and adjusting the interpretation policy accordingly (i.e., learning). Note that in this case, the interaction is limited to the point of contradiction.

Contradiction plays an important role in learning and pinpointing the agent which is responsible for a contradiction and resolving it, insuring the correct distribution of responsibilities in the hyperstructure.

DISTRIBUTED LEARNING

The problems regarding multi-agent learning have been extensively ignored. Designing agents that would learn about anything in the world is against the basic philosophy of distributed AI. This issue has not really received considerable attention, which might be the reason behind the ill-behavior of some systems (the more they learn, the slower they perform) [13]. Through allowing the agents to adapt, refine and improve, automatically or under user control, a holistic system can be created in which the whole is significantly more than the sum of its parts [12].

The combination of machine learning and multi-agent systems can have benefits for both. Multi-agent systems having learning capabilities reduce cost, time and resources and increase quality in a number of forms [13]:

- Ease of programming,
- Ease of maintenance,
- Widened scope of application,
- Efficiency,
- Coordination of activity.

On the other hand, machine learning in a multi-agent set-up becomes faster and more robust [13].

Adaptability in AAOSA materializes in the following forms:

- The ability of the system to accept new agents at run time,

- The ability of each agent to adapt its behavior according to the feedback it receives (i.e., learning).

A sample of the program run and the contradiction resolution process is described in Figure 4. The learning algorithm in the simplest form could be a memorization of the user response. This method lacks generalization and is context independent. In other words, the interpretation cannot be modified to include the previous state of the system.

In step 3 of Figure 6, a contradiction is resolved based on the user response. In this case, "move it closer" is considered to fall into the map view-port domain and, therefore, the user response is learned by this agent. In the example of Figure 7, however, the request's domain is simply not identifiable. The system learns the appropriate response to grow through its interactions with the user, but "grow", which seems to fall into the map view-port agent's domain, has also been learned by the input regulator agent (step 3a of Figure 7).

Learning can be applied to AAOSA in a number of ways depending on the objectives and application of the software:

- Inside the agents: In large and complex software, distributing the learning over a hyperstructure of more simple sub-domains is less complex than centralized learning. Learning can be used to improve the agent's own specialized performance and to improve its interpretation policy to reduce ambiguities. This latter form of learning is driven by the ambiguities themselves. There are various machine learning algorithms that can be used in the learning module of the white-box, sometimes in combination. For instance, reinforcement learning can be used to fine-tune the choice of relevant interpretation rules, while rule learning algorithms add or update them. The former being more gradual and statistic based while the latter changes the agent behavior in quantum leaps and is based on a comparison of the actual interpretation with the desired one;
- Over the architecture (Dynamic AAOSA): Evolutionary and statistical learning can be used to split agents that are more complex into hyperstructures of simpler ones, or join redundant agents to form more efficient ones. This brings about the possibility of hyperstructures self-organizing themselves to achieve a balance between the degree of distribution and the efficiency of the overall software.

AAOSA should maintain the localization of each agent. In other words, AAOSA should guarantee that:

1. The agents each stay responsible for the limited domain they were originally assigned to, while:
2. Containing the distribution of responsibilities, thus:
3. Demonstrating the simplicity of each component through adaptive change or developmental upgrades.

Therefore, learning should guarantee the balance of distribution and learning methods should not impede each other. For instance, when a new interpretation rule is learned by a down-chain agent, "A", it may have to send Un-Learn messages to all up-chain agents requesting them to remove any identical rule that results in delegation of input to agent "A".

5) User inputs "move it closer"

a) "move it closer" is sent to the input regulator agent with a "This-Is-Yours" performative.
b) The input regulator agent in turn sends "move it closer" down to the information community and the map view-port community with an "Is-This-Yours?" performative.
c) Agents in the communities beneath the information all respond with "Not-Mine" and therefore the information agent also sends a "Not-Mine" message to the input regulator agent.
d) The magnification agent and the shifting agent both claim "move it closer" to be theirs by sending "It-Is-Mine" messages to the map view-port agent. Simple word spotting techniques may be used for the interpretation of each agent: There is a "move " in the request so it may belong to the shifting agent. There is a "closer" in the request so it may belong to the magnification agent.
e) The map view-port agent announces a contradiction by sending a "Maybe-Mine" message to the input regulator agent.
f) The input regulator agent sends a message to the map view-port agent asking it to "resolve" its contradiction because it has not had any other positive responses from agents with a higher priority.
g) The map view-port agent interacts with the user to resolve the contradiction:

6) System Asks User: "Do you mean magnification or shifting?"
7) User responds with "Magnification"

a) The map view-port agent learns that "move it closer" belongs to the magnification agent and resolves the contradiction.
b) The map view-port agent sends "move it closer" to the magnification agent with "This-Is-Yours".
c) The magnification agent interprets "move it closer" and sends a "This-Is-Yours" performative with "Zoom in" as the content to the view-port output agent.

8) View-port is magnified

Figure 6. A sample run of the multimodal map example.

6) User inputs "grow"

a) "grow" is sent to the input regulator agent with a "This-Is-Yours" performative.
b) The input regulator agent in turn sends "move it closer" down to the information community and the map view-port community with an "Is-This-Yours?" performative.
c) Agents in the communities beneath the information agent all respond with "Not-Mine" and therefore the information agent also sends a "Not-Mine" message to the input regulator agent.
d) Agents in the communities beneath the map view-port agent also respond with "Not-Mine" and therefore the map view-port agent also sends a "Not-Mine" message to the input regulator agent. It is assumed that the magnification agent does not recognize "grow" as an interpretation keyword.
e) The input regulator agent interacts with the user to resolve the contradiction:

7) System Asks User: "Do you mean map view-port or information?"
8) User Responds With "map view-port"

a) The input regulator agent learns that "grow" belongs to the map view-port agent and resolves the contradiction.
b) The input agent sends "grow" to the map view-port agent with "This-Is-Yours".
c) The map view-port agent has not been able to interpret "grow" before (this can be checked by looking up a temporary memory of interpretations done for this request).
d) The map view-port agent cannot interpret the request to decide between magnification or shifting and so sends "grow" down to the shifting community with an "Is-This-Yours?" performative.
e) Agents in the communities beneath the shifting agent all respond with "Not-Mine" and therefore the shifting agent also sends a "Not-Mine" message to the map view-port agent.
f) The map view-port agent interacts with the user to resolve the contradiction:

9) System Asks User: "Do you mean magnification or shifting?"
10) User Responds With "magnification"

a) The map view-port agent learns that "grow" belongs to the magnification agent and resolves the contradiction.
b) The map view-port agent sends an "Un-Learn" to all calling agents for "grow".
c) The map view-port agent sends "grow" to the magnification agent with "This-Is-Yours".
d) The magnification agent interprets "grow" and sends a "This-Is-Yours" performative with "zoom in" as the content to the view-port output agent.

Figure 7. Example of domain localization in AAOSA.

Learning can be deployed to automate disambiguation, and/or resolve conflicts between interpretation rules in a single agent. The latter case occurs when a single agent has rules that may result in conflicting interpretations based on similar decision criteria. In these cases, weighting the rules based on past experience and utilizing this weight when making a choice between rules that apply to a certain input is a form of learning.

The learning used in the current version of AAOSA, on the other hand, is a very simple rote-learning algorithm that records interpretation results for ambiguities explicitly disambiguated for the agent by the user. This learning happens implicitly and, therefore, it is very important that the learning be as conservative as possible. For instance, learning will only occur when a single contiguous focus of the input string is identified as unencountered rule invocation criteria for the agent's policies. As it is will be seen, this learning algorithm is sufficient in the interactive natural language interface application. In other cases where implicit statistical (history-based) disambiguation is used more often, the learning algorithm will also have to be more complex. In these cases, reinforcement learning methods could be used.

UN-LEARN

To resolve the domain problem and ensure localization, a new performative, UN-LEARN, is added. Agents that learn a new interpretation rule, send the rule to all agents they know with the UN-LEARN performative. The agents receiving this message from a certain agent will check to see if they are using the same rule to send requests to this agent. If this is the case, they will remove this rule from their interpretation policy. For example step 5b of Figure 7 will cause the Input Regulator agent delete the rule telling it to redirect requests containing the "grow" keyword to the map view-port community.

This change may seem to make the system less efficient because the agents will have to communicate more often to be able to resolve input requests. However, keeping the agents from learning local domain information belonging to other agents will guarantee the intended distribution of responsibilities according to user interactions. This will prevent the upward or downward drift of responsibilities in the agent hierarchy, which may cause certain agents to become overloaded while others are left idle.

EVALUATION AND CONCLUSION

Proof that AAOSA can be used to parse any context sensitive language is provided in [14], showing the potential power of the interpretation phase of an AAOSA application and its application in Natural Language Processing.

AAOSA has been evaluated in several large scale

applications, namely, as an interface to a home A/V system, an e-mailing and contact application interface, a database retrieval application and a car entertainment interface in which more than 150 agents where used. User testing has been conducted on all of these applications to generate test corpus. The size of the corpus for each test was determined based on the estimated number of functions involved and at its largest (for the car interface) included 2000 entries. Each entry may result in one or more function calls and/or ambiguity resolution interactions, for which average rates of more than 90% have been accomplished. Each of these applications was developed by teams of up to three engineers that spent no more than 45 days for it.

Viewing software as a hyperstructure of agents (i.e., intelligent beings) results in designs that are much different in structure and modularization. Some of the benefits of this approach are noted here.

- Flexibility: There is no rigid predetermination of valid input requests. An AAOSA application attempts to map any possible input to the functionality of its agents, thus input is not limited;
- Parallelism: The independent nature of the agents creates a potentially parallel design approach. Current implementations of AAOSA can be executed in a distributed environment. The message driven communications between the agents make this distributed execution possible;
- Multi-platform execution: Agents can run and communicate over networks of computers (on the internet for instance). This is, again, a result of having agents only communicate through messages. The implementation platform of the AAOSA agents is not important as long as the messaging protocols are observed. The current version of AAOSA is coded in Java language, which furthers this possibility;
- Runtime addition of new agents and, thus, incremental development of software is possible. Agents introduce themselves to each other, thus, forming the hyperstructure at runtime. Consequently, the "capabilities" of each up-chain agent can be modified and changed upon the introduction of new down-chain agents at runtime;
- Reusability of agents due to the increased modularity brought about by the agentification of modules. This type of modularization can also increase encapsulation and provide for mobility of modules, and the incremental design and evaluation of an AAOSA application;
- Learning and intelligence: The distributed nature of learning introduced in this paper suggests a powerful adaptive software design that potentially breaks down an application to a hyperstructure of simple learning modules [11].

FUTURE WORK

Although the use of AAOSA architecture in the area of interactive systems seems to be promising, this methodology is still at its infancy and, therefore, should be tested for its scope of applicability. Two main areas in AAOSA have the potential for greater improvement:

- Each agent's learning algorithm can be improved immensely. Even so, in simple implementations the fact that the learning was distributed resulted in an effective system. Improvements could be made to provide for:
  1. Generalization,
  2. Context sensitivity,
  3. Acquisition of undefined concepts [15].
- Using AAOSA, it was possible to successfully localize the responsibilities of a society of agents collaborating in an interactive application. This localization is vital if usage of distributed learning, effectively, is intended because it helps in limiting the complexity each agent has to deal with to the pre-designed scope for that agent. The problem here is that the success of this localization depends on designers. One improvement to the current system is to have the initial design evolve and optimize at run time. This optimization could be done using evaluational feedback from different sources that give an estimate for the complexity the different agents should deal with. The estimate can be made using:
  1. Explicitly from the user,
  2. Implicitly from the user (e.g., the feedback agent can interpret user actions),
  3. From the rate of contradictions occurring in different agents,
  4. Time or resources.

According to the feedback, new agents could be added to the system once the points of highest complexity are found. The ultimate goal would be, in this regard, the capability of agents in splitting their responsibilities and knowledge (like a single cell organism dividing). On the other hand, using evolutionary methods, unwanted and redundant agents can be removed, passing their responsibilities to other agents.

REFERENCES

1. Franklin, S. and Graesser, A. "Is it an agent or just a program? A taxonomy for autonomous agents", in *Proc. of the Third International Workshop on Agents Theories, Architectures, and Languages*, Springer-Verlag (1996).
2. Cohen, P.R., Cheyer, A., Wang, M. and Baeg, S.C. "OAA: An open agent architecture", *AAAI Spring Symposium* (1994).

3. Martin, D. and Moran, D. "Building distributed software systems with the open agent architecture", *Proc. of the Third International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, The Practical Application Company Ltd., Blackpool, Lancashire, UK (March 1998).
4. Hayes-Roth, B., Pfleger, K., Lalanda, P., Morignot, P. and Balabanovic, M. "A domain-specific software architecture for adaptive intelligent systems", *IEEE Transactions on Software Engineering* (April 1995).
5. Shoham, Y. "Agent-oriented programming", *Artificial Intelligence*, **60**(1), pp 51-92 (1993).
6. Genesereth, M.R. and Ketchpel, S.P., *Software Agents*, *Communications of the ACM*, **37**(7) (July 1994).
7. Hodjat, B., Savoie, C.J. and Amamiya, M. "Adaptive agent oriented software architecture", due for presentation in *the Pacific Rim International Conference on Artificial Intelligence (PRICAI 98)* (Nov. 1998).
8. Cockburn, D. and Jennings, N.R. "ARCHON: A distributed artificial intelligence system for industrial applications", *Foundations of Distributed Artificial Intelligence*, G.M.P. O'Hare, N.R. Jennings, Eds., pp 319-344, John Wiley & Sons (1996).
9. Cheyer, A. and Julia, L., *Multimodal Maps: An Agent-Based Approach*, http://www.ai.sri.com/cheyer/papers/mmap/mmap.html (1996).
10. Khedro, T. and Genesereth, M. "The federation architecture for interoperable agent-based concurrent engineering systems", in *International Journal on Concurrent Engineering, Research and Applications*, **2**, pp 125-131 (1994).
11. Hodjat, B. and Amamiya, M., *The Self-Organizing Symbiotic Agent*, http://www_al.is.kyushu-u.ac.jp/bobby/1stpaper.htm (1998).
12. Beale, R. and Wood, A. "Agent-based interaction", *Proc. of HCI'94*, Glasgow, pp 239-245 (1995).
13. Brazdil, P., Gams, M., Sian, S., Torgo, L. and van de Velde, W., *Learning in Distributed Systems and Multi-Agent Environments*, http://www.ncc.up.pt/ltorgo/Papers/LDSME/LDSME-Contents.html
14. Hodjat, B. and Amamiya, M. "Applying the adaptive agent oriented software architecture to the parsing of context sensitive grammars", in *Proc. of FOSE'99*, Sapporo, Japan (1999).
15. Brazdil, P. and Muggleton, S. "Learning to relate terms in multiple agent environment", *Proc. of Machine Learning-EWSL-91*, pp 424-439, Springer-Verlag (1991).