The Adaptive Agent Oriented Software Architecture and its Application in Natural Language User Interfaces

Babak Hodjat **Makoto Amamiya**

Kyushu University
Department of Intelligent Systems
Graduate School of Information Science and Electrical Engineering
6-1 Kasugakoen, Kasuga-shi, Fukuoka 816, Japan
+1 650 969 4930
babak@dejima.com

ABSTRACT
Adaptive Agent Oriented Software Architecture (AAOSA) is a new approach to software design based on an agent-oriented architecture. In this approach, agents are considered adaptively communicating modules divided into a "white box" module, which is responsible for communications and learning and a "black box" which is responsible for the independent specialized processes. An AAOSA parser can parse context sensitive languages. AAOSA attempts to solve the complex problem of interaction in a Natural Language User Interface by breaking up the lingual problem space into nodes of a semantic hyperstructure, which, at its lowest order, is the simple set of input functions. The use of this methodology in designing user interfaces helps overcome many human-machine interface problems by limiting the domain of language processing to the functional domain of the application. Preliminary test results described at the end of the paper show the promising potential of the method.

Keywords
Distributed Artificial Intelligence, Agent oriented Architectures, User Interfaces, Natural Language Processing.

1 INTRODUCTION

Agent abstraction is a natural extension of object-oriented technology, encapsulating the agent's knowledge within an active process and providing a standard interface for communication. The concept of large ensembles of semi-autonomous intelligent agents working together is emerging as an important model for building the next generation of sophisticated software applications ([1], [13]).

An important difference between agents in an agent-oriented system and objects is that agents contain predefined structures and functionality that gives them the ability to communicate. In many cases, this commonality is extended to include such processes as learning and planning. Thus, although the environment and responsibilities of different agents in an agent-oriented system may be different they can still have much in common. In the Adaptive Agent Oriented Software Architecture (AAOSA) paradigm we encourage the exploitation of this feature as much as possible so that the designer of an AAOSA based system is faced with as simple a task as possible.

AAOSA is a software methodology that proposes the break-up of complex software into a community of simpler, independent, collaborating, adaptive, message-driven components (*AAOSA Agents*). The goal of AAOSA is to provide software designers with the necessary coordination amongst AAOSA agents representing sub-domains of the software being developed in order to better meet the needs of the entire application. This coordination is provided through pre-defined messaging schemes between AAOSA agents.

We divide an agent into a *white box*, which contains standard data structures and methods for communications, interpretations, and learning provided by AAOSA, and a *black box*, which is defined by the designer and contains the agent specific communications, interpretations, and processes (Figure 1). AAOSA, being object oriented in design, allows the black box to override, inherit from, or change any module in the white box (*i.e.,* the data structures and methods in the white box are inherited within the black box, which can therefore access and modify them). In [2] and [3], Baas shows that a problem is cov-

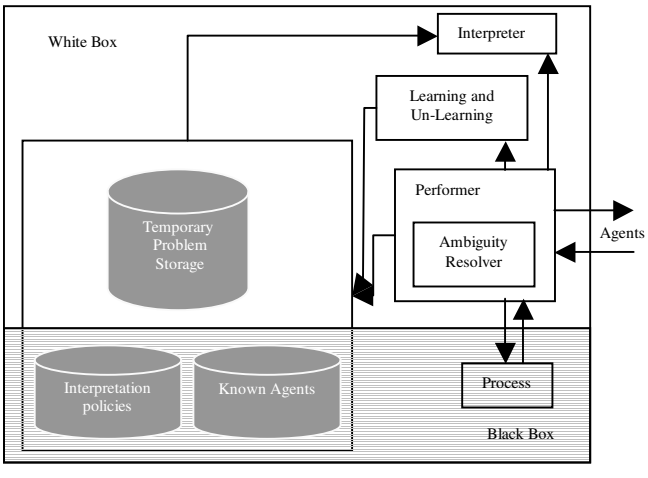


Figure 1) Each agent is comprised of a black box section (specialties) and a white box section (communications).

ered by a *hyper-structure* of computing elements (Figure 2). We propose the representation of each level of a given hyper-structure by AAOSA agents.

The designer of an AAOSA application will:

1. Break down the software to its manageable sub-domain elements (*i.e.,* AAOSA Agents),
2. Define which agents will be in direct communication with each other. These direct links are important because they concretize the designer's view of the different hyper-structure levels. The higher the level, the higher priority an agent will have in interpreting and processing input.
3. Devise interpretation policies for each agent by considering the input to the application from each agent's point of view to decide if this agent is responsible for processing all or parts of this input. An agent claims a particular input if it, or one or a number of agents down-chain to it are responsible for processing that input. In other words an agent claiming an input would, upon delegation, either process that input, or delegate it down-chain to one or a number of agents that had in turn claimed that input, or both.

AAOSA, through predefined communication schemes, should pin point the agent or agents in this hyper-structure that are responsible for processing a certain input, and provide the necessary coordination between them in order to achieve desired output. Our hope is that in this way the designer will have to deal mostly with the breaking up and design of the software elements themselves rather than the complexities of how to coordinate them. Therefore another important difference between AAOSA agents and objects in the Object Oriented methodology is that an agent does not have to know which agents are responsible for a certain process or data structure, or that process' invocation details ([4]).

By taking each software module to be an agent we can take advantage of a number of desirable features which we will impose on the designer as definitions of AAOSA agents: Each agent should be independent of the others and the only means of communication is messaging, handled by the white-box. This will provide for the possibility of parallel, distributed, and even mobile modules. Agents can be processing several requests at the same time.

No centralized control is enforced over the resulting network of AAOSA agents covering the scope of a software application. In this architecture, agents introduce themselves and their abilities to one another at the beginning or during execution. Agents can therefore be added to or removed from the application at runtime. This is one of the major differences between AAOSA and its precursor the Open Agent Architecture (OAA, [5]).

Another close relative of AAOSA is ARCHON (architecture for cooperative heterogeneous on-line systems, [6]). Unlike ARCHON though, the predefined portion of each agent (i.e., the white box) does not have to maintain a model for the designer defined portion (i.e., the black box) or any other agent it communicates with.

By now it may be apparent to the reader that the kind of agents we are characterizing in this paper, having taken a bottom-up view of *multi*-agent technology, can be quite fine grained. Unlike the AI sense of multi-*agency* ([7], [8]), AAOSA agents are not centralized human-like agents with potentially conflicting intentions working together. Rather they are distributed software-object-like agents designed to work together cooperatively to implement complex applications. The Actor model [13] is a close DAI relative and perhaps suitable as a platform to implement AAOSA agents but too fine-grain and low level to replace it.

In this paper, we will introduce the AAOSA architecture and some of its applications. An AAOSA system is actually parsing input in its interpretation phase. By examining the capabilities of an AAOSA parser, we will be able to give concrete evidence of the power of the AAOSA methodology. We have shown that the AAOSA based parser can parse context-sensitive grammars with reasonable complexity. The main application of AAOSA has so far been in natural language user interfaces. We will discuss the differences between natural language processing and grammatical processing of languages and discuss the application of a more robust design for natural language user interfaces. The paper ends with an outline of what has been done and what lies ahead in this area including a brief description of some of the commercial Natural Language User Interface (NLUI) applications of AAOSA and some test results.

2 AAOSA AGENTS AND COORDINATION

Processing of the input is done in two main phases. The interpretation phase, in which the agent, or agents responsible for actuating an input are located, and a delegation phase in which the processes that have been located are called.

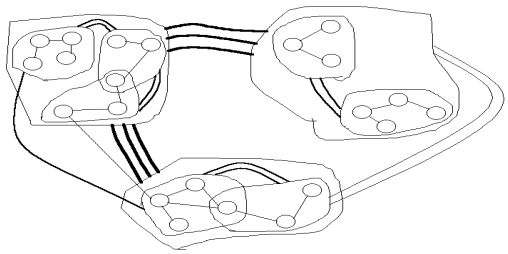


Figure 2) Example of third-order interaction graph of a hyper-structure allowing cumulative interactions and overlapping aggregates. Circles represent *first order hyper-structures* or computing modules of the lowest complexity in the given domain.

Each AAOSA agent must be able to interpret input sent to it as the content field of messages from other agents if so requested in the performative field of that message (Table 1). The result of this interpretation may cause the agent to claim that input as its own and/or to declare certain other agent/s responsible for processing it. Agents may consult other agents in order to complete their claims. These latter agents we will call down-chain agents relative to the requesting agent. This is a relative term and depending on the direction of the flow of requests, an agent may be *down-chain* (receiving) or *up-chain* (requesting) with respect to another agent.

Unique Query ID
Sender Agent
Message content (input to AAOSA Agent)
Performative
Priority or sender agent's hyperstructure degree
Claim made by agent about the content
Depth of search or allowable distance from initiator of query
Depth so far, or distance query has been propagated down-chain
History of agents that have processed this input before

Table 1) AAOSA standard Inter-agent message fields.

Agents that are first to receive input to the AAOSA system are called *input agents*. These agents initiate the interpretation phase for that input and are the entry points to the system, generating unique query IDs for new input. This does not mean that other agents do not query input agents. Cycles are avoided by preventing the agents from repeating processes already executed over the same query. Input agents are also responsible for announcing the end of the processing of a specific query to all down-chain agents.

Input may also have been generated inside the system and therefore any agent could potentially be an input agent. In the simplest form, a claim means all of input belongs to the agent making the claim. In many cases, as we shall see in the examples, a claim should contain other information as well (e.g., confidence in claim, name of claiming agent or symbol representing various claims one agent can make, the level of the claiming agent relative to the input agent of this query, parts of input that is being claimed).

The software designer is responsible for providing each agent with its interpretation policy. An *interpretation policy* is comprised of a set of rules used to decide to return a claim that a piece of the input belongs to that particular agent. The interpretation criterion may be the message content but is not limited to it. Process history, probabilities and outside information (e.g., interaction with other agents) are examples of some of the other parameters that may be used by the interpretation policy. Note that interpretations do not determine whether a particular input does not belong to the agent. Determining whether an input does not fall into the scope of responsibilities of an agent, as well as whether it does, amounts to modeling the world ($\mathbf{W=P \cup \sim P}$) and undermines the distributed nature of AAOSA agents. Therefore, the application of interpretation policies to input either result in a successful interpretation or a "don't-know" state.

The performer module in the white box actuates other modules in the agent based on the message performatives received from other agents. Each message is comprised of a message content, and a performative that specifies what should be done with that content ([9]). No overall standard data representation is needed for the message content. Agent specific data can be transferred in messages in whatever format the sender and receiver agree upon. The designer can add agent-specific performatives in subdomains to facilitate special communications between agents. AAOSA provides a set of predefined general performatives by which the coordination of agents is managed (Table 2).

Register	Agents make each other aware of their existence.
Advertise	An agent declares it can handle certain input.
Un-Advertise	An agent requests not to receive input from another agent.
This-Is-Yours	An agent announces another agent as responsible for handling certain input.
Is-This-Yours?	An agent that can not interpret a particular input requests interpretation from down-chain agents.
Restore	Agent requests another agent to backtrack to a state before processing took place on certain input.
Not-Mine	Down-chain agent has failed to interpret input sent down with an *Is-This-Yours?* Performative.
Maybe-Mine	Down-chain agent has encountered an ambiguity in interpreting input sent down with an *Is-This-Yours?* Performative.
It-Is-Mine	Down-chain agent has been successful in interpreting input sent down with an *Is-This-Yours?* Performative.
Commit	Agent requests immediate response, be it incomplete, to input sent down with an *Is-This-Yours?* Performative.
Learn	A new interpretation policy is suggested to an agent that will result in the sender agent being interpreted as responsible for certain input.
Un-Learn	An interpretation policy that results in the sender agent being interpreted as responsible for certain input is revoked.
Dissatisfied	Alternative process or interpretation is requested for input that has already been processed.
Forget-Problem	A previous request is canceled and the receiving agent will remove any temporary storage of interpretation results.

Table 2) Some AAOSA predefined inter-agent message performatives.

In the interpretation phase, each agent, upon receiving input with an "*Is-This-Yours?*" performative, attempts to interpret the input by itself. If interpretation is successful, the agent will report claims using the "*It-Is-Mine*" performative. As we shall see, this does not always mean that this agent will be assigned to do its processing of the input.

On the other hand, if an agent can not interpret the input as its own, before reporting failure, it must check with other down-chain agents. If all down-chain agents report "*Not-Mine*", this agent will also report "*Not-Mine*" to its requesting agent. If at least one down-chain agent is able to interpret the input successfully and reports back with "*It-Is-Mine*", our agent will also report success. It follows that agents that have no down-chain agents to query may report *"Not-Mine"* upon failure to find an interpretation policy that applies to the contents of the query message

they have received. To prevent agents from repeatedly processing the same queries in a cycle, each agent keeps track of queries it has processed and will reply *"Not-Mine"* to any query it has already responded to and has no new claims for.

After a path of down-chain links from a top-level agent to some agent or agents responsible for processing input is found (using the "*Is-This-Yours?*" performative), the delegation phase can start. In this phase, the "*This-Is-Yours*" performative is used to call agents on these paths to do the actual processing. Agents receiving a "*This-Is-Yours*" request may reinterpret the delegated input, or they may use pre-stored interpretation or down-chain query results to, in turn, process or delegate (or both) the input or parts of it.

Ambiguities of which agent owns a particular piece of input, and methods for resolving them, are central to the proper operation of AAOSA. Ambiguities occur when an agent that a job has been delegated to (i.e., has received a message with the *This-Is-Yours* performative) has not been able to interpret the message content as belonging to it based on it's interpretation policies, and

- Either more than one down-chain agent that was consulted with claim it, or,
- None of the agents consulted with claim it.

An ambiguity can be resolved by explicit interaction with another agent (e.g., an agent representing the human user). This is not always desirable or possible and therefore implicit resolution methods must be used. Table 3 describes methods that can be used to resolve ambiguities in AAOSA. The choice of ambiguity resolution methods and the way they are combined to achieve best results depend on the application in which they will be used.

Ambiguity and its resolution is particularly important in AAOSA because it provides a means by which agents can change their behavior (i.e., learn) and react to unexpected input.

As said before, an agent that does not have a suitable interpretation for input contents of a message sent to it with the "*Is-This-Yours?*" performative will propagate this message to it's down-chain agents. By suitable interpretation, we imply that in cases where the interpretation policy uses a small part of the whole input as its decision making focus, the agent may decide to query down-chain agents on the remainder of the input anyway, so as to make more accurate claims. In other words, agents complete their claims after receiving the results of their queries to their respective down-chain agents. Hence, there may even be cases in which an agent that has successfully interpreted parts of the input decides to query its own down-chain agents.

Even if this was not allowed, the problem of query propagation should be addressed by AAOSA: When do we decide to abandon a query or stop propagating it down-chain?

This depends very much on the application. In cases where the depth of propagation is not that much there may

Priorities		Claims from agents representing higher degrees in the hyperstructure may have priority over the rest.
Context	***Recency***	Agent that has claimed input most recently is more likely to be responsible for processing disputed input.
	Status	Current status of a data-structure an agent is responsible for may make it more eligible for claiming disputed input.
Focus and Focal point		Agents basing their claims upon a larger portion of the input (i.e., focus), or parts of the input closer to the requesting agent's focus (Focal Point) are more likely to be responsible for disputed input. Agents claiming mutually exclusive input may all be responsible at the same time.
Statistics and Probabilities		More successful agents are chosen to process disputed input based on their prior performance history.
Interaction		A dispute is settled by referring to another agent.

Table 3) Ambiguity resolution methods.

be no need for stopping it. In other cases, such as interactive applications, in which response time is important, time elapsed since first agent received input from user may be used to issue a message with the "*Commit*" performative. This performative will cause receiving agents to abandon any query response not received and act upon the information they have, be it incomplete. Another approach would be to time-stamp requests at origin so each agent can reject requests older than the allowable overall response time.

3 DESIGNING AAOSA-BASED APPLICATIONS

The AAOSA design methodology is essentially a bottom-up approach: The tasks necessary to achieve overall goals are identified and suitably decomposed ([6]). Then the data-flow between these tasks is determined. This way, pre-existing code 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 to avoid claiming input that really belongs to other agents. But 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 to the agent and resolving ambiguities that arise as a result of overlapping interpretations are the responsibility of up-chain agents.

It is advisable that each agent be kept simple in its responsibilities and be limited in the decisions it needs to make to reap the benefits of distribution and to enhance its learning abilities. The overhead of the required units (the white box) should 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 (e.g., Using the learning module in the white-box rather than creating a separate learning meta-agent).

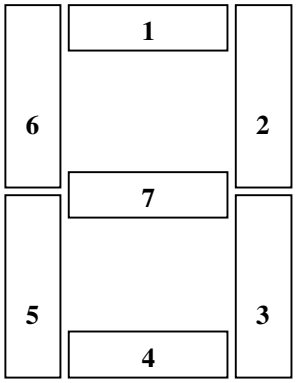
In the following example we shall see that the manner by which a system is agentified depends on the various objectives the designer has in mind.

3.1 The seven-segment example

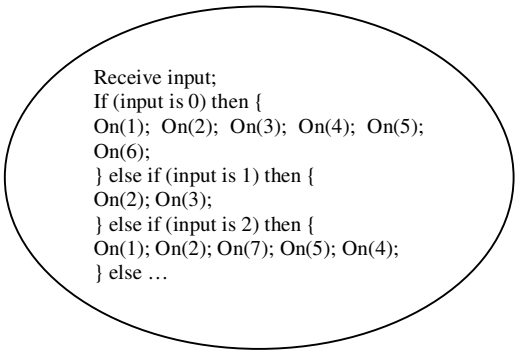
Let us follow the design of a simple application to observe the various advantages AAOSA may bring. The system to be designed takes a number between 0 and 9 and switches on the appropriate LEDs in a seven-segment display (Figure 3.a). There are, of course, tried and tested algorithms for designing this system that give us optimal results. This is mainly because the problem is a limited one, and all possible input and desired output is known.

The first step in the design of this system would be to identify 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) on. A non-modular centralized solution (Figure 3.b) would involve 48 functions and 5.5 condition checks on average assuming each number is inputted with equal probability (1/10).

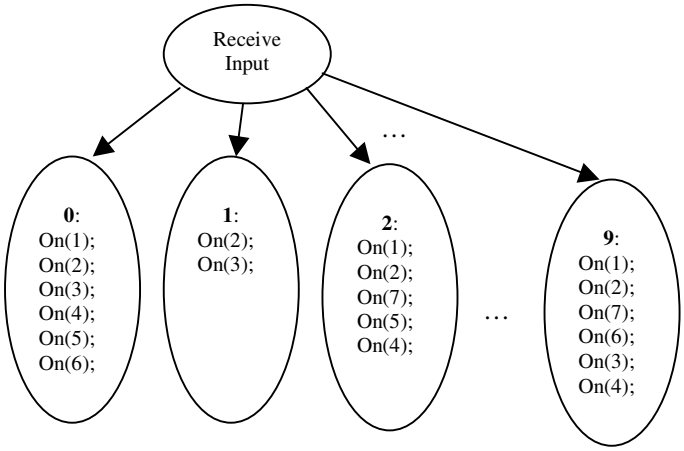
An alternative to this approach would be to have an agent represent each function and an input agent to receive the input and distribute it (Figure 3.c). If this agent (i.e., the Input-agent) were to have any interpretations of



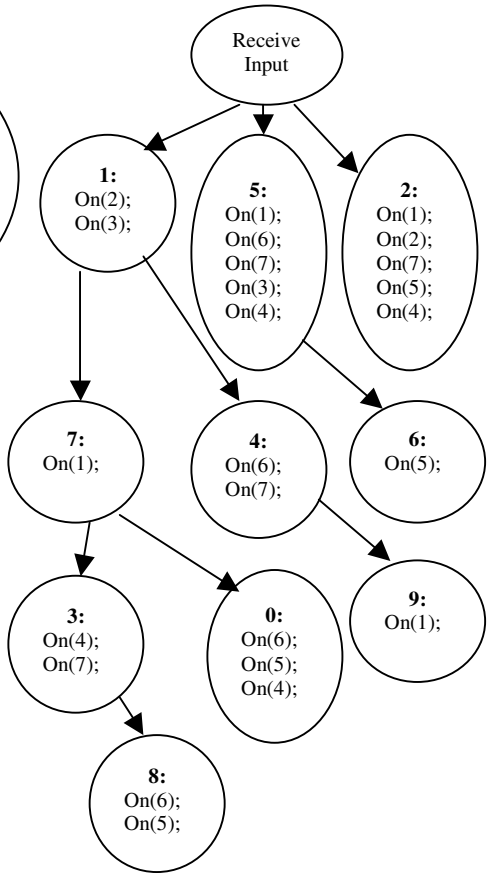
a) A seven-segment display.



b) The centralized approach.



c) Simple AAOSA hyperstructure. Each agent's interpretation policy is to check input against the number it represents.



d) An AAOSA hyperstructure designed to reduce the number of functions and condition checks.

Figure 3) Designing a seven-segment system using AAOSA. Arrows depict the direction of querying.

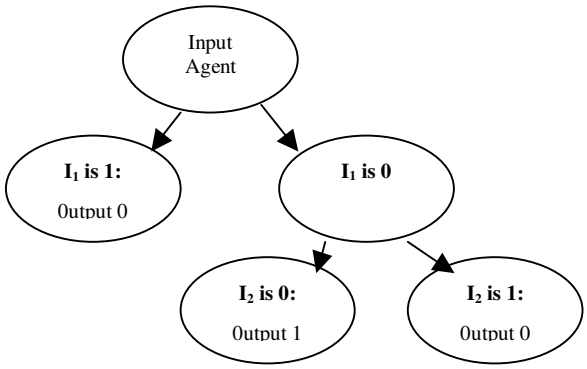


Figure 4) The NOR function using AAOSA. The Input Agent receives I_1I_2 as input. This proves that there exists an AAOSA hyperstructure with no transitive interpretations for any computable function.

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 because the fact that input has been handed down through the Input-agent does not effect the route or process it may be taking later. It is always preferable not to use transitive interpretations as this prevents the agents from being self-sufficient (Figure 4).

Therefore, in the case of the hyperstructure in figure 3.c, 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 one in 3.b, certain useful features have come about because of the way we have modularized. 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 1 condition on average).

As we stated before, 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. Let us consider the hyperstructure in figure 3.d. 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 that of the last two designs). The average condition check, if the system is taken as a running on a fully parallel platform, can be calculated as follows.

Each possible input between 0 to 9 would occur 1/10 of the time,

If input were 1, 2, or 3, we would be checking 1 condition,

For inputs 4, 6 and 7, 2 conditions would have been checked,

For inputs 0, 3 and 9, the number of conditions checked would be 3, and

For input 8, 4 conditions would have to be checked.

Thus, the average conditions checked would be 2.2. Of course, in calculating this number we disregarded the conditions checked in the white-box of the agents during the query and delegation phase. 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 2.c, we have reduced the reusability and increased the average condition checks, in order to minimize the number of functions.

3.2 Learning in AAOSA

The combination of machine learning and multi-agent systems can have benefits for both. Multi-agent systems having learning capabilities will reduce cost, time, and resources and increase quality in a number of ways ([10]):

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

On the other hand, machine learning in a multi-agent setup becomes faster and more robust.

Learning can improve performance in AAOSA software by improving speed and accuracy, reducing interactions, providing generalizations, and helping the system to tune in to different user preferences.

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 also to improve it's 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 hyper-structures of simpler ones, or join redundant agents to form more efficient ones. This brings about the possibility of hyper-structures self-organizing themselves to achieve a balance between the degree of distribution and the efficiency of the overall software. A similar method is described in [14].
  AAOSA should guarantee:
  - The agents each stay responsible for the limited domain they were originally assigned to, while:
  - Containing the distribution of responsibilities, thus:
  - Containing 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".

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 is a form of learning.

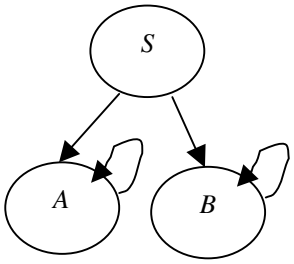


Figure 5: An AAOSA hyper-structure for example 1.

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. As we shall see in the next section, 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.

4 AN AAOSA PARSER

The examples of the previous section are relatively simple because:

- All possible input is known and manageable at design time,
- Only one agent is delegated to at each one time,
- No ambiguities can occur because the interpretation policies of each agent are mutually exclusive over the input.

We will now discuss examples in which some or all of the conditions above are not met.

AAOSA can be used to parse input given in the form of strings of characters. We show that the AAOSA parser can parse any context-sensitive grammar and discuss the time-complexity of this parser in [12]. A parser only interprets input and so an interpretation phase similar to that discussed in section 2 is enough. We will also not need any learning therefore a subset of the performatives in the previous section are needed here. The AAOSA parser is of importance to us because it shows the power of the interpretation phase of the AAOSA methodology.

In a parser, input is not predictable at the time of design and so each agent will have to consider parts of the whole input when interpreting it. Therefore, the claims made by different agents will have to include the portions of the input being claimed.

Example 1. Consider the context-free grammar $G = (\{S, A\}, \{a, b\}, P, S)$, where P consists of:

$$S \rightarrow AB$$
$$A \rightarrow aAb \mid ab$$
$$B \rightarrow cBd \mid cd$$

The string *aabbccdd* does belong to the language this grammar represents because:

$$S \Rightarrow AB \Rightarrow aAbcBd \Rightarrow aabbcBd \Rightarrow aabbccdd$$

4.1 The Algorithm

To parse a language using AAOSA we first need to build a hyperstructure based on the grammar to be parsed:

I. We create an AAOSA agent for each production rule. We are assuming that each production rule in the grammar has a unique left-hand side. We will denote the production rule represented by agent A with R_A.

II. Agent B should be down-chain with respect to agent A, if in the right-hand side of R_A, there is a reference to a variable that exists in the left-hand side of R_B.

III. The agent representing S is the input-agent to this hyperstructure.

IV. The right-hand side of the production rule an agent is representing is that agent's interpretation policy.

As an example, the hyperstructure for parsing the grammar of example 1 is shown in figure 5.

In an agent new claims can only be made based on ex-

isting claims C_n, …, C_m, $(m \geq n)$

Figure 5 shows the AAOSA hyperstructure for example 1. Note that agents may be down-chain with respect to themselves. There is no need, in this case, for agents to query themselves and they may simply reapply their interpretation policies on the claims they have made so far every time they make a new claim. Let us see how this system can parse the input *aabbccdd* to see if it belongs to the language represented by the grammar in example 1:

1) Agent *S* receives input. No new claims apart from the input claims can be made by *S* at this point,
2) Agent *S* queries agents *A* and *B* (*Is-This-Yours?*),
3) Agent *A* makes the following claims and sends them up to agent *S*:

   (*A*, {[1..2]}) (*A*, {[0..3]})
4) Agent *S* cannot make any new claims based on Agent *A*'s response,
5) Agent *B* makes the following claims and sends them up to agent *S*:

   (*B*, {[5..6]}) (*B*, {[4..7]})
6) Agent *S* is able to make the following new claim

   (*S*, [0..7])

   and we conclude that *aabbccdd* is valid in this language:

Note that the order of the sending and receipt of queries and responses does not have any effect on the overall outcome.

The grammar in example 1 is that of a context-free grammar. A less restricted form of grammar is the context-sensitive grammar. In this grammar for every production $\alpha \rightarrow \beta$ in *P*, we may have $|\beta| \geq |\alpha|$ (we use $|x|$ to stand the number of symbols in the string *x*).

Example 2. The following grammar is context-sensitive:

$$S \rightarrow aSBC \mid aBC$$
$$CB \rightarrow BC$$
$$aB \rightarrow ab$$
$$bB \rightarrow bb$$
$$bC \rightarrow bc$$
$$cC \rightarrow cc$$

The language $L(G)$ contains the word $a^n b^n c^n$ for each $n \geq 1$.

The AAOSA hyperstructure for this grammar is shown in figure 6. Let us see how this system can parse the input *aabbcc* to see if it belongs to the language represented by the grammar in example 1:

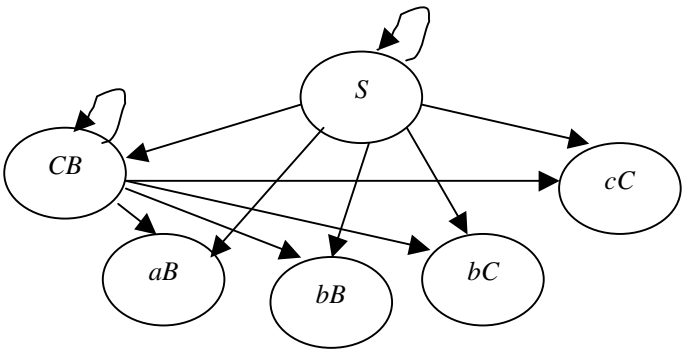


Figure 6) An AAOSA hyperstructure for example 2.

1) Agent *S* queries all at *Depth of Search* 1,
2) Agent "*aB*" makes claim ({[2..2]}, *B*) in reply to agent *S*'s query,
3) Agent "*bB*" makes claim ({[3..3]}, *B*) in reply to agent *S*'s query,
4) Agent "*bC*" makes claim ({[4..4]}, *C*) in reply to agent *S*'s query,
5) Agent "*cC*" makes claim ({[5..5]}, *C*) in reply to agent *S*'s query,
6) (*S* cannot make any new claims and increments *Depth of search* to 2)
7) Agent "*CB*" receives first responses from down-chains and claims ({[3..3]}, *C*) and ({[4..4]}, *B*),
8) S, having received this latest claim from "*CB*" can claim ({[1..3]}, *S*) based on ({[1..1]}, a), ({[2..2]}, B), and ({[3..3]}, *C*),
9) The checking loop for *S* will not break because the resulting claim from step 8 makes the agent be able to make a new claim, namely ({[0..5], *S*) based on ({[0..0]}, *a*), ({[1..3]}, *S*), ({[4..4]}, *B*), and ({[5..5]}, *C*). The parsing thus ends successfully at a depth of 2.

5 AAOSA AND NATURAL LANGUAGE USER INTERFACES

Most human-computer interfaces being used today are complicated and difficult to use. This is due mostly to the growing number of features the interface should provide easy access to.

Users usually have the following problems with current interface [11]:

- Prior to selecting an action. They have to consider if the application provides an appropriate action at all. This hints on a need for some sort of feedback from the application.
- It is hard to access the actions they already know about. This implies that the user should be able to freely express his or her needs without being bound

to the limited conventions preset by the application.

- They have to imagine what would be an appropriate action to proceed with in order to perform a certain task of the application domain. The application, therefore, should be able to guide the users through the many options they may at any stage of the interaction.

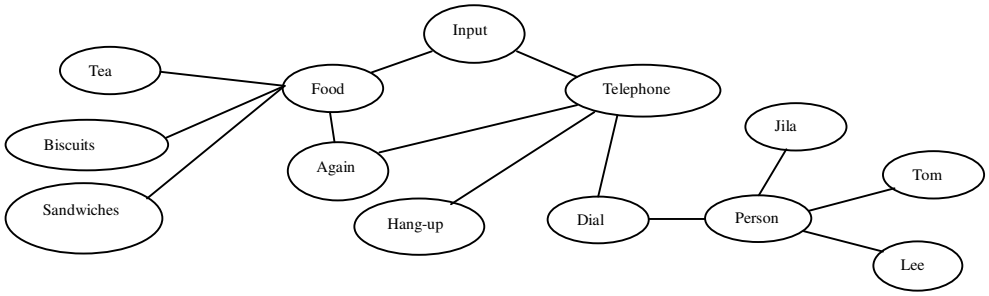


Figure 7) An example hyperstructure covering the semantic domain for the servant robot. Lines show communication paths.

Thus, some of the desirable features in a user interface may be as follows:

- Natural expression: The user should be able to express his or her intentions as freely and naturally as possible.
- Optimum interaction: Interaction should be limited to the following:
  - The user is in doubt as to what she can do next or how she can do it.
  - The system is in doubt as to what the user intends to do next.
- Adaptability: Adaptability could be about the changing context of interaction or application, but most importantly, the system should be able to adapt to the user's way of expressing her intentions. Two main issues that will have to be taken into account in this regard are generalization and contradiction recovery:
  - Generalization: An adaptable system in its simplest form will only learn the instance that it has been taught (implicitly or explicitly). Generalization occurs when the system uses what it has learned to resolve problems it deems similar. The success and degree of generalization, therefore, depend directly on the precision of the similarity function and the threshold the system uses to distinguish between similar and dissimilar situations.
  - Contradiction: A system that generalizes may well over-generalize. The moment the system's reaction based on a generalization is in a manner the user does not anticipate, the system has run into a contradiction. The resolution of this contradiction is an integral part of the learning and adaptability process.

Ease of change and upgrade: The application designer should easily be able to upgrade or change the system with minimum compromise to the adaptation the system has made to users. This change should be done at runtime (i.e., on the fly).

Some work in this regard has been done in the Natural Language Processing world under the title of robust parsing ([15] and [16]). These approaches differ from that of this paper in that they pursue a top down approach to language striving to model the language it self as opposed to the application sub-domain.

5.1 Extending the AAOSA parser

Although, as shown in section 4, a grammatical parser can be implemented using AAOSA, practical problems force us to make some improvements to it. Creating a grammar, be it context-sensitive, is a complicated task.

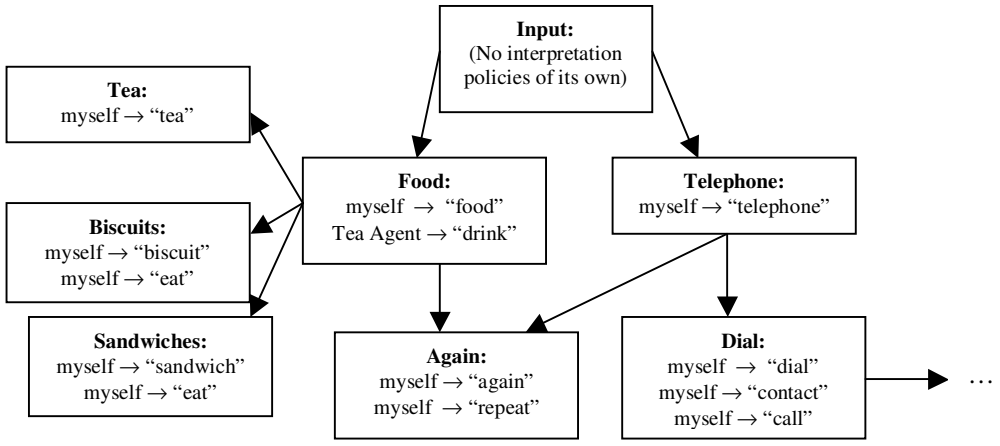


Figure 8) Simple keyword interpretation policies for the robot servant example.

Changing grammars based on learning (section 5) is also difficult. Grammars alone are not enough to fulfill the requirements noted above.

Furthermore parsing alone is not sufficient either. A

parser, after all, tells us if the input string belong to a language or not. Our objective though, is to find the best match for any given input. This means the AAOSA system:

- Should be able to accept non-grammatical input (e.g., "Tea for Jila bring!"),
- Should be able to handle previously encountered input (e.g., "Yabadabadee some milk for me!").

On the other hand, AAOSA should also be able to pinpoint the semantic sub-domains responsible for responding to input. Therefore, we propose a semantic approach to the problem of grammar definition. The designer of a natural language interface application should design a semantic hyperstructure of agents. The input agent at the top would be responsible for receiving input and initiating the query and delegation phase, and the agents representing the functionality of the system would be lowest order nodes of the hyperstructure.

The interpretation policies should be much fuzzier than that of the parser. For instance, rather than requiring the claims on which a new claim is based to be in sequence (step V in section 4), we can require them only to be exclusive. Two claims C_1 and C_2 are *exclusive* ($C_1 \otimes C_2$) if:

based on the history of successful claims made, or the status of the semantic domain it is representing, or even based on interactions with the user.

5.2 The robot servant example

We will explain our implementation on a toy problem: A robot is to run certain errands around the house using a natural language interface. We will start with a limited set of functions and show that this set is extendible:

- Serve tea, biscuits or sandwiches,
- Make telephone calls.

The idea is to distribute the natural language processing over nodes that represent different levels of a hyperstructure covering the functionality space. One example of such an hyperstructure is given in figure 7.

The natural language interpretation is done through a series of claims and delegations carried out by the agents. Agents claim an input string as belonging to them (*internal interpretation*), or decide that it belongs to some other agent or agents based on their interpretation policies (*transitive interpretation*). Agents that are not able to find appropriate policies that interpret certain input will consult down-chain agents. In our example the sandwich

Notes	User	System
1. Flexibility	-"Show me ESPN"	[Channel jumps to 206 on DSS]
	-"Put on freakin' CNN"	[Channel jumps to 202 on DSS]
	-"Mute the sound and change to HBO and put ABC in the window."	[Sound is muted] [Channel jumps to 910 on DSS] [Picture in picture is put on and switched to channel 7]
2. Ambiguity	-"play" -"The VCR!"	-"Would you like to play, the VCR or the DVD?" [VCR starts playing]
3. Context-based ambiguity resolution (DVD and VCR can both "stop")	-"Stop it!"	[VCR stops playing]
4. Ambiguity based Query handling	-"What can I do with the lights?"	-"Would you like to turn the lights on or off, or set them at a specific level?"
5. Learning	-"Sports channel" -"I mean ESPN"	-"Would you like to go to the next or previous channel or jump to a specific one?" [Channel jumps to 206 on DSS]
6. Complex operations	-"Record the Sports channel in half an hour"	[Okay]
7. Limited scope	-"What! You can do that?"	-"Would you like to use the TV, the VCR, the DVD, or the lights?"

Table 5) Examples from the AAOSA based home A/V interface (Scotty). A total of 42 agents are involved in interpreting user responses.

tem it would be enough for B and C to be exclusive. A number of heuristics may be used to approximate the extent of difference from the desired status (e.g., differences in order or proximity of the foci). Confidence in the claims themselves will also have to be taken into account when basing a new claim on them.

Another main difference between the parser and our proposed natural language system is that the context considered in the reductions of a context-sensitive grammar is limited to the input. In the real world, though, the decision to make a claim may be made based on context information that is not necessarily present in the input. For instance, an AAOSA agent may decide to make a claim

positively, causing the Food agent to send a similar affirmative response to the Input agent.

So far, no processing has been done. It is now up to the initiator agent (i.e., the Input agent) to decide whether this input should be actuated. Upon actuation, agents use temporarily stored interpretation results to send the actuation request down to the responsible agents and have them execute the necessary processes without having to reinterpret the input. In our example, "Give me biscuits!", the Biscuit agent will be sent down the actuation command and it will issue the necessary commands needed for the robot to get some biscuits.

Ambiguities occur when the interpretation of input is not possible because of there not being any agents to claim it, or more than one agent claiming it. In the first case something unknown to the system has been inputted and should be clarified and possibly learned by the agents. Take for example the input: "I'm thirsty!". Let's say none of the agents claims this input. In this case, the input agent can ask the user whether "I'm thirsty!" has something to do with telephones or food (i.e., its immediate down-chain agents). The result of this interaction may be learned by the Input agent (e.g., Input agent learns that the input "I'm thirsty!" should be delegated to the Food agent). The Food agent, in turn not being able to interpret, will ask the user whether "I'm thirsty!" has anything to do with tea, biscuits, or sandwiches. Eventually it is up to the Tea agent to learn that "I'm thirsty!" belongs to it. To keep the responsibilities from drifting up-chain or down-chain in the hyperstructure of agents, the Tea agent should at this point declare to it's up-chain agents (in this case the Food agent) to remove any interpretations of "I'm thirsty!" that result in the delegation of the input to the tea agent (i.e., uses the *Un-learn* performative). The Food agent, having received an *Un-learn* performative and processed it, should in turn propagate it further up-chain.

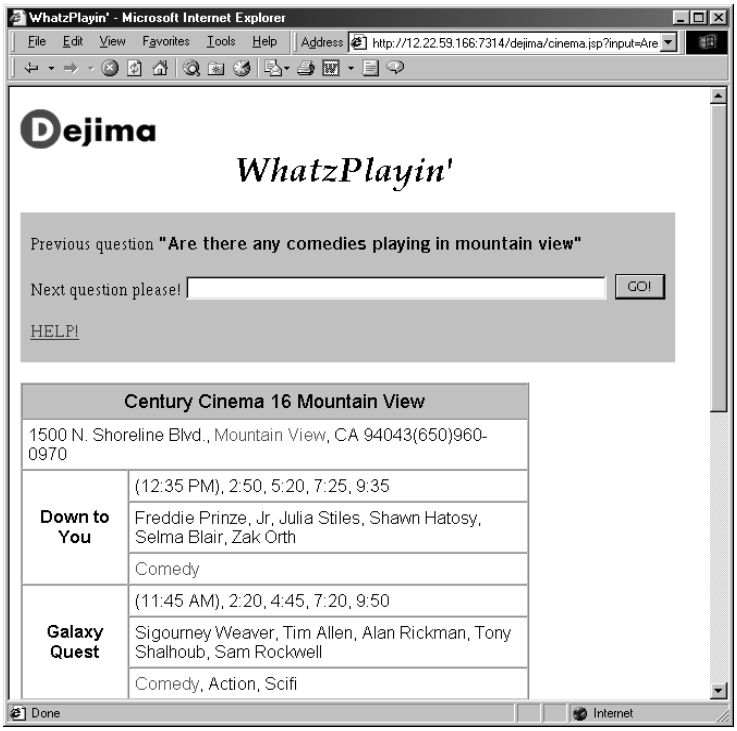


Figure 9) A screen shot of a Cinema NLUI using AAOSA.

Another example of ambiguity is when the natural language input is vague. For example, "I want to eat!" would cause the Biscuit agent and the Sandwich agent both to claim it.

By adding agents to the system, we can extend its capabilities. An example would be adding fax capability to the system in which Fax agent will be added at the same level as the Telephone agent connecting to the same down-chain agents as the telephone agent. Note that by doing so we have also added ambiguity to the system. For instance, "Contact Jila" may result in both the Fax and Telephone agents claiming it.

Some of the interesting attributes of our interface are as follows:

- It is modeless, in that the user does not have to follow preset menus in order to achieve her intentions; (e.g., "Tea!" is a valid input).
- It supports context-based interaction (e.g., If you have just ordered it to "Get some tea!" and follow that order up with "Again!" it can resolve the ambiguity between the Telephone and Food agents based on this context in-

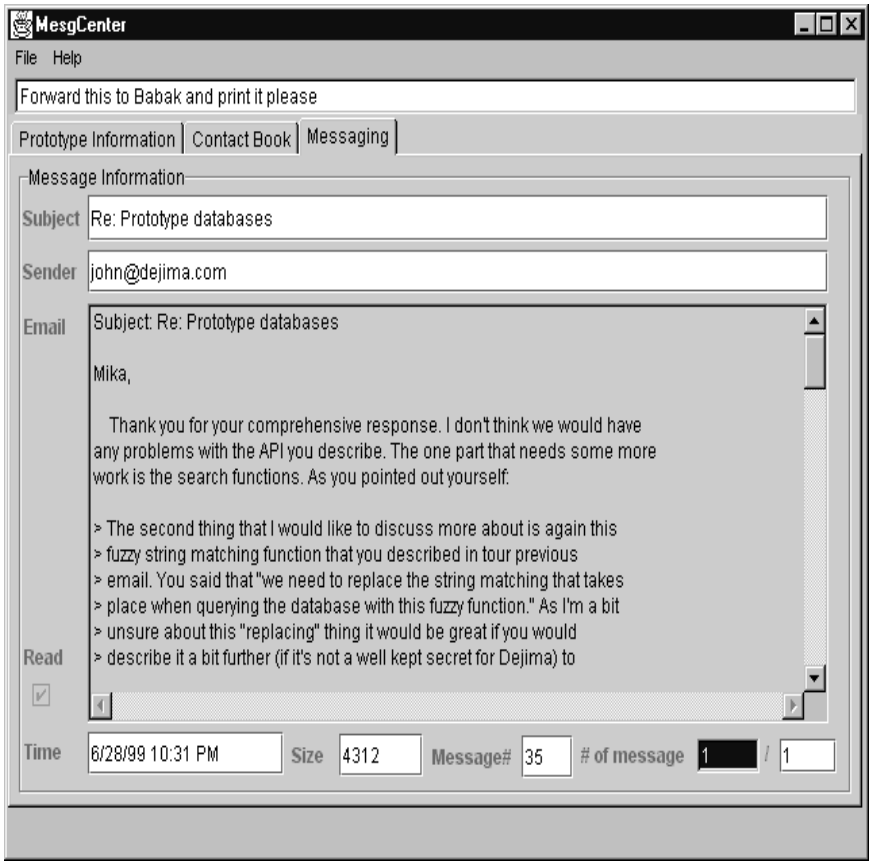


Figure 10) A screen-shot of the AAOSA e-mail and contacts NLUI application.

formation, namely recency of invocation.)

- It can be upgraded easily to be able to handle Multi-lingual input because each agent's interpretation policy is relatively simple and the grammar and semantics are mostly handled over the architecture.
- When mistakes are made by the system graceful error recovery can be achieved by backtracking to the furthest down-chain point of ambiguity resolved implicitly and interact with the user to resolve it. For instance in the "Again!" example if the user really means a re-dial should be attempted, she can express her dissatisfaction by, say, pressing the escape key, and the system will respond with: "Should I re-dial the phone or bring you tea again?"
- Handles incomplete, unpredictable, and grammatically incorrect input. This is possible due to the simple interpretation policies in each agent and the fact that agents can extend their interpretation policies by learning.
- Relatively small memory/processor requirements with respect to similar interfaces based on classical Natural Language Processing methods.

Simple Queries	**Compound Queries**
I feel like a Movie	Comedies with Kevin Spacey
Where is American Beauty playing	Is American Beauty available at Century Park 12
Find theaters near Whipple Avenue	Any Kevin Spacey Comedies at Redwood City
Where are Movies with Kevin Spacey Playing	Are there any Comedies Playing in Whipple Avenue tonight?

Table 6. Sample of the AAOSA Cinema NLUI corpus.

Simple Queries	**Compound Queries**	**Out-of-functionality Queries**
Remove contact with 6504040330	show me John ostrem's address	Pass along unread messages to hank
Open John's contact	Get John's info printed	Trash for contact with email address sun
Delete current message	Print all the e-mails received today	Forward contact to phone 2222222
Print messages smaller than 500	Fax John's message to Albert and print it	Pass along contact to email address bad.com

Table 7. Sample of the AAOSA e-mail and contacts NLUI corpus.

6 AAOSA IN THE REAL WORLD

The first incarnation of an AAOSA NLUI was in the form of a natural language interface to home theater systems (Table 5). AAOSA has since been used in a number of diverse commercial NLUIs. In this section we present a brief summary of two such applications and provide some sample corpus tests and test results[1].

6.1 The Cinema Information NLUI

This is a web-based NLUI that responds to user queries about theatre names and locations, and movie titles, times, actors and genres. Figure 9 shows a screen shot of this application.

A total of 23 agents were used in this application 12 of which used query-by-content on fields of a database as interpretation clues. Corpus creation was done on a total of 10 users while part of the corpus was generated automatically to test the dynamic interpretation done by the query-by-content agents. A total of 348 natural language queries were tested in two categories of simple and compound. The simple corpus where 179 queries that either invoked a single function in the system or caused a single ambiguity. The compound corpus where 169 queries that invoked more than one function in the system simultaneously, or caused more than one ambiguity. Only the first level of ambiguity resolution interactions were tested.

The overall success rate of the system was %92.2. The simple corpus test yielded a %98.9 success rate, while the compound corpus test reached an %86 success rate. Table 6 lists some sample simple and compound queries successfully handled by the system.

6.2 The Email and Contacts NLUI

The e-mail and contacts application is an NLUI for a mailer with the following functionality:

- Show Messages
- Fax Messages
- Print Messages
- Remove Messages
- Forward Messages
- Find Messages by
  - Subject
  - Size
  - Number
  - Date/Time
  - Sender
  - Read/Unread flag
- Show Contacts

[1] Test results and screen shots courtesy of Dejima Inc. (www.dejima.com)

- Fax Contacts
- Print Contacts
- Remove Contacts
- Forward Contacts
- Add Contacts
- Find Contacts by
    - First name
    - Last name
    - E-mail address
    - Cell phone number

Figure 10 shows a screen shot of this application.

A total of 29 agents were used in this application 10 of which used query-by-content on fields of a database as interpretation clues. Corpus creation was done on a total of 29 users. A total of 708 natural language queries were tested in three categories of simple, compound, and out-of-functionality. The simple corpus where 366 queries that either invoked a single function in the system or caused a single ambiguity. The compound corpus where 278 queries that invoked more than one function in the system simultaneously, or caused more than one ambiguity. The out-of-functionality corpus where 64 queries that were classified as outside the defined functionality of the application and were tested for the proper ambiguity resolution interaction. Only the first level of ambiguity resolution interactions were tested (i.e., The user interactions were not included in the tests).

The overall success rate of the system was %91.7. The simple corpus test yielded a %98.9 success rate, while the compound corpus test reached a %91.4 success rate. The out-of-functionality test yielded a %51.6 success rate. Table 1 lists some sample simple, compound, and out-of-functionality queries successfully handled by the system.

6.3The Car NLUI

The Auto application is an NLUI for a Car Interface with the following functionality:

- Climate control
- Audio system
- Navigation system

A total of 155 agents were used in this application 3 of which used inexact query-by-content on fields of a database as interpretation clues. Corpus creation was done on a total of 10 users. Out of a total of 1088 user utterances, 870 natural language queries were used as the corpus to create the application and %20 of the collected data, 218, were used as a control corpus to evaluate the system. Results of the tests are summarized in table 3.

7 CONCLUSIONS AND FUTURE WORK

The application of AAOSA as a web-browsing interface is currently being investigated. Dynamic AAOSA (DAAOSA) is also being investigated as a means to adaptively index large amounts of information. In DAAOSA, agents are created or removed based on the complexity requirements of the system by the agents themselves.

The current version of AAOSA has been implemented in Java because if it's unique portability and multi-platform execution capabilities and multi-threading features. We have used simple rote learning as the learning module in the white-box and other, more robust machine learning methods are being investigated.

In this paper we introduced AAOSA as a new software architecture and gave an example of its implementation in the form of a natural language interactive interface. AAOSA is flexible, primarily because there is no rigid predetermination of valid input. It is modular providing for easier revision, extension and development. AAOSA agents can be re-used inside an application or in other software. The independent nature of AAOSA agents provides for an inherently parallel architecture. Agents can run and communicate over a network of heterogeneous hosts. Run-time addition of new AAOSA agents is possible and therefore incremental development and evaluation is possible. Following guidelines set by the original designer; other designers can also contribute to a system making it commercially attractive. The built-in learning and ambiguity resolution features make AAOSA a more intelligent software architecture.

As in most other distributed designs, AAOSA's unpredictable nature makes it hard to guarantee the correctness or stability of applications. More work needs to be done in this regard to provide sound evaluation and testing methods. AAOSA is highly reliant on messages and the high message traffic between AAOSA agents may create bottlenecks and network load problems.

	Climate	Audio	Navigation	Total
Corpus %80	%93	%67	%82	%80
Control %20	%86	%46	%63	%65
Total %100	%91	%63	%78	%77

Table 8. Summary of the AAOSA car interface NLUI interface.

REFERENCES

[1] V. R. Lesser. Reflections on the Nature of Multi-Agent Coordination and its implications for an Agent Architecture, *Autonomous Agents and Multi-Agent Systems.* pp. 89-111, Kluwer Academic Publisers, 1, 1998.

[2] N. A. Baas. Emergence, Hierarchies, and Hyperstructures. *C.G. Langton ed., Artificial Life III.* Addison Wesley, 1994.

[3] N. A. Baas. Hyper-structures as Tools in Nanotechnology and Nanobiology. *S. Rasmussen, S. R. Hameroff, J. Tuzinki, P. A. Hansson ed.'s, Towards a Nanobiology: Coherent and Emergent Phenomena in Bimolecular Systems*. MIT Press, 1995.

[4] M. R. Genesereth and S. P. Ketchpel. Software Agents. *Communications of the ACM*. Vol. 37, No. 7, July 1994.

[5] D. Martin and D. Moran. 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.

[6] D. Cockburn and N. R. Jennings. ARCHON: A Distributed Artificial Intelligence System for Industrial Applications. *G. M. P. O'Hare, N. R. Jennings, ed.'s, Foundations of Distributed Artificial Intelligence*. pp. 319-344. John Wiley & Sons, 1996.

[7] B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot and M. Balabanovic. A domain-specific Software Architecture for adaptive intelligent systems. *IEEE Transactions on Software Engineering*. April 1995.

[8] S. Franklin, and A. Graesser. Is it an Agent or just a Program? A Taxonomy for Autonomous Agents. *Proceedings of the Third International Workshop on Agents Theories, Architectures, and Languages*. Springer-Verlag, 1996.

[9] J. M. Bradshaw. KaoS: An Open Agent Architecture Supporting Reuse, Interoperability, and Extensibility. *Proceedings of Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*. http://ksi.cpsc.ucalgary.ca/KAW/KAW96/KAW96Proc.html, 1996.

[10] P. Brazdil, M. Gams, S. Sian, L. Torgo, and W. Van de Velde. Learning in Distributed Systems and Multi-Agent Environments. *Machine Learning: EWSL-91 (European Working Session on Learning), Y. Kodratoff (Ed.), Lecture Notes in Artificial Intelligence*, Springer-Verlag, 1991.

[11] T. Kuhme. Adaptive Action Prompting – A complementary aid to support task-oriented interaction in explorative user interfaces. *Technical Report #GIT-GVU-93-19*. Georgia Institute of Technology, Dept. of Computer Science, Graphics, Visualization, and Usability Center, 1993.

[12] B. Hodjat, M. Amamiya, Applying the Adaptive Agent Oriented Software Architecture to the Parsing of Context Sensitive Grammars, IEICE Transaction on Information and System, Vol.E83-D, No.5, pp.1-11, Japan, 2000.

[13] N. Jamali, P. Thati, G. A. Agha, An Actor-based architecture for customizing and controlling agent ensembles, *IEEE Intelligent Systems*, pp. 38-44, March/April 1999.

[14] S. Willmott and B. Faltings , The Benefits of Environment Adaptive Organizations for Agent Coordination and Network Routing Problems, Fourth International Conference on MultiAgent Systems (ICMAS-2000), 2000.

[15] E. J. Briscoe and N. Waegner. Undergeneration and robust parsing. In W. Meijs, editor, *Proceedings of the ICAME Conference*, Amsterdam, 1993. Rodopi.

[16] E. J. Briscoe. Prospects for practical parsing: robust statistical techniques. In P. de Haan and N. Oostdijk, editors, *Corpus-based Research into Language: A Feschrift for Jan Aarts*, pages 67--95. Rodopi, Amsterdam, 1994.