Design of a Generic Learning Interface Agent

by

Max Edward Metral

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1993

Author____
Department of Electrical Engineering and Computer Science
July 9, 1993

Certified By____
Patricia E. Maes
Assistant Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by____
Leonard A. Gould
Chairman, Departmental Committee on Undergraduate Theses

Design of a Generic Learning Interface Agent

by

Max Edward Metral

Submitted to the Department of Electrical Engineering and Computer Science

May 17, 1993

In Partial Fulfillment of the
Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

ABSTRACT

An interface agent is a semi-intelligent, semi-autonomous computational system which assists a user with a computer application. Such an agent is called a learning agent if it monitors and memorizes the user's actions and preferences, discovers regularities in these, and learns to automate them. This thesis discusses research done into the design of a generic learning interface agent. An off-the-shelf electronic mail application was modified to take advantage of current learning interface agent technology. In this test arena, many issues and unresolved questions about a generic agent were discovered. This research led to an initial generic agent architecture presented in this document.

Thesis Supervisor: Patricia E. Maes
Title: Assistant Professor of Media Arts and Sciences

Contents

List of Figures

Chapter 1

Introduction

This thesis describes research into a generic learning interface agent. An interface agent is "software that can take independent actions in the interface on behalf of the user's goals, without explicit intervention by the user (Lieberman 1993)." Learning interface agents act as personalized assistants which look over the user's shoulder and learn to act on the user's behalf. Take for example an electronic mail application. In this information age, users are inundated with hundreds and hundreds of messages of varying priorities. A learning interface agent can observe the user's filing habits. If the user files all messages from his or her superior into the "Important" folder, the agent can discover this pattern and automate this task. Once the agent has observed a large set of user actions, the agent can make decisions and act on behalf of the user in a range of more and less complicated situations.

There are basically three approaches to building interface agents. They differ in the technique used for knowledge acquisition. The three main approaches to knowledge acquisition are user programming, expert programming (or knowledge-engineering), and machine learning. Commercially available agents are mainly user programmed or knowledge-engineered. This thesis uses a learning approach because it seems to provide the best results in practice, and requires virtually no explicit training or effort from the user nor the application designer.

The algorithmic aspects and performance characteristics of learning interface agents have been widely explored in systems such as the Calendar Agent [Kozierok 93], and the ChainMail project [Drake 93]. Both of these systems use Memory Based Reasoning as the machine learning algorithm. It would be advantageous to adapt and test the MBR algorithm on different types of applications such as word processors and the like. Unfortunately, until a generic learning interface agent is designed, this sort of adaptation carries lots of wasted time in coding the application.

Until now, most agent-enabled applications have been written from the ground up. That is to say, the designer was forced to implement the agent as well as the application. There is no fundamental reason why such a tight connection must be made. In this thesis, I explore the issues behind loosening and all together discarding this connection. A first draft of a generic agent architecture provided direction in this adaptation effort. In turn, executing the adaptation refined the generic architecture specification.

The goal of this thesis project was to adapt a commercial application to use learning interface agent services to create the illusion of an agent-enabled application. The next step in the evolution of interface agents is to design a protocol and architecture for adapting any application to use learning interface agents. This thesis presents the preliminary design stage of such a generic interface agent architecture.

Code from the ChainMail [Drake 93] mail handler was used as the core for the learning interface agent. By using pre-written code, there was more freedom to explore the issues and complications associated with generic agents rather than re-coding the agent implementation. Furthermore, the code implements a fully functional agent, allowing us to ignore any agent implementation complications. For results and a detailed description of the ChainMail system, see [Drake 93]. A brief overview of the system is presented in Chapter 2.

The Eudora electronic mail handler was the application which was given agent capabilities [Dorner 91]. There are various reasons why this application was chosen. First and foremost, Eudora is widely used in the Macintosh community. Therefore, there will be many people ready to use and test the Eudora agent when it is available. Secondly, the source code is publicly available. We had no delusions about being able to "agentize" applications without modifying their source code. In a truly generic agent model however, it would be possible to agent-enable an application without modifying application code (See Chapter 5). Furthermore, electronic mail handlers present many opportunities for interface agents' automation efforts.

The crux of this thesis effort involved laying out the communication lines between Eudora and the ChainMail agent. For this project, a platform dependent inter-process messaging protocol, AppleEvents, was chosen. In the generic agent architecture, it may be necessary to abstract IPC one level further in order to allow applications on different platforms to share information. AppleEvents, however, provide many of the services needed for a generic agent architecture. AppleEvents are blocks of information that can be accessed by keywords and can contain complex references to complex objects. For more detailed information about AppleEvents, see [Apple 93] and Chapter 3.

While there have been various instances learning interface agents, one may wonder why a generic agent architecture is necessary at such an early stage. While the motivations behind standardizing an area are clear, most standards emerge after technology comes into public use. In this case, we cannot allow application designers to begin haphazardly programming agents into their applications. With hundreds of separate agents, users are robbed of the trust that is gained from one agent. Furthermore, a generic agent architecture allows agent designers and application designers to work independently, which in turn may deliver higher quality technology in a shorter time. For these reasons, we present the generic agent architecture at this early stage of interface agent technology. Since there is no system in place, the architecture may seem a bit hand-waving. The architecture has been carefully thought out and will hopefully be successful. However, the design is definitely evolving and will continue to evolve until a generic agent is in place.

This thesis report is organized as follows. Chapter 2, "Previous Work in Specialized Interface Agents" explains the agent side of this thesis. In particular, it explains the memory based reasoning algorithm and its adaptation to an electronic mail learning agent. Chapter 3, "Attaching an Agent to an Off-the-shelf Mail Application" explains how Eudora, the commercial electronic mail application, was modified to take advantage of a learning interface agent. Chapter 3 also explains the communication protocols between Eudora and the agent. Chapter 4, "Experimental Results" describes system behavior and discusses the complications when this system is generalized. Chapter 5, "A Generic Learning Interface Agent Architecture" proposes a protocol for applications and agents to communicate in a general manner. Chapter 6, "Conclusions," sums up the work and lays a course for future work in this field.

Chapter 2

Previous Work in Specialized Interface Agents

There are several ways to implement interface agents on modern day computers. The three approaches discussed here are classified by the way they acquire the knowledge the agent uses to make its decisions. User-programmed rule systems require the user to program rules which instruct the agent. Knowledge-engineered systems are given rules by a field expert. Learning system acquire their knowledge from the user by observation. This thesis uses the learning approach to intelligent interface agents. Specifically, we use Memory Based Reasoning [Stanfill and Waltz 86] as the learning technique. This algorithm relies on past examples to predict future behavior.

There have been several experiments in memory based learning interface agents which serve as a proof of concept. One of these, the ChainMail electronic mail handler serves as the foundation of the agent described in this thesis. The ChainMail application shares its Memory Based Reasoning(MBR) engine with the Calendar Agent [Kozierok 93]. This chapter explores the foundations of Memory Based Reasoning and the applicability of MBR to a wide range of applications. It will also discuss the advantages of MBR over other knowledge acquisition methods and learning techniques.

Making real-time predictions for user actions is the core of the interface agent concept. As such, it is necessary to carefully evaluate the three paradigms on a wide range of applications and a wide range of traits. These traits include the running time, domain model required, training procedure, and effectiveness. Domain model is defined as the amount of knowledge the algorithm must have beforehand about the problem it is solving; i.e., can the algorithm make a decision such as filing a message without actually knowing what "filing a message" really means? We now examine the three approaches in detail.

2.1 *Knowledge-engineered Systems*

Knowledge-engineered systems rely on field experts to program intelligence into an application. For example, a tax attorney may program rules into a tax filing program which instruct the program on how to optimize taxation. Knowledge-engineered systems may be implemented without user involvement. The intelligence may operate transparently to the user, since the knowledge is not "tailored" to each individual user's needs.

The running time of knowledge-engineered systems is usually equivalent to user-inputted rule systems. Implementing both of these systems requires the application to detect when rules are applicable and applying those rules. One possible improvement over the user-programming approach is a result of the source of the rules. The knowledge can be built into the application during production, rather than at run time. This can lead to better performance when programmed correctly.

Knowledge-engineered systems require a very strong domain model. The knowledge is programmed into the application by an expert in the field. There is really no way to abstract expert knowledge so that it becomes applicable to other domains. Another disadvantage is that personalized information is not a part of knowledge-engineered systems. Knowledge-engineered systems do not adapt to the user's particular habits or preferences. On the other hand, there is no training for these systems. Knowledge-engineered systems arrive fully knowledgeable and fully effective.

2.2 *End-User Programming Systems*

End-user programming systems are easy to implement, and take little processor time in proportion to the number of rules. The basic concept is that of rule-firing. The user specifies a set of rules, which consist of a trigger and an action. The trigger specifies when the action is to be taken. The agent is then in charge of monitoring the application to see if any of the triggers are applicable; if so, the agent may take the action. For an example of an end-user programmed system, see Object Lens [Lai, Malone, and Yu 88].

The running time of rule-based systems is very good. Triggers can be placed into the applications representation such that they fire automatically. There is no complex calculation of rule applicability. If a trigger is satisfied, its corresponding action fires.

Rule-based reasoning is very popular in commercial applications for exactly this reason. In more complex systems, the rule building process may take computation cycles. User inputted rules may need to be generalized or internalized, requiring a stronger domain model and complex logic.

General rule based reasoning requires a moderately strong knowledge of the application architecture. Specifically, the trigger must know when it is to fire, which requires that it have access to application data and control flow. At the very least, the application must provide an indication of when triggers may fire. Furthermore, specification of rules requires knowledge of how triggers are specified. The domain model problem can be alleviated by clever choice of domain. For example, to make a rule-based system generic one may define the domain as "the operating system." In this way, all applications running on a specific operating system may use the rule system. Gaining this generality often loses the specificity (obviously) of the triggers and actions.

Entering rules into an end-user programmed system can be a simple or difficult task. Triggers can be specified by directing the application to snapshot its state (this may not be space efficient). Suffice to say, the application can make rule specification as easy or as difficult as it wants. From this vantage point end-user programmed systems seem like a good option. However, we must question why the user needs to specify rules at all. Ideally, the agent should be able to determine rules dynamically. To take that step even further, the agent may not even need rules. It may be able to create a decision for a specific instance of a problem, as most humans do, rather than sticking to a hard and fast rule. This is where memory based reasoning shines.

2.3 A Learning Approach

Learning agents enjoy several advantages over user programmed or knowledge-engineered systems. One version of learning agents relies on Memory Based Reasoning to make user decisions. Unfortunately, nothing comes free, and MBR is a computationally intensive algorithm. Memory-based reasoning relies on past situations to predict future actions. By collecting statistics on past situations and what actions were taken, the agent can come up with a prediction and report its confidence in that prediction. MBR does not attempt to make interpretations or generalizations of these past situations; it merely stores them.

MBR's running time balloons as the number of situations increases. Each prediction requires examining all past situations in memory and correlating all the data. Experience with current computers shows, however, that the use of MBR in learning interface agents is becoming tractable. Furthermore, modified MBR systems may use limited generalization to minimize the number of situations in memory.

The power of MBR comes mostly from its lack of generalization. Rather than implementing the intelligence to generalize situations, MBR takes a brute force remembrance approach. This approach, coupled with the inherent generality of the algorithm, requires a very weak domain model. The MBR engine need only know how to extract necessary fields from situations and how to instruct the application to carry out actions. Furthermore, the agent only needs to carry out actions that it has seen before, and therefore doesn't require knowledge of all possible situations or actions.

There is no need for explicit training in MBR systems. If training is desired, it can be done by providing canned examples which enter into the memory as any other situation would. Otherwise, the agent merely watches the user until it believes it has some suggestion that the user may want. By collecting user feedback, the agent can become more or less confident in its decisions. Eventually, the agent is able to automate routine tasks without the intervention of the user.

MBR has proven very effective in "active" applications. That is to say, applications which require the user to respond to certain situations are well suited to MBR techniques. It remains to be seen how well MBR will do with passive, word processing type applications. Passive applications are applications in which there is no obvious "situation-action" paradigm. By the very nature of MBR, these passive applications will have to be re-examined for their "active roots" to be used with interface agents. It is not hard to imagine that this may be possible. For example, a word processor may begin making predictions on the words you are typing or automatically correct spelling on often misspelled words.

For a complete description of MBR, see [Stanfill and Waltz 86].

2.4 MBR Implementation in ChainMail

It is important to understand the general algorithm that results in a prediction from the agent. To make a prediction about a new situation, the agent compares the new situation with memorized situations in the same memory. The distance metric used is a weighted sum of the distances for all different features of the situations. The weights used are based on correlation statistics so that relevant features carry more weight. The *n* closest matching situations, where *n* is a system parameter, are retrieved and a prediction is made based on what actions were taken in those situations. The algorithm also produces a confidence level in this prediction, which ranges from 0 to 1. The prediction and confidence levels are used as follows: If the confidence level in the prediction is above the "tell me threshold" the agent will offer its suggestion to the user. If the confidence level is above the "do-it" threshold the agent will automate the action on behalf of the user and notify the user in a user log.

We now turn to the implementation of MBR in this thesis. The core abstraction is that of a *situation-action pair*. A situation is defined as an application state. An action is defined as a user response. A situation-action pair represents an application state and the user's response to that state. Situation-action pairs (henceforth SA-pairs) are stored in *memories*.

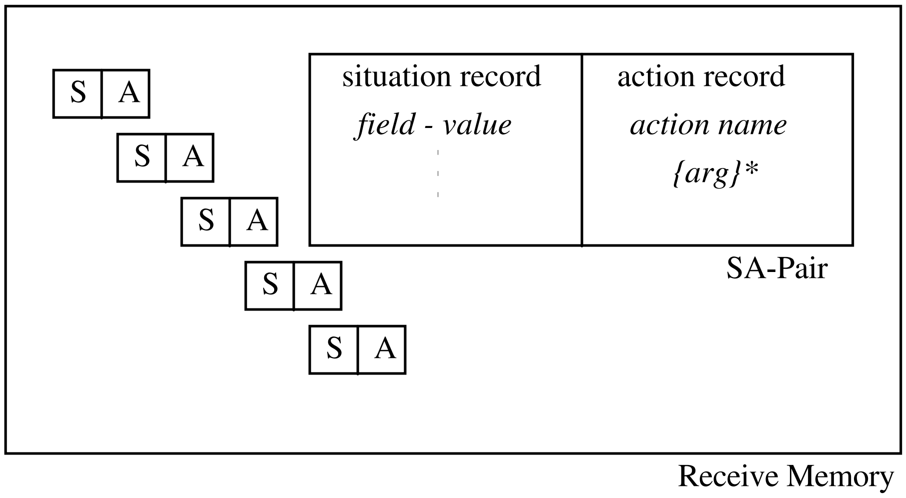


Figure 1. Low level abstractions in the ChainMail MBR engine.

A memory is a collection of SA-pairs of a certain type. For example, one memory in the electronic mail agent consists of SA-pairs relating to the event of a message being received

(See Figure 1). Using this segmented memory scheme speeds up reaction time by pre-filtering irrelevant situation-action pairs. Unfortunately, as we will see in Chapter 5, a distinct delineation may not be adequate, as there may be "specializations" of situation classes. All recollection and action prediction on a given situation takes place in that situation class's memory. Presently, there are four memories in the electronic mail agent: Refile, Receive, Read, and Delete; i.e., there is a separate memory for memorizing what happens after filing, receiving, reading, and deleting a message respectively.

We now back up and examine the high-level system organization. All the memories are stored in a global lookup table. This lookup table functions as the *knowledge base* for the agent. The memories in the lookup table only store completed SA-pairs. Therefore, only past situations are stored in the knowledge base and predictions can be made solely from these experiences. Unfinished SA-pairs are bundled with the mail message until they are paired with an action. In the original ChainMail code, these messages are stored in their proper folders or mailboxes. Since this concept is not necessary for this project, all messages are stored in another lookup table. Using this message store, Eudora can refer to messages by their lookup id rather than always passing all information structure repeatedly. In the generic agent architecture, some sort of object reference protocol will need to be implemented in order to avoid such repetition. Both of these abstractions are shown in Figure 2.

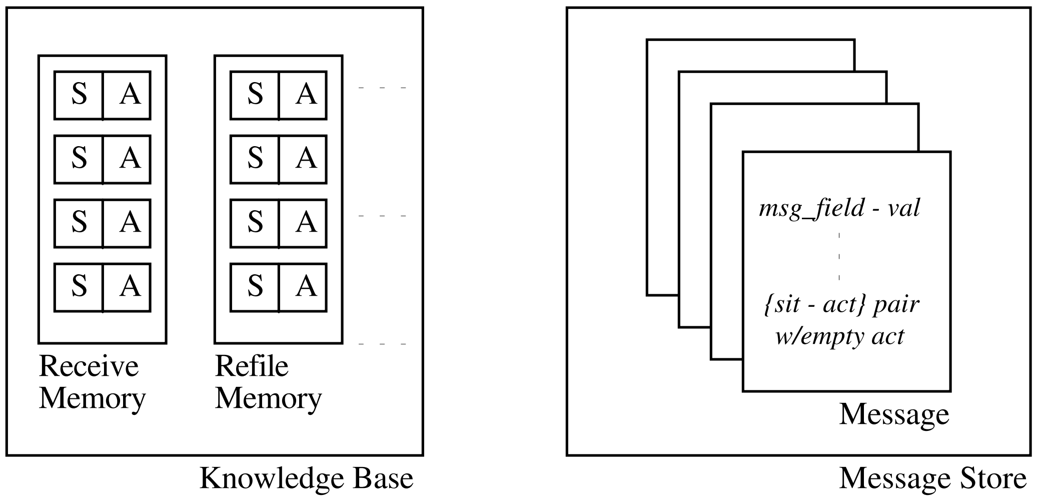


Figure 2. High level system abstractions

To tie all the pieces together, let’s examine the path of a newly arrived message through the system. Mail messages make their way into the system through outside sources. The user begins to deal with messages as soon as they enter the system. Therefore, the first thing that happens to a message is its reception. When the message is received, ChainMail creates a *Receive* situation. The MBR engine then examines the *Receive Memory* for any similar situations. If the agent has a prediction, it will prompt the user for approval. If the suggestion is approved, the action will be carried out. If there is no prediction, the situation stays unmatched. Let's assume the agent suggested that the message be refiled, and let's also assume the user agrees. A *Refile Action* is created and paired with the Receive situation. This newly created SA-Pair is placed in the Receive Memory as an experience. A new situation is now constructed for the message, in this case a *Refiled* situation. The process repeats and eventually terminates.

When the SA-pair generation process terminates or if there is no initial prediction, there will likely be an unmatched situation. This situation is carried with the message. Any future user action on that message is considered a resolution to this unmatched action. The user action generates a corresponding action record which is paired and placed in the appropriate memory.

Chapter 3

Attaching an Agent to an Off-the-Shelf Mail Application

The process of agent-enabling the Eudora mail handler provides many insights into the design of a generic agent architecture. The main goal in the generic agent design will be to minimize the task of the application programmers. Furthermore, there should be a standard process to inject agent capabilities into existing applications. This chapter presents the process used to enable Eudora, which serves as a model for the generic adaptation process. Our process was simplified by the fact that the ChainMail agent was designed as electronic mail agent, giving it a strong domain model.

3.1 The Eudora Mail Handler

The Eudora mail handler is a complex, complete, Internet mail handler for the Macintosh family of computers. It has gained popularity for its ease of use and speed. In this case, the popularity is well deserved. Eudora is coded very well, and its parts are distinctly separated. Without such excellent coding, adapting this large application would be virtually impossible. We can directly predict the difficulty of adapting an application by examining the design of the code. When strict object oriented practices are employed, an agent can be added by inserting a few hooks in the code and exporting a few data structures.

In this case, a subset of the Eudora functionality was selected for automation. This subset corresponds to the functionality implemented in the ChainMail handler. By choosing this subset, we can merely emulate the state of the Eudora handler in ChainMail. Reading, refiling, and mail reception are “exported” to the ChainMail agent. Since these actions provide a solid basis for testing the agent's success, they are sufficient. In a full implementation, however, there are easily over one hundred actions that are eligible for automation.

Only a few Eudora details need to be known to understand the communication between the agent and Eudora. Relevant code fragments are presented in Appendix B. The basic message storage unit is a *mailbox*. A mailbox contains the text of all the messages in that box. It also contains a *table of contents* which describes the layout of the mailbox. This table of contents is read into the application's memory when a mailbox is opened. We use the table of contents to access all pertinent information about messages.

The table of contents contains an array of *message summaries* for each of the messages in the mailbox. To pass information to the agent, we use the header fields contained in this summary structure. There is one exception. The "To:" field of a message is not stored in the message summary. Since the To field can often be useful in determining what happens to a message, we make a call to another Eudora function *FindHeaderString*. *FindHeaderString* loads the message text into memory and searches for the To field. In a released version, it would be advantageous to add the To field to the message summary and save the disk read time.

One of the fields of the message summary serves a special purpose. The *seconds* field stores the value of the machine clock when the message arrived at the computer. This field serves as the unique identifier of the message. This unique identifier, the *u_id*, is used as the ChainMail lookup key. Any situations involving a message can just reference the *u_id* instead of passing the entire message information block to ChainMail.

3.2 Communication Between the Agent and Eudora

As stated in the previous section, mail enters the user realm from an external source. When mail is received, Eudora builds a message information block for each new message and transmits it to ChainMail. The message information block is formed as follows:

```
{
      long u_id         % unique message identifier
      string from       % who the message is from
      string to         % who the message is addressed to
      string cc         % carbon copy list
      string subject    % the subject of the message
} MsgInformationBlock
```

Once again, it's important to realize that after this information block is transmitted, it can be referenced by its unique identifier. In the generic architecture, it will be necessary to implement some form of this inter-application object reference.

There are two other data blocks which correspond to the other actions that are exported to ChainMail:

```
{
      long u_id
      string from_box   % Box that message is in
      string to_box     % where the message will be filed
} RefileInfoBlock

{
      long u_id
} ReadInfoBlock
```

The above messages are sent to ChainMail using the AppleEvents protocol. AppleEvents provide location independent inter-process communication. In other words, processes can communicate over the network as if they were communicating on the same machine. AppleEvents are keyword specified data blocks. An AppleEvent block maps four character field identifiers (such as 'fbar') to data values. To send an AppleEvent, the application selects a target, builds the AppleEvent block, and calls a function to send the AppleEvent. The sending application can either wait for a reply or continue about its work.

In our implementation, the user chooses the agent to link with when the first event is ready to be sent. Figure 3 shows the dialog box presented to the user. The user picks a Macintosh and the agent application on that Macintosh. There is also a menu option to pick another agent after this first choice. Situation notification events (the events described above) are sent when they occur. Eudora sends the events to the agent and waits for a reply. If the agent has a suggestion, it is returned in this reply. Eudora displays the suggestion. At the moment, Eudora never takes the action autonomously. This allows us to monitor the agent's performance at all times. If there is no suggestion, ChainMail still sends a reply, but it is empty. This empty reply serves as an assurance that ChainMail received the AppleEvent.

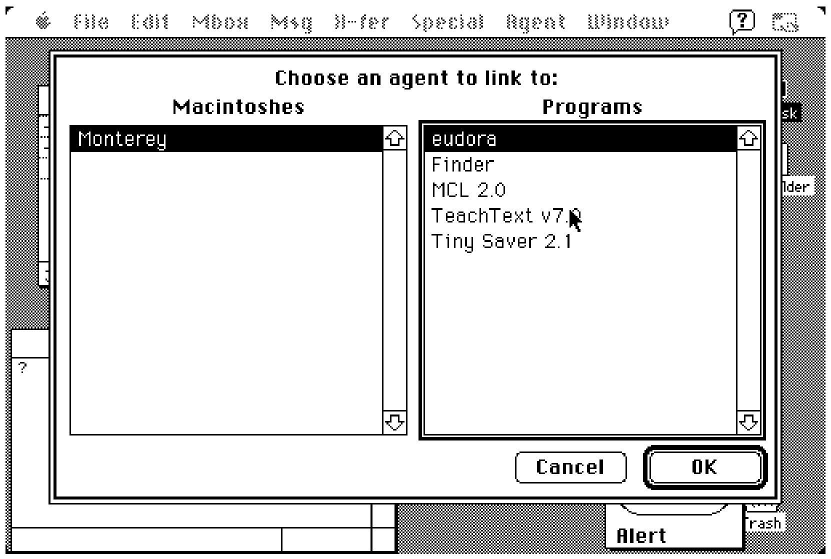
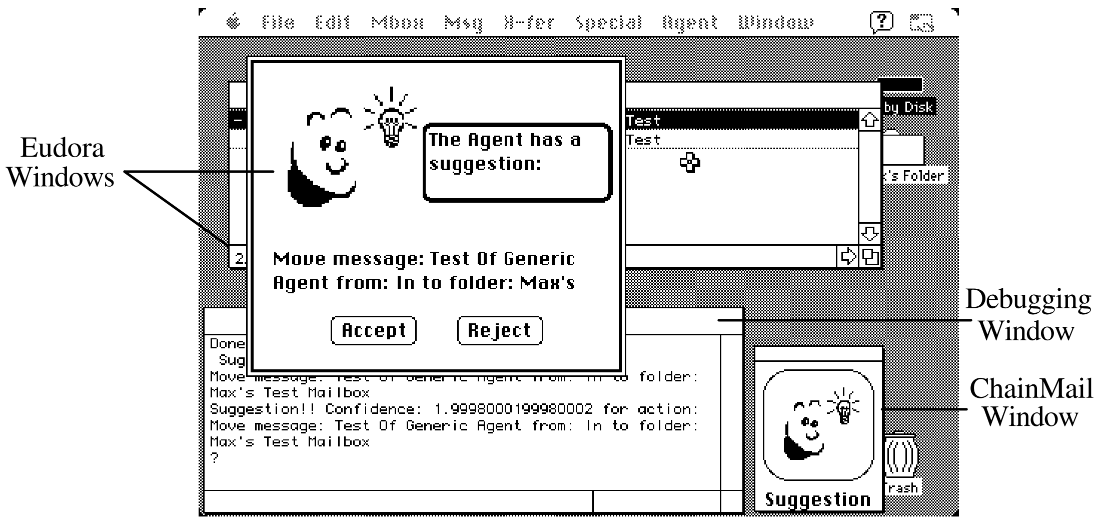


Figure 3. Selecting a target for Eudora notification.

After a suggestion is presented to the user as in Figure 4, he or she may decide whether to accept or reject the suggestion. If the suggestion is rejected, the agent sits back and waits for the user to take an action. If the suggestion is accepted, the action is performed and the agent is informed that its suggestion was accepted. The act of executing the action may initiate another notification cycle, which may lead to another suggestion. Theoretically, this cycle can continue indefinitely. For example if the user accepts a suggestion to read a message, the execution of the read action will be sent to the agent. In turn, the agent may suggest that the message be refiled...*ad infinitum*.



```
Move message: Test Of Generic
Agent from: In to folder: Max's
```

```
Done
 Sug
Move message: Test Of Generic Agent from: In to folder:
Max's Test Mailbox
Suggestion!! Confidence: 1.9998000199980002 for action:
Move message: Test Of Generic Agent from: In to folder:
Max's Test Mailbox
?
```

Figure 4. Presentation of Agent Suggestions in Eudora.

To execute an action, ChainMail sends an AppleEvent to Eudora. Presently, this AppleEvent can take two forms: refile and read. A refile action provides the source mailbox, destination mailbox, and unique message identifier. A read action provides the unique message identifier. When Eudora receives one of these messages, it looks up the message in question by its identifier. The requested action is then performed on the message.

Situation notification can take place in various circumstances. These circumstances can be divided into two categories: user initiated and system initiated. The user initiates agent interaction through the agent menu. The selection “Test Agent” dumps the contents of the inbox to the agent. This choice allows the user to update the agent's view of the application across user session. In the current implementation, this operation is necessary in the absence of persistent knowledge base storage. Another menu selection, “Suggest” transmits any currently selected messages and asks for agent suggestions.

System initiated transactions include new mail, read mail, and refiled mail. When new mail arrives from the server, Eudora automatically sends the relevant information about these messages to the agent. Any predictions are received by Eudora at this time. When the user reads a mail message, Eudora passes this information along to the agent as discussed previously. User initiated and system initiated transactions provide a strong basis for testing the interface agent hypothesis. With the current suite of events, the agent can provide practical assistance to the user.

Chapter 4

Experimental Results

4.1 Hindsight

After adapting Eudora to use the ChainMail agent, the system was used on real mail for a few days. The results gave insight into several problem areas as well as reinforcing certain decisions. Problems included delay, action synchronization, and action correctness. The communication strategy was quite successful, however. Furthermore, the overall adaptation was fairly straightforward, requiring modest additions to existing code.

Communication between the ChainMail agent and Eudora is effective, but slow. The situation-reaction cycle can take several seconds. During this time, the user cannot do anything. As the running time of the agent prediction increases when the number of situations grows, the user becomes more and more annoyed. A better strategy would be to send the agent a situation, and then let the agent reply on its own time. If a reply is not received, the application can proceed unmolested. If a reply is received, the application can present this suggestion to the user.

Another problem we encountered was atomic action execution. For example, if a message is refiled, we must wait until the message is actually refiled before notifying the agent. If we do not wait, further predictions on the message may result in inconsistent values; e.g., if the message is read through an agent prediction, it will not be in the proper mailbox and therefore the read will fail. The solution is to wait until an action is completely finished before notifying the agent. Furthermore, we must insure that the user cannot bypass the atomic nature of operations. In other words, we must allow agent suggestions to be executed fully before any other transactions occur.

A more serious problem comes from the structure of the knowledge base. The problem is illustrated as follows: all reactions to reading a message are stored in the same memory. Let's assume that there are two actions in this memory, one suggesting we refile a message from “In” to “Out” and another suggesting we refile “Out” to “In.” If a message in the “In” box matches the second situation better, the suggestion may be to refile from “Out” to “In.” Of course, the message is not in the “Out” box. In the Eudora agent, we can merely check the action's applicability to the current situation. In a generic agent architecture which has a very weak domain model, this problem is not so easily solved. If we rely on the user, the problem will be solved by “survival of the fittest.” In other words, the user will not select this action, and therefore this action will not be suggested anymore. However, if we rely on the user, the agent may corrupt the program and/or user data by suggesting an impossible action.

4.2 Foresight

While the ChainMail agent engine will eventually be discarded, it is important to decide what aspects of its design will apply to the generic agent. The concept of a global knowledge base will definitely carry over into the generic agent. The components of this global knowledge base will need to be examined. Other problem areas include the situation representation, the action representation, and situation action matching. We examine these problems in this section and present a solution in Chapter 5.

A more complex memory structure will be required to implement the truly generic agent architecture. The global knowledge base is an appropriate reflection of a user's habits. This knowledge base could store user habits for multiple applications and multiple situation types. The problem with the current representation lies in the exclusivity of the individual memories. An attempt to match a situation cannot reference multiple memories. There are many applications which would benefit if not require this type of cross referencing. For example, when a message is read, it may be desirable to leave the situation in the Receive memory as well as in the Read memory. If the message is refiled, it may not be relevant that the message is read.

A closely related problem is that of generalized situation-action matching. Assuming we have solved the multiple memory problem, we are left with two other problems. First and foremost, there may not be a strong binding between an action and a situation; i.e., it may not be clear which situation is satisfied by an action. Secondly, an action may be a response to multiple situations (or vice versa). The general problem can be restated as, "Given an action A and a set of unresolved situations S, how can we match A to S?"

There are many options available for solving these problems. The easiest is to delegate situation-action matching to the application. This is undesirable since we would like to make agent-enabled application programming as painless as possible. At the other end of the spectrum, the agent could be required to do the matching. This option has several weaknesses. We must provide a means for matching without having a strong domain model. The most obvious means is to execute a decision making algorithm of some kind which matches situation fields with action fields using some sort of best-fit algorithm. Unfortunately, this may carry a very heavy computation penalty. Another possibility would be to introduce a unique identification concept which specifies what situations are resolved by a given action. A possible solution and rationalization will be presented in Chapter 5.

Situation specification presents various problems when extended to multiple domains. In the electronic mail domain, situations are clearly specified at compile time. We must provide a protocol to negotiate situation formats dynamically. It would also be desirable to have the capability to add individual fields to existing situations dynamically. Furthermore, we must do all of this as quickly and infrequently as possible. Situation specification should not need to be done with each new instance of a situation. Action specification presents similar problems. Additionally, we must know how to fill in the arguments of an action. To further complicate matters, filling in action arguments requires knowledge of the situation representation. To summarize, predicting an action requires the agent to know how to fill in the action fields using the situation fields.

It is important to realize that many of the scalability issues presented here are not directly relevant to the generic agent architecture. While we must keep this issues in mind, the generic agent architecture should not concern itself with the specifics of agent implementations. However, it is necessary to visualize an agent implementation that functions with the generic agent architecture. We must know that implementing an agent in the architecture is feasible and practical. By presenting solutions to the above problems in the framework of the architecture, we prove feasibility.

Chapter 5

Design For a Generic Learning Interface Agent

The generic agent architecture is meant to allow applications to use services of *learning interface agents*. To successfully specify a protocol for learning interface agents, we must deal with the following issues.

- **Binding** - A means for applications to locate and use agents.
- **Situation Specification** - A means for applications to provide the agent with sufficient state information to make predictions.
- **Inter-application Object Reference** - Allowing applications to create objects in the agent's world.
- **Dynamic Field Negotiation** - A means to extend situation specification at run-time.
- **Action Specification** - A means for the agent to "pull marionette strings" in the application. In other words, a means for the agent to specify which action the application is to execute and on what objects it is to be executed.
- **Action Execution** - A specification which details how the application is to execute agent predictions in order to insure atomicity and synchronization.
- **Situation-Action Matching** - Matching user or agent predicted actions to unresolved situation(s).
- **User Feedback** - A specification detailing how to provide feedback to the user about the status of the agent. This encompasses real-time feedback about the state of the agent as well as the explanation of agent suggestions.

5.1 Overall System Architecture

The generic agent architecture relies on several components to achieve location independence. Location independence allows an application on one machine to use an agent running on another machine. In a computationally intensive environment, it may be desirable to off-load agent algorithms onto a separate machine.

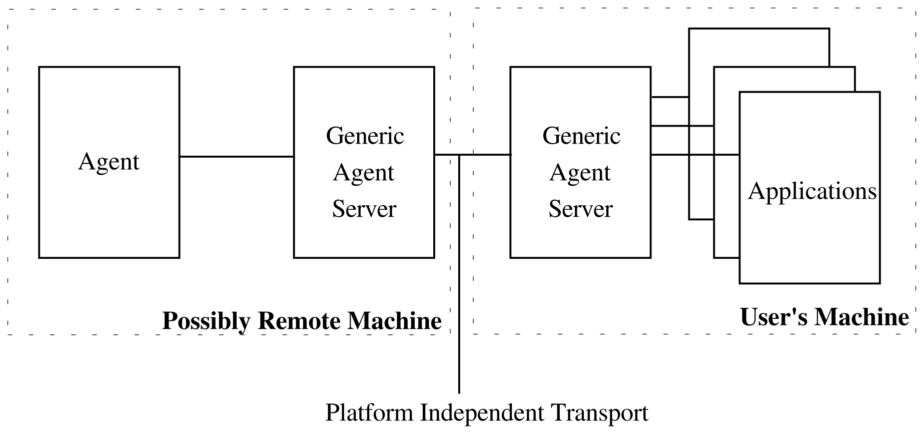


Figure 5. Key subsystems in the generic architecture.

A generic agent system instantiation consists of at least three components, and possibly four. Applications form the first component of the system. The Generic Agent Server forms the second, intermediary component. The actual agent implementation forms the final component. If the applications and the agent run on separate machines, there may be two generic agent servers. The generic agent server (GAS), seemingly unnecessary, proves central to the generic interface agent abstraction. The agent need not concern itself with network transport systems or other platform's IPC mechanisms, since it is guaranteed that a generic agent server will be running on the same machine. The GAS may be bypassed in the *binding* phase if the agent and application reside on the same machine.

5.2 Binding

Binding is the first stage in the generic agent protocol (GAP). The application must locate the user's agent and inform the agent of the application's presence. We briefly treat the issue of actual agent location. One option is to instruct the user to locate the agent. This approach is taken in the Eudora agent. Another option would be some sort of name service, binding user's to their agents. To use name service, an application would contact a *name server* which would give the application the location of the agent it is to use. The name service approach is likely to become popular as more applications begin using the GAP.

After agent location, we must specify what occurs in the binding phase. The application will send a message to the agent (through the GAS) containing information about the application (deliberately vague at this point). This information block may contain such things as application name, manufacturer, or type of application. It will definitely contain a *user feedback preference* (described in the User Feedback section).

The binding stage also presents the opportunity to bypass the GAS when the application and agent are running on the same machine. The GAS returns a platform dependent *location specifier* to the application. In an optimized case, the GAS will return the agent's location so that a direct conversation can be established. As a side effect of this optimization, we must insure that the GAS does not mutate any communication between the application and agent. If it does, we will be unable to remove the GAS from the communication path. After the location specifier is returned, the binding phase is completed.

5.3 Situation Specification

Situation specification is the primary means for an application to provide an agent with relevant application data. From the application's point of view, it may provide this information in a well structured way. From the agent's point of view, however, it has no idea as to the structure or meaning of situations. As such, we must specify a means for the agent to deduce relevant structural data from the application. Furthermore, we must supply the agent with verbal explanations of the structural content so that the agent can explain its actions.

The problem is thus reduced to dynamic situation specification. We turn to an AppleEvents [Apple 93] like mechanism for situation classification. A situation type is represented by two identifiers, its *application class*, and its *situation class*. Each of these identifiers is a 4 character value, called a long, as in the AppleEvents protocol. A situation record is passed in the following format:

```
long application_class;
long situation_class;
{DATA_BLOCK field_n;}*
```

(* signifies zero or more repetitions of the enclosed data.)

Upon initial receipt of a new situation record, the agent has no knowledge of the format of the record. To gain this knowledge, the agent issues a *define* request to the application. The define request returns a template describing the situation record in the following form:

```
long application_class;
long situation_class;
long superclass_app_class;
long superclass_sit_class;
int number_of_fields;
{
     FLD_TYPE field_type_n;
     FLD_METRIC_TYPE field_metric_type_n;
     OPTIONAL FLD_METRIC_DESC field_metric_n;
     STRING field_description_n;
}*
```

The situation block record contains class identification members. The `superclass_app_class` and `superclass_sit_class` identify the situation class from which the current class inherits. These fields may be zero if there is no super class. By using a limited class structure, we allow applications to generalize their state. For example, the electronic mail application could have a “base class” *message_situation*. *Read_situation*s and *refile_situation*s could be subclasses of this base class. The agent can then make predictions based on examples in **both** the read and refile memories, based on the common *message_situation* super class.

The situation block record also contains a description of each field and its contents. The `field_metric_type` describes the metric that will be used in distance calculations. The meaning of this field and its companion, `field_metric`, is discussed later in this section. The last record member, the field description, gives a user-language description of the field's purpose. This description will be used in explanations. For example, for a subject field in an electronic mail application, the description may be “The subject field.” In this manner, an explanation may be given as “I thought you might like to take action *xxx* since **The subject field** of the message *yyy* was 'File Me'.” While there may be small grammatical or aesthetic mistakes, the explanation is clear.

An interesting problem that arises from the need for explanations is: Given an arbitrary field, how do we display it? One solution would be to allow application specified presentation functions. This option is not taken in order to minimize complexity of the protocol and minimize effort of application designers. In a fully generic architecture, however, this would be the correct solution. If the field is of a known field type, such as a string or number, display can be handled by the agent alone. If the field is of an unknown type, we rephrase the explanation. For example, take an unprintable field foo which is described as "The foo graph component." We would explain a match in this field as "I though you might like to take action *xxx* since **The foo graph component** of the current document *yyy* is similar to a past example." Basically, the architecture makes no provisions for unprintable fields. The individual agent instantiation is responsible for printing the field or rephrasing the explanation. Furthermore, in the initial versions of this protocol, we do not allow application defined presentation functions.

While custom presentation may not be a priority, custom distance measurements are virtually a requirement. Learning algorithms, especially MBR, require a means to determine if two field values are equal. Some algorithms must determine the distance between two values. We can not require the agent to successfully compute equality or distance on unknown field types. While the agent should provide a core set of standard field types and distance measures, we must pawn off custom field types onto the applications which created them. To get a distance measure for custom fields, the agent sends a message containing the two fields to the application. The application will return a number to the agent indicating the absolute distance of the two fields. This distance measurement is on a scale of 0 to the 1. A value of 0 indicates equality. The agent may discard relative distance measurements in favor of a simple equality test, but we shall not impose this restriction on all agents. Furthermore, as we will see, custom metrics allow objects to be used without an inter-application object system.

Fields in a situation may take on many different types. The `field_type` member of the field descriptor tells the agent what type of field is contained within the descriptor block. In the electronic mail agent, all fields are either strings or numbers. In other applications, arbitrary data blocks may be part of a situation specification. The generic agent provides basic data types which are commonly used. Arbitrary data blocks are given a single field type code. The default distance metric is provided by the agent, but is not required to do anything more than a bit by bit comparison. To effectively use bit fields, applications must provide a distance metric.

A bit field (or any type of field) with an application installed distance metric provides a primitive object reference facility. For example, an application could store an object reference identifier (such as a number) in a field. When the metric is called, the application can lookup the object identifier and find the object. Unfortunately, this sort of object reference is *not* sufficient. We require multi-field objects; this “hack” only provides single field objects.

We must make one final note before discussing the inter-application object protocol. There are two meta-types of fields: arguments and "true fields." An individual field can be one or both of these meta-types. True fields represent features used in learning algorithms. Arguments are values which will be found in an action corresponding to a situation. For example, if an action is "compose a message," we need to specify who the message will be to, what the subject will be, etc. The values of these action fields may be dependent upon what situation is being resolved; e.g. if a message has been received from user *memetral*, with subject *Meeting Request*, a message to *memetral* with subject *Meeting Acknowledgment* may be composed. This topic will be explained further in the *Action Specification* section.

5.4 Inter-application Object Reference

To provide an inter-application object reference, there are many different options. There are several available protocols on many different platforms. We do not use any of these protocols for various reasons. First, many of these protocols are paltry attempts at forming an object oriented operating system. Since these attempts are not successful, many do not provide the kind of flexibility we require. Furthermore, many of these systems carry much baggage. We do not need many of the services provided by something like OLE [Microsoft 92]. Finally, different platforms use different object protocols. We do not want to write translators into the GAS, as this would entail significant programming and computation efforts. In a truly object oriented operating system, a new protocol would not be required. Currently, there are no truly popular truly object oriented systems.

With the above discussion in mind, we set out to define an Agent Object Reference Protocol (AORP) which is as simple as possible. We define two types of objects: simple and complex. Simple objects are merely a conglomeration of fields. Complex objects are a conglomeration of fields with custom accessor functions. Simple objects should serve most applications sufficiently. Complex objects provide an opportunity to almost completely bypass the agent's storage mechanisms. The next paragraphs describe these objects in detail.

5.4.1 Simple Objects

Simple objects closely resemble situation records. They are merely groups of fields. An instantiation of a simple object is referenced by the application class, object class, and an object identifier. The application class is the same as the application class of the situation record. This is not a hard and fast rule; an application which uses multiple application classes may give its objects any application class identifier. The object class and object identifier are used to uniquely identify the object, and can be set to any value the application wishes. To sum, an object is referenced by a twelve byte value. To avoid clashes between applications, some sort of real life “object registry” such as the one in place for AppleEvents, will need to be established.

To create an object, an application issues an object creation request to the agent. This message consists of a four byte application class identifier and a four byte object class, and an optional four byte object identifier. If the agent knows about the object class being requested, the agent will request the object block itself. After the application supplies the data block, the agent returns the four byte object identifier, which will be the same as the optional identifier, if it was supplied (and not in use by another object). An object block consists of `{DATA_BLOCK field`$_n$`;}*`. Since we assume the agent already knows the format of the simple object, this data block is sufficient.

As in situation specification, if the object type is unknown, the agent issues a define request. An object definition has the form:

```
int number_of_fields;
{
     FLD_TYPE field_typeₙ;
     FLD_METRIC_TYPE field_metric_typeₙ;
     OPTIONAL FLD_METRIC_DESC field_metricₙ;
     DATA_BLOCK field_descriptionₙ;
}*
```

which is very similar to a situation class template. After an object is created, it can be referenced by its twelve byte `<application_class, object class, object identifier>` value. Also, an object can be deleted by sending a delete object request to the agent.

5.4.2 Complex Objects

Complex objects are very similar to simple objects with a straightforward extension. Complex objects allow application defined accessor functions. These accessor functions allow the same customization options as application defined metrics. Furthermore, they provide a back door to custom presentation functions. A complex object is defined as follows:

```
int number_of_fields;
function accessor_function;
{
     FLD_TYPE field_type_n;
     FLD_METRIC_TYPE field_metric_type_n;
     OPTIONAL FLD_METRIC_DESC field_metric_n;
     DATA_BLOCK field_description_n;
}*
```

There is only one new field, the `accessor_function` field. This field points to a function which takes the object block and a field number and returns the requested field. The accessor function enables several capabilities. Custom presentation functions can be implemented by returning printable fields from the accessor function. Custom metrics could be implemented by returning numbers or some such field type (although this is not necessary).

5.4.3 Object Persistence

Application objects in the agent data space must be persistent. In other words, agent objects must persist across multiple user sessions. There are several reasons for this requirement. Situations and actions in the agent's space which reference objects will persist across user sessions. Furthermore, the corresponding objects in the application space may also persist. Non-persistent objects present difficulties in both these situations.

If objects were not persistent, situations referencing the object would be invalid. The agent could create local copies of the objects for its situations, but this would lead to problems down the road. For example, if the application recreates an object on a subsequent session, we must insure that the comparison of the agent-local object and the newly created application object returns equality. Without provisions for object persistence, this would be computationally difficult. Unfortunately, with simple object persistence, we require the application to keep state information with regards to the agent object set. With a compromise we can satisfy both camps.

The AORP deals with object persistence by providing Location Functions, Disposal Functions, and Garbage Collection. Full treatment of this protocol is beyond the scope of this thesis, but we will explain the functionality of each of these functions. Location Functions take an object block and search for an equivalent object in the agent space. If an object is found, the object handle is returned. Disposal Functions remove an object from the agent's object space. They are not required to remove any object references in situations in the agent's space. Garbage Collection removes any objects in the agent's space which are not referenced by situations in the agent's space. Garbage collection can be an expensive operation and should be used with care.

In order to determine how objects will persist, we obviously must know when a user session is about to end. There are two ways to end a user session: the application may quit, or the agent may quit. In the case of application exit, the application is in charge of cleaning up any AORP-created objects. When an agent quits, there may be a problem since applications are counting on agent services. To insure graceful shutdown, the agent sends a message indicating that the user session is about to end to all applications connected to the agent (possibly including the GAS). At this time, the application should clean up any stray data and notify the user of the agent's untimely demise.

5.5 Dynamic Field Negotiation

With a limited situation class hierarchy, dynamic field negotiation becomes less necessary. To create a new field, one can merely derive a class from the previous class. Dynamic field negotiation is still desirable, since adding a field to a class will add that field to all its subclasses as well. Dynamic field negotiation will also provide for deleting fields. Adding and deleting fields requires making changes to all situations in a given class, and therefore should be done sparingly.

Adding a field to memory provides a new prediction feature. We must update any previous situations with a value for the new field. It would be desirable to give the old situations a meaningful, individual value for that field, but this is not plausible. In absence of this method, the new fields of the old situation are all given the same value. This value is supplied by the application and must be the same type as the type of the new field. A field is added by providing a field block as in situation specification:

```
long application_class;
long situation_class;
{
     FLD_TYPE field_type;
     FLD_METRIC_TYPE field_metric_type;
     OPTIONAL FLD_METRIC_DESC field_metric;
     DATA_BLOCK default_value;
     STRING field_description;
}
```

This message adds field *field_description* of type *field_type* to situation type <*application_class, situation_class*> and initializes the new field of all old situations of this class to *default_value.*

Deleting a field is straightforward. The application provides the application and situation class identifiers as well as a field number to delete. The agent will go through all situations in its memory and remove the specified field.

5.6 Action Specification

Action specification describes the methods used for the agent to pull "marionette strings" in the application. By pulling marionette strings, the agent can simulate user actions in the interface. Action specification presents several difficulties to agents with an imprecise domain model. The main problem is concisely directing the application to execute a certain action. The agent must learn to cull action arguments from situations. The best way to simplify this matching is to require the application to identify action arguments in situation templates.

Argument marking is accomplished by a straightforward extension to field type identification. There is a set of basic types T, such as integers, floating point numbers, characters, objects, etc. The field type codes include two codes for each element of the set T. The first code is used when a field is solely a prediction feature. When the field is meant to be an argument, the second code is used. Furthermore, when the argument type is used, it must be followed with an argument name (a string). An action record then consists of the following data block:

```
STRING action_id;
int fixed_args;
{
     STRING arg_name;
```

```
          DATA_BLOCK field_value;
      }*
```

While the arg_name field is not strictly necessary, it will help us with matching this action to other situations. Since the situation template also contains the string identifier, we can uniquely match an argument to its corresponding situation field. The corresponding fixed_args number tells us how many of these arguments are gleaned from situation records, and how many are prediction arguments. This concept will be explained fully in Section 5.8.

5.7 Action Execution

When an agent makes a suggestion, the application is in charge of executing the action. However, an action must be executed carefully, so as to insure consistency of the agent's data and the applications data. The application must look at two aspects of action execution: atomicity and applicability.

Action atomicity has been previously discussed. The basic tenet we must follow is: when a given action is executed, we must insure that the application does not unintentionally inform the agent again. Some applications may need this type of recursion, and it isn't prohibited, but it must be thought out very carefully. To help solve this problem in the general case, application designers need remember one thing. Inform the agent of an action only *after* it has been completely done.

As discussed in Section 4.2.1, it is very difficult to guarantee that all actions suggested are applicable. We can take several approaches to dealing with this problem. We could try to tackle the difficult task of guaranteeing actions, we could ignore the problem, or we can off-load the problem onto the application. To guarantee actions, we could provide a third field type, “hard matches,” which *must* match in a situation. This may complicate matters too much. If we all together ignore the problem, we may put the application at risk by relying on unspecified error handling capabilities. Instead, we take the third approach. We require the application to check whether or not an application is applicable. If an action is not applicable, the application must ignore it. This frees the agent from attempting to learn too much about the application, and frees the application from attempting to choose which situation fields must match fully.

5.8 Situation-Action Matching

One of the more difficult problems we must solve is situation-action matching. By naming the action arguments, we've made the problem feasible. While situation-action matching is a key portion of the system, we can sum up the strategy and system in one sentence. *An action is matched to all situations which exactly match its fixed arguments.* In other words, if an action has one argument, e.g. “message”, which is equal to an object **foo**, it will be matched to all situations which contain the same argument with the same value, **foo**. Another approach would be to match it to a single situation which matches. We postpone this decision until an agent is designed and tested.

Fixed arguments, as specified in Section 5.6, divide the arguments used in this matching algorithm and arguments for the use of the application. For example, assume a *refile action* consists of {message_object, from_box, to_box}. An instance of this action would be {**foo**, Inbox, Outbox}. If we attempt to match this to situations, we may find {**foo**, Inbox} in some situations, but we will not likely find the target box “Outbox” in any situation, since it is the predicted mailbox. In this record, {**foo**, Inbox} are the fixed arguments and {Outbox} is the prediction argument.

It is important to prove the correctness of this approach formally.

An action should be matched to any situations containing all its fixed arguments.
We force this assumption onto application designers. They must design their situations such that this statement holds. In an electronic mail handler, for example, making the message object an argument would help accomplish this task. Any action relating to that message would usually resolve any situations containing the message. If a situation does not contain all fixed arguments of an action, we can be sure it's not a match. There is no way to create the action record with the situation record, since we do not know all the argument fields. Therefore, we cannot allow it to match the situation since we could not use this information in the future.

As a side benefit of this method, insuring action correctness is less difficult. The problem presented in the refile example in Section 5.7 is solved in this section by making the from_box a fixed argument.

5.9 User Feedback

User feedback is necessary for successful user/agent interaction. The ability to know what the agent is up to at all times is key to user confidence. Unfortunately, presenting this information presents various problems. If the agent constantly informs the application and relies on the application to inform the user, the agent will lack a common user interface. If the agent is in charge of user feedback, feedback may be obscured by the active application. We must provide a means of customizing user feedback to specific applications while providing a common user interface. There are two types of user feedback: agent status, and suggestion explanation. We will first discuss agent status feedback, and we refer to it as status feedback in the next sections to simplify the discussion.

The GAP provides four methods of status feedback corresponding to the location of the visual feedback and the application which actually performs the feedback. The methods are: AppOwned-AgentDrawn, AgentOwned-AgentDrawn, SystemOwned-AgentDrawn, and AppOwned-AppDrawn. During the binding phase, the application requests one of these options. We will discuss each of these options in detail.

AppOwned-AgentDrawn places the feedback window in the applications data space/screen space, and asks the agent to take care of updating this window. This option is useful for applications which don't wish to concern themselves with the details of status feedback, but wish to present the feedback in their window space. Each platform will have its own version of this method. Applications and agents will code to the specifications of the platform on which they run. The generic agent server will be in charge of translating in situations where the agent and application reside on separate machines.

AgentOwned-AgentDrawn feedback is equivalent to the feedback in the Eudora agent. The agent feedback window resides in the agent's workspace and is completely handled by the agent. The application is not at all involved or concerned with status feedback. Unfortunately, in practice, the agent feedback window is easily obscured by the application's windows. AgentOwned-AgentDrawn feedback over the network is handled by the GAS on the user's machine. The remote agent sends update events to the GAS, which in turn draws the window.

SystemOwned-AgentDrawn feedback is equivalent to AgentOwned-AgentDrawn feedback from the application's point of view. In this form of feedback, the agent modifies some visible system component, such as the cursor, to reflect the current agent state. The particular implementation of this feedback will be handled by each platform's GAS. For example, changing the cursor is feasible in a Windows environment, where any application can modify the system cursor. In a system like X Windows, where each application has their own cursor, another system resource may need to be modified. Furthermore, on a Macintosh, it may be more appropriate to use the Notification Manager [Apple 93] to modify the application icon of the system menu.

Finally, AppOwned-AppDrawn feedback allows the application to handle feedback in any way it wishes. The application will receive messages from the agent indicating the current status. The application is in charge of reflecting this state in any way it desires. For example, on a Windows system changing the system cursor is feasible. However, in an X Windows environment where each cursor has its own cursor, another system resource would be a better indication of agent status.

We will now briefly treat the issue of explanations. In the action section, we mentioned the *prediction identifier*. This identifier is used to look up an explanation for a given suggestion. The explanation is returned as a string. Agents are only required to store one explanation per application, corresponding to the last suggestion given to that application. In this case, the *prediction identifier* can be meaningless. In an ideal case, the *prediction identifier* will uniquely identify a prediction and its corresponding explanation. To obtain an explanation, applications send a request with the identifier and the agent returns a string. This solution is not optimal, but provides a generic, computationally light way to obtain explanations.

5.10 Loose Ends

There are various issues that have been glossed over in this specification. Networking issues have not been clearly specified. Field types have not been defined. Furthermore, no security has been designed into the system. Security is definitely necessary, as surreptitious agents could wreak havoc on users. These topics are best suited for a "white paper" technical document describing the intimate details of the architecture.

As a final thought, we consider how to agent-enable applications which cannot be modified. For example, how can we provide agent services to commercial applications which we cannot change the source code? There is no simple answer to this question. The application must be able to export data objects and action notifications to the system level. Many applications, especially on the Macintosh, do export this information. If the information is accessible, we can write a translator that accepts this information and translates it to the GAP. While this translator will likely not be as useful as a full GA implementation, it is likely to help to some degree.

Chapter 6

Conclusions

Adapting the Eudora electronic mail handler exposed many issues in the design of a generic interface agent architecture. Inter-application object reference, user feedback, and action correctness issues were discovered in the adaptation effort. Overall, the system performed well and proved that a generic agent was a feasible concept. The next step in this experiment will be to design an agent for Eudora which complies with the generic agent architecture.

Adding agent notification routines to Eudora was fairly straightforward. As discussed, it's important to minimize the application designer's task so that application can quickly come up to speed in agent technology. To agentize Eudora we merely added a few hooks in the code where relevant actions or situations arose. These hooks made calls to separate routines which built a message for the agent and sent it. This modularity is key and will carry over into the generic agent architecture. While the GAP is quite general and complex, its generality gives us the ability to write a primitive agent which satisfies the requirements.

Our first agent will fully implement various aspects of the GAP and design around others. For example, many binding issues can be avoided by working solely on one platform. Inter-application object persistence can be implemented primitively, by agreeing to place all object responsibility on the application. In other words, we need not implement the garbage collection and lookup functions if we count on the application to handle its own objects. Dynamic field negotiation will not be a necessary feature for Eudora, which will not attempt to use this advanced capability. Custom field metrics will also likely not be needed. The main point is that the GAP contains provisions for these advanced features, but a first cut at implementing a GA does not require us to use all the features of the GAP.

References

[Apple 93]

Inside Macintosh: Inter-Application Communication, Apple Computer, Addison-Wesley, 1993.

[Dorner 88]

Eudora: Bringing the P.O. Where You Live, Qualcomm Inc., Copyright 1988-1992 University of Illinois Board of Trustees.

[Drake 93]

ChainMail, Bachelor's Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA.

[Kozierok 93]

A Learning Approach to Knowledge Acquisition for Intelligent Interface Agents, Master's Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA.

[Lai, Malone, and Yu 88]

Object Lens: A “Spreadsheet” for Cooperative Work, *ACM Transactions on Office Information Systems* 5(4):297-326.

[Maes and Kozierok 93]

Learning Interface Agents, *Proceedings of AAAI 1993*, AAAI Press Forthcoming

[Microsoft 92]

Singer, Jon, “OLE: A Short Overview”, *Microsoft Professional Developers CD Pre-release 2.*

[Stanfill and Waltz 86]

Toward Memory-Based Reasoning, *Communications of the ACM* 29(12):1213-1228

Appendix A

Modified ChainMail Code

runme.lisp - Loads all system components and initializes variables.

```
;-*- Mode: Lisp; Package: CCL -*-
(defclass Message () ())
(defclass Action () ())
(defclass File-Action () ())
(defclass Copy-Action () ())
(defclass Delete-Action () ())
(defclass Remove-Action () ())
(defclass Reply-Action () ())
(defclass Forward-Action () ())
(defclass Read-Action () ())
(defclass Reaction () ())
(defclass Field () ())
(defclass Memory () ())
(defclass Situation-Action-Pair () ())
(defclass Prediction () ())

(defvar *dbg-break* nil)
(defvar *dbg-act* nil)
(defvar *dbg-mem* nil)
(defvar *dbg-pred* nil)
(defvar *dbg-react* nil)

;  Redefine this function to end debug messages
(defun dbg-print (s)
   (print s))

(defvar *print-dbg* t)
(defun dbg-format (&rest args)
   (when *print-dbg*
      (apply #'format *debug-io* args)))

(defun dbg-break (dbgid)
   (when (and *dbg-break* dbgid) (break)))

(defvar *dbg-break* nil)

;  Load the files in order
(defvar *disk-path* "Moby Disk:Desktop Folder:Max's Folder:Generic Agent:")
(load (concatenate 'string *disk-path* "macros.lisp"))
(load (concatenate 'string *disk-path* "agent-exp.lisp"))
(load (concatenate 'string *disk-path* "Memory.lisp"))
(load (concatenate 'string *disk-path* "Message.lisp"))
(load (concatenate 'string *disk-path* "Fake-Mailbox.lisp"))
(load (concatenate 'string *disk-path* "Field.lisp"))
(load (concatenate 'string *disk-path* "Action.lisp"))
```

```
(load (concatenate 'string *disk-path* "Reaction.lisp"))
(load (concatenate 'string *disk-path* "Prediction.lisp"))
(load (concatenate 'string *disk-path* "eudora.lisp"))
(init-field-list)
(init-reaction-list)
(init-action-list)
```

agent-exp.lisp - Code to update the user feedback window

```
;;---------------------------------------------------------------------------
;;  Need to bring up the suggestion box here ;;
(defvar *suggestion-list* '())
(defmethod suggest ((pred Prediction))
 (dbg-format "~&Suggestion!! Confidence: ~F for action: ~A" (confidence pred)
                         (present (action pred)))
   (setf *suggestion-list* (adjoin pred *suggestion-list* ))
   (set-agent-expression 'suggestion)
   pred)

;; This is a kludge
(defvar *suggestion-dialog-result*)
(defun activate-agent (exp)
   ;;; Bring up box, and do agent stuff
      (when (not (first *suggestion-list*)) (return-from activate-agent))
   (let* ((suggestion (pop *suggestion-list*))
           (suggested-act (action suggestion)))
      (if (= 1 exp)
         (progn
            (do-action suggested-act)
            (set-agent-expression 'gratified))
         ;;
;; Need to do something right here to adjust weights etc away!!! ;;
         (progn
            (set-agent-expression 'surprised)))
      ;; should delay, then set to alert
      (view-draw-contents *agent-window*)
      (sleep 1)
      (set-agent-expression 'alert)))

(defvar *count*)
(setf *count* 0)
(defun maybe-confused ()
   (setf *count* (+ 1 *count*))
   (unless (eq *count* 3)
      (progn
(set-agent-expression 'confused) (view-draw-contents *agent-window*)
         (sleep 2)
         (set-agent-expression 'alert))))

(defmethod make-suggestion-dialog ((suggestion string)) (make-instance 'dialog
         :window-title "May we be so bold..."
         :view-size #@(200 145)
```

```
        :view-font '("Geneva" 12 :plain)
        :view-subviews (list
                            (make-dialog-item 'static-text-dialog-item
                                        #@(5 5)
                                        #@(190 15)
                                        "Should we:"
                                     nil
   :view-font '("Geneva" 12 :bold)) (make-dialog-item 'static-text-dialog-item
                                        #@(10 20)
                                        #@(180 30)
                                        suggestion
                                     nil
   :view-font '("Geneva" 12 :plain)) (make-dialog-item 'default-button-dialog-
item
                                        #@(35 120)
                                        #@(60 20)
                                        "Accept"
                                        #'(lambda (item)
                                               (declare (ignore item))
(setf *suggestion-dialog-result* t) (return-from-modal-dialog :closed)))
                    (make-dialog-item 'button-dialog-item
                                        #@(105 120)
                                        #@(60 20)
                                        "Reject"
                                        #'(lambda (item)
                                               (declare (ignore item))
(setf *suggestion-dialog-result* nil) (return-from-modal-dialog :closed))))))
```

Action.lisp - Action execution functions

```
;;-----------------------------------------------------------------------------
;;
;; An Action is the stored representation of a descrete happening to a
;; message.  Actions are stored in situation action pairs.  Each message has a
;; list of it's SAs, and each SA that has an action can refer back to the
;; message.  This allows better descriptions of matching situations, as well
;; as allowing different forms of analysis (someday)
;;
;; Actions can be created either by UI actions or as the result of a predicted
;; Reaction.
;; The following actions  are currently defined:
;;           Reply-Action       user selects Reply
;;           Forward-Action     user selects Forward
;;           Remove-Action      user removes a message from a folder (UNUSED)
;;           Delete-Action      user deletes a message entirely
;;           File-Action        user moves a message to another folder
;;           Copy-Action        user copies a message to another folder
;;
;; You must run init-action-list before using actions.
;;-----------------------------------------------------------------------------
(defclass Action ()
   ((name
      :initform "Unknown!"
      :allocation :class
      :accessor name
      :type string
      :documentation "Name of this action")
     (message
      :initarg :message
      :accessor message
      :type Message
      :documentation "The message that was acted upon")
     (succeeded
      :initform nil
      :accessor succeeded
      :type symbol
      :documentation "t if action succeeded, nil otherwise")))
;;----------------------------------------------------------------------------
;; Required methods for subclasses
;;----------------------------------------------------------------------------
;;
;; Do-action does the requested action.  This is called indirectly either by
;; the UI or by the prediction, which is made in an Reaction object.
;;
;; If do-action fails to do anything, it should return nil.
;;
(defgeneric do-action (act))
(defmethod do-action :around ((act Action))
   (let ((msg (message act)))
      (dbg-format "~&Taking action: ~A on message: ~A"
         (name act) (subject-string msg))
      (when (last-memory-incomplete-p msg)
         (let ((lm (last-memory msg)))
            (dbg-format  "~&Updating  message:  ~A"  (subject-string  msg))  (dbg-
            break *dbg-act*)
            (setf (action lm) act)
            (memorize lm (memory lm))
            (dbg-break *dbg-act*)))
      (call-next-method)
      (if (succeeded act)
         (dbg-format  "~&Action  complete:  ~A  on  message:  ~A"  (name  act)
         (subject-string  msg))  (dbg-format  "~&Action  failed!:  ~A  on  message:
         ~A" (name act) (subject msg)))
      (succeeded act)))

(defgeneric stuff-appleevent (action reply))
(defmethod stuff-appleevent :around ((action Action) reply)
   (ae-put-parameter-char reply :|text| (present action))
   (call-next-method))
;;
;;  Predicate  to  test  if  two  actions  are  equivalent.   Crucial  to  the  MBR
algorithms.
;;
(defgeneric actions-equalp (act1 act2))
(defmethod actions-equalp (act1 act2)  nil)
```

```
;; Two different type actions are certainly not equalp (defmethod actions-
equalp ((act1 Action) (act2 Action))  nil)
;;
;; Generate a printed representation of the action
;;
(defgeneric present (act))
(defmethod present (act)
   (format nil "~&Unknown Action! ~A" act))
(defmethod present ((act Action))
   (format nil "~&Unknown Action ~A! Message: ~A"
                (name act) (subject-string (message act))))
(defgeneric make-copy (act msg))
(defmethod make-copy (act msg) nil)
;;-----------------------------------------------------------------------
(defclass File-Action (Action)
   ((name
      :allocation :class
      :initform "File"
      :accessor name
      :type string
      :documentation "Print name of this action")
    (to-mailbox
      :initarg :to-mailbox
      :accessor to-mailbox
      :type string
      :documentation "The mailbox to file it under")
    (from-mailbox
      :initarg :from-mailbox
      :accessor from-mailbox
      :type string
      :documentation "The mail box to move it from")))

(defmethod do-action ((act File-Action))
   (let ((from-mbox (from-mailbox act))
          (to-mbox (to-mailbox act))
          (item (message act)))
      (when from-mbox (delete-from-mailbox from-mbox item))
      (add-to-mailbox to-mbox item) (setf (succeeded act) t)))

(defmethod actions-equalp ((act1 File-Action) (act2 File-Action)) (eql (to-
   mailbox act1)
         (to-mailbox act2)))

(defmethod make-copy ((act File-Action) msg) (make-instance 'File-Action
     :message msg
     :to-mailbox (to-mailbox act) :from-mailbox (from-mailbox act)))

(defmethod present ((act File-Action))
             (format nil "~&Move message: ~A from: ~A to folder: ~A" (subject-
             string (message act))
             (from-mailbox act)
             (to-mailbox act)))

(defmethod stuff-appleevent ((action File-Action) reply)
   (ae-put-parameter-type reply :|type| :|file|)
   (ae-put-parameter-char reply :|obox| (from-mailbox action))
   (ae-put-parameter-char reply :|dbox| (to-mailbox action)))

;;--------------------------------------------------------------------
(defclass Receive-Action (Action)
   ((name
     :allocation :class
     :initform "Receive"
     :accessor name
     :type string
     :documentation "Print name of this action")))

(defmethod do-action ((act Receive-Action))
   (add-to-mailbox "In" (message act))
   (setf (succeeded act) t))

(defmethod actions-equalp ((act1 Receive-Action) (act2 Receive-Action))
   (declare   (ignore act1)
              (ignore act2))
   t)

(defmethod  make-copy  ((act  Receive-Action)  (msg  Message))  (make-instance
   'Receive-Action :message msg))

(defmethod present ((act Receive-Action))
     (format nil "~&Received message: ~A" (subject-string (message act))))

(defmethod stuff-appleevent ((action Receive-Action) reply)
   (print "That's a neat trick.  How did you predict a receive action?"))

;;------------------------------------------------------------------------
(defclass Delete-Action (Action)
   ((name
     :allocation :class
     :initform "Delete"
     :accessor name
     :type string
     :documentation "Delete the message")))

(defmethod do-action ((act Delete-Action))
   (delete-message (message act))
   (setf (succeeded act) t))

(defmethod actions-equalp ((act1 Delete-Action) (act2 Delete-Action))
   (declare   (ignore act1)
              (ignore act2))
        t)

(defmethod make-copy ((act Delete-Action) (msg Message))
    (make-instance 'Delete-Action :message msg))

(defmethod present ((act Delete-Action))
    (format nil "~&Delete message: ~A" (subject-string (message act))))
```

```
(defmethod stuff-appleevent ((action Delete-Action) reply)
  (ae-put-parameter-type reply :|type| :|del |))

;;---------------------------------------------------------------------
(defclass Remove-Action (Action)
  ((name
     :allocation :class
     :initform "Remove"
     :accessor name
     :type string
     :documentation "Remove a message from a mailbox")
    (mailbox
     :initarg :mailbox
     :accessor mailbox
     :type string
     :documentation "The mailbox to remove it from")))

(defmethod do-action ((act Remove-Action))
  (remove-message (message act) (mailbox act))
  (setf (succeeded act) t))

(defmethod actions-equalp ((act1 Delete-Action) (act2 Delete-Action))
  (eq (mailbox act1) (mailbox act2)))

(defmethod make-copy ((act Remove-Action) (msg Message))
  (make-instance 'Remove-Action
     :message msg
     :mailbox (mailbox act)))

(defmethod present ((act Remove-Action))
  (format nil "~&Remove message: ~A from folder: ~A"
          (subject-string (message act))
          (to-mailbox act)))

;;----------------------------------------------------------------------
(defclass Forward-Action (Action)
  ((name
     :allocation :class
     :initform "Forward"
     :accessor name
     :type string
     :documentation "Forward a message to recipients")
    (to-list
     :initarg :to-list
     :accessor to-list
     :type list
     :documentation "Who we're directing this message to")))

(defmethod do-action ((act Forward-Action))
  (resend-message (message act) :address (to-list act))
  (setf (succeeded act) t))

(defmethod actions-equalp ((act1 Forward-Action) (act2 Forward-Action))
   (declare   (ignore act1)
              (ignore act2))
    t)

(defmethod make-copy ((act Forward-Action) (msg Message))
   (make-instance 'Forward-Action
      :message msg
      :to-mailbox (to-list act)))

(defmethod present ((act Forward-Action))
   (format nil "~&Forward message: ~A to: ~A"
                (subject-string (message act)) (to-list act)))

;;-----------------------------------------------------------------------
(defclass Reply-Action (Action)
   ((name
      :allocation :class
      :initform "Reply"
      :accessor name
      :type string
      :documentation "Reply to message")))

(defmethod do-action ((act Reply-Action))
   (reply-message (message act))
   (setf (succeeded act) t))

(defmethod actions-equalp ((act1 Reply-Action) (act2 Reply-Action))
   (declare   (ignore act1)
              (ignore act2))
    t)

(defmethod make-copy ((act Reply-Action) (msg Message))
   (make-instance 'Reply-Action :message msg))

(defmethod present ((act Reply-Action))
   (format nil "~&Reply to message: ~A"
              (subject-string (message act))))

;;-------------------------------------------------------------------------
(defclass Copy-Action (Action)
   ((name
      :allocation :class
      :initform "Copy"
      :accessor name
      :type string
      :documentation "Copy a message to another folder")
    (to-mailbox
      :initarg :to-mailbox
      :accessor to-mailbox
      :type string
      :documentation "The mailbox to copy to")))

(defmethod do-action ((act Copy-Action))
   (copy-message (message act) (to-mailbox act))
```

```
  (setf (succeeded act) t))

(defmethod actions-equalp ((act1 Copy-Action) (act2 Copy-Action))
  (eq (to-mailbox act1) (to-mailbox act2)))

(defmethod make-copy ((act Copy-Action) (msg Message))
  (make-instance 'Copy-Action
    :message msg
    :to-mailbox (to-mailbox act)))

(defmethod present ((act Copy-Action))
  (format nil "~&Copy message: ~A to folder: ~A"
          (subject-string (message act)) (to-mailbox act)))

;;----------------------------------------------------------------------
(defclass Read-Action (Action)
  ((name
    :allocation :class
    :initform "Read"
    :accessor name
    :type string
    :documentation "Read a message")))

(defmethod do-action ((act Read-Action))
  (setf (succeeded act) t))

(defmethod actions-equalp ((act1 Read-Action) (act2 Read-Action))
  (declare (ignore act1)
           (ignore act2))
           t)

(defmethod make-copy ((act Read-Action) (msg Message))
  (make-instance 'Read-Action :message msg))

(defmethod present ((act Read-Action))
    (format nil "~&Read message: ~A" (subject-string (message act))))

(defmethod stuff-appleevent ((action Read-Action) reply) (ae-put-parameter-
  type reply :|type| :|read|))

;;======================================================================
(defvar *global-action-list* nil)
(defun init-action-list ()
  (setf *global-action-list* nil)
  (push (make-instance 'Reply-Action) *global-action-list*)
  (push (make-instance 'Forward-Action) *global-action-list*)
  (push (make-instance 'Remove-Action) *global-action-list*)
  (push (make-instance 'Delete-Action) *global-action-list*)
  (push (make-instance 'File-Action) *global-action-list*)
  (push (make-instance 'Receive-Action) *global-action-list*)
  (push (make-instance 'Read-Action) *global-action-list*))
```

eudora.lisp - AppleEvent handlers for Eudora messages

```
;-*- Mode: Lisp; Package: CCL -*-
;;-----------
;;Eudora Test

(require :appleevent-toolkit)
(defvar *message-store* (make-hash-table :test #'equalp))

(defmacro get-msg (unique_id)
"Gets a message from the message store.  Keyed by unique id."
  `(gethash ,unique_id *message-store* nil))

(defmethod newmsg-handler ((a application) theAppleEvent reply handlerRefcon)
  (declare (ignore handlerRefcon))
  (let*    ((unique_id (ae-get-parameter-longinteger theAppleEvent :|u_id|))
           (prev-msg (get-msg unique_id))
           (message (make-instance 'Message
                 :to (string-to-list
                    (ae-get-parameter-char theAppleEvent :|to  |))
                 :cc (string-to-list
                    (ae-get-parameter-char theAppleEvent :|cc  |
                 :from (string-to-list
                    (ae-get-parameter-char theAppleEvent :|from|))
                 :subject (string-to-list
                    (ae-get-parameter-char theAppleEvent :|subj|))
                 :body "This is a Eudora message"))
         (suggestion (UI-receive-message message)))
         ; Now put the message in the unresolved list, and try to finish it
         ; Note this may overwrite any old message w/this unique id.  That's
         ; ok.
    (setf (get-msg unique_id) message)
    ; Send a reply
    (ae-put-parameter-longinteger reply :|u_id| unique_id)
    (if suggestion
      (stuff-appleevent (action suggestion) reply)
      (ae-put-parameter-type reply :|type| :|null|))))

(install-appleevent-handler :|edra| :|nmsg| #'newmsg-handler)

(defmethod movemsg-handler ((a application) theAppleEvent reply handlerRefcon)
  (declare (ignore handlerRefcon))
  (let* ((frombox (ae-get-parameter-char theAppleEvent :|obox|))
         (tobox (ae-get-parameter-char theAppleEvent :|dbox|))
         (unique_id (ae-get-parameter-longinteger theAppleEvent :|u_id|))
         (message (get-msg unique_id))
         (suggestion (UI-move-message message frombox tobox)))
    (ae-put-parameter-longinteger reply :|u_id| unique_id)
    (if suggestion      (stuff-appleevent (action suggestion) reply)
      (ae-put-parameter-type reply :|type| :|null|))))

(install-appleevent-handler :|edra| :|refl| #'movemsg-handler)

(defmethod readmsg-handler ((a application) theAppleEvent reply handlerRefcon)
  (declare (ignore handlerRefcon))
  (let* ((unique_id (ae-get-parameter-longinteger theAppleEvent :|u_id|))
         (message (get-msg unique_id))
         (suggestion (UI-read-message message)))
    (ae-put-parameter-longinteger reply :|u_id| unique_id)
    (if suggestion
      (stuff-appleevent (action suggestion) reply)
      (ae-put-parameter-type reply :|type| :|null|))))

(install-appleevent-handler :|edra| :|read| #'readmsg-handler)

(defmethod accept-suggestion-handler
    ((a application) theAppleEvent reply handlerRefcon)
  (declare (ignore handlerRefcon)
           (ignore reply))
  (activate-agent (ae-get-parameter-longinteger theAppleEvent :|exp |)))

(install-appleevent-handler :|edra| :|sack| #'accept-suggestion-handler)

(defmethod getsugg-handler ((a application) theAppleEvent reply handlerRefcon)
  (declare (ignore handlerRefcon))
  (let* ((unique_id (ae-get-parameter-longinteger theAppleEvent :|u_id|))
         (message (get-msg unique_id))
         (reaction (last-reaction message))
         (suggestion (do-reaction reaction message)))
     ; Send a reply
    (ae-put-parameter-longinteger reply :|u_id| unique_id)
    (if suggestion
      (stuff-appleevent (action suggestion) reply)
      (ae-put-parameter-type reply :|type| :|null|))))

(install-appleevent-handler :|edra| :|sugg| #'getsugg-handler)
```

Appendix B

Additional Eudora Code

agent-events.h - Header file for agent-events.c

```
/* Header file for generic agent notification routines
 * All source code in this file was written by Max E. Metral
 * MIT Media Lab - 1993
 * Initial Apple Event code from bonura@apple
 */
void InformLispOfNewMail(TOCHandle tocH);
void InformLispOfReadMail(TOCHandle tocH, int sumnum);
void InformLispOfRefile(TOCHandle fromtocH, TOCHandle totocH, long u_id);
void RelinkAgent();
void AddMessageFieldsToAEvt(AppleEvent *theEvent, TOCHandle tocH, int sumnum);
Boolean PutUpSuggestion(Str255 suggestion);
void HandleSuggestiveReply(AppleEvent *theReply);
int FindMsgByUid(long int u_id, TOCPtr *toc);
void FeedAgentEveryMessage();
pascal Boolean myIdleProc( EventRecord *theEvent, long *sleeptime,
       RgnHandle *mouseRgn);
```

agent-actions.h - Header file agent-actions.c

```
/* Header file for generic agent action routines
 * All source code in this file was written by Max E. Metral
 * MIT Media Lab - 1993
 * Initial Apple Event code from bonura@apple
 */

void DoReadAction(long int u_id);
void DoRefileAction(long int u_id, UPtr tobox);
```

agent-events.c - Agent Notification Routines

```
#define FILE_NUM 50
/* Generic agent notification routines
 * All source code in this file was written by Max E. Metral
 * MIT Media Lab - 1993
 * Initial Apple Event code from bonura@apple
 */
#pragma load EUDORA_LOAD
#include <Processes.h>
#include <PPCToolbox.h>
#include "agent-events.h"
#include "agent-actions.h"

static TargetID* theTargetID = NULL;

/* Informs LISP of NewMail Arrival.  The unique_id param is a unique
```

```
 * identifier for this message situation.  Right now, it's the receive
 * date for the message.  This is not "correct" and should be thought
 * about more.
 */
void InformLispOfNewMail(TOCHandle tocH)
{
	AEAddressDesc		addrDesc;
	OSErr			myErr;
	AppleEvent		theEvent , theReply;
	char holder[255];
	int i;

	if (theTargetID == NULL) {
		RelinkAgent();
	}

	for( i = (*tocH)->count; /* ((*tocH)->sums[i-1].state & UNREAD) && */
i>0;
		i--) {
		myErr = AECreateDesc (typeTargetID, theTargetID,
			sizeof(*theTargetID), &addrDesc);
		myErr = AECreateAppleEvent ('edra' ,  'nmsg' , &addrDesc,
			kAutoGenerateReturnID, kAnyTransactionID,
			&theEvent);
		AddMessageFieldsToAEvt(&theEvent, tocH, i-1);

		myErr = AESend (&theEvent , &theReply , kAEWaitReply,
			kAENormalPriority, kAEDefaultTimeout, &myIdleProc, NULL);
		AEDisposeDesc(&addrDesc);
		HandleSuggestiveReply(&theReply);
		}
}

void InformLispOfReadMail(TOCHandle tocH, int sumnum)
{
	AEAddressDesc		addrDesc;
	OSErr			myErr;
	AppleEvent		theEvent , theReply;

	if (theTargetID == NULL) {
		RelinkAgent();
	}

	myErr = AECreateDesc (typeTargetID, theTargetID,
		sizeof(*theTargetID), &addrDesc);
	myErr = AECreateAppleEvent ('edra' ,  'read' , &addrDesc,
		kAutoGenerateReturnID, kAnyTransactionID,
		&theEvent);
	myErr = AEPutParamPtr(&theEvent, 'u_id', typeLongInteger,
		&((*tocH)->sums[sumnum].seconds), sizeof(long int));

	myErr = AESend (&theEvent , &theReply , kAEWaitReply,
		kAENormalPriority, kAEDefaultTimeout, &myIdleProc, NULL);
	AEDisposeDesc(&addrDesc);
```

```
        HandleSuggestiveReply(&theReply);
        return;
}

void InformLispOfRefile(TOCHandle fromtocH, TOCHandle totocH, long u_id)
{
        AEAddressDesc       addrDesc;
        AEDesc              theAEholder;
        OSErr               myErr;
        AppleEvent          theEvent , theReply;
        TOCPtr myTocTest;

        if (theTargetID == NULL) {
                RelinkAgent();
        }

        myErr = AECreateDesc (typeTargetID, theTargetID,
                sizeof(*theTargetID), &addrDesc);
        myErr = AECreateAppleEvent ('edra' ,  'refl' , &addrDesc,
                kAutoGenerateReturnID, kAnyTransactionID,
                &theEvent);
        myErr = AEPutParamPtr(&theEvent, 'u_id', typeLongInteger,
                &u_id, sizeof(long int));
        myErr = AECreateDesc(typeChar, &((*fromtocH)->name[1]),
                strlen(&((*fromtocH)->name[1])), &theAEholder);
        myErr = AEPutParamDesc(&theEvent, 'obox', &theAEholder);
        myErr = AECreateDesc(typeChar, &((*totocH)->name[1]),
                strlen(&((*totocH)->name[1])), &theAEholder);
        myErr = AEPutParamDesc(&theEvent, 'dbox', &theAEholder);

        myErr = AESend (&theEvent , &theReply , kAEWaitReply,
                kAENormalPriority, kAEDefaultTimeout, NULL, NULL);
        AEDisposeDesc(&addrDesc);
        AEDisposeDesc(&theAEholder);
        HandleSuggestiveReply(&theReply);
        return;
}

void AskForSuggestion(void)
{
        WindowPtr qdWin = FrontWindow();
        int kind, i;
        TOCHandle tocH;
        AEDesc addrDesc;
        AppleEvent theEvent, theReply;
        OSErr myErr;

        /*
         * figure out just what's in front
         */
        if (qdWin==(WindowPtr)(-1)) {qdWin = nil;}
        if (qdWin==nil)
                return;
        else
```

```
            kind = ((WindowPeek)qdWin)->windowKind;
        if (kind != MBOX_WIN)      /* Not a mailbox */
            return;
        tocH = (TOCType **)((WindowPeek)qdWin)->refCon;
        for( i = (*tocH)->count; i>0; i--) {
          if ((*tocH)->sums[i-1].selected) {
            myErr = AECreateDesc (typeTargetID, theTargetID,
                  sizeof(*theTargetID), &addrDesc);
            myErr = AECreateAppleEvent ('edra' ,  'sugg' , &addrDesc,
                  kAutoGenerateReturnID, kAnyTransactionID,
                  &theEvent);
            myErr = AEPutParamPtr(&theEvent, 'u_id', typeLongInteger,
                  &((*tocH)->sums[i-1].seconds), sizeof(long int));

            myErr = AESend (&theEvent , &theReply , kAEWaitReply,
                  kAENormalPriority, kAEDefaultTimeout, &myIdleProc, NULL);
            AEDisposeDesc(&addrDesc);
            HandleSuggestiveReply(&theReply);
            }
          }
        return;

}

void RelinkAgent()
{
            Str255 prompt;
            OSErr myErr;
            PortInfoRec thePortInfoRec;

            strcpy(prompt, "\pChoose an agent to link to:");
            theTargetID = NewPtr(sizeof(TargetID));

            myErr = PPCBrowser(prompt, NULL, FALSE,
                  &(theTargetID->location), &thePortInfoRec,
                  NULL, '');
            theTargetID->name = thePortInfoRec.name;

            if (myErr != noErr) {
                  DisposePtr(theTargetID);
                  return;
            }
}

void AddMessageFieldsToAEvt(AppleEvent *theEvent, TOCHandle tocH, int sumnum)
{
        long size;
        MSumType *msg = &((*tocH)->sums[sumnum]);
        UPtr msg_buf = NewPtr(msg->length+1), hdr;
        OSErr myErr;

        /* Add the from field */
        myErr = AEPutParamPtr(theEvent, 'from', typeChar,
              (Ptr) msg->from+1, strlen(msg->from+1));
        /* Now the subject field */
        myErr = AEPutParamPtr(theEvent, 'subj', typeChar,
                (Ptr) (msg->subj+1), strlen(msg->subj+1));

        /* Unique id is generated by the seconds that it arrived */
        myErr = AEPutParamPtr(theEvent, 'u_id', typeLongInteger,
                &(msg->seconds), sizeof(long int));

        /* To handle the other header fields we have to parse them */
        ReadMessage(tocH, sumnum, msg_buf);
        size = msg->length;
        /* Steve fails to tell us that the second arg is a Pascal String */
        /* Then again, I'm knew to this whole Mac thing, maybe I should known
*/
        hdr = FindHeaderString(msg_buf, "\pCc:", &size);
        myErr = AEPutParamPtr(theEvent, 'cc  ', typeChar, hdr, size);
        size = msg->length;
        hdr = FindHeaderString(msg_buf, "\pTo:", &size);
        myErr = AEPutParamPtr(theEvent, 'to  ', typeChar, hdr, size);
        DisposPtr(msg_buf);
}

/* FeedAgentEveryMessage - Feed all message to the agent.  Doesn't do
 * inbox or trash, but everybody else.  For non-inbox messages, creates
 * the message in the inbox and then sends a refile message
 */
void FeedAgentEveryMessage()
{
        return;
}

Boolean PutUpSuggestion(Str255 suggestion)
{
        int btn;

        ParamText(suggestion, 0, 0, 0);
        btn = Alert(AGENT_SUGGEST_ALRT, NULL);
        if (btn == 4)
                return (TRUE);
        return (FALSE);
}

void HandleSuggestiveReply(AppleEvent *theReply)
{
        Str255 suggestion;
        long type;
        Size sz = 0L;
        long int fakeBool;

        AEGetParamPtr(theReply, 'type', typeType, NULL,
                &type, sizeof(long), NULL);
        if (type == 'null')
                return;
        AEGetParamPtr(theReply, 'text', typeChar, NULL,
                &(suggestion[1]), 255, &sz);
        suggestion[0] = (char) sz;
        if (!sz)
                return;

        /* If we make it here, the agent has a suggestion */
        fakeBool = (long int) PutUpSuggestion(suggestion);

        {
                /* Now we tell the agent what happened */
                AEAddressDesc       addrDesc;
                OSErr               myErr;
                AppleEvent          theEvent , theOReply;

                if (theTargetID == NULL) {
                        RelinkAgent();
                }

                myErr = AECreateDesc (typeTargetID, theTargetID,
                        sizeof(*theTargetID), &addrDesc);
                myErr = AECreateAppleEvent ('edra' ,  'sack' , &addrDesc,
                        kAutoGenerateReturnID, kAnyTransactionID,
                        &theEvent);
                myErr = AEPutParamPtr(&theEvent, 'exp ', typeLongInteger,
                        &fakeBool, sizeof(long int));

                myErr = AESend (&theEvent , &theOReply , kAENoReply,
                        kAENormalPriority, kNoTimeOut, NULL, NULL);
                AEDisposeDesc(&addrDesc);
        }
        if (fakeBool) {
                long int u_id;

                /* Do the action (This should be some sort of switch statement)
*/
                AEGetParamPtr(theReply, 'u_id', typeLongInteger, NULL,
                        &u_id, sizeof(long int), &sz);
                switch (type) {
                case 'read':
                        DoReadAction(u_id);
                        break;
                case 'file': {
                        unsigned char tobox[255];
                        AEGetParamPtr(theReply, 'dbox', typeChar, NULL,
                                &(tobox[1]), 254, &sz);
                        if (!sz)
                                return;
                        tobox[0] = (char) sz;
                        DoRefileAction(u_id, tobox);
                        break;
                        }
                }
        }

        return;
}

/* Find a message in the TOCList by its UID.  Return the offset
 * of the summary in the TOC, and put the TOC it's in into toc
 */
int FindMsgByUid(long int u_id, TOCPtr *toc)
{
        int i;
        TOCHandle ptr;

        for( ptr = TOCList; ptr != NULL; ptr = (*ptr)->next ) {
                for(i = 0; i<(*ptr)->count; i++) {
                        if (((*ptr)->sums[i]).seconds == u_id) {
                                *toc = ptr;
                                return (i);
                        }
                }
        }
        return (NULL);
}

pascal Boolean myIdleProc( EventRecord *theEvent, long *sleeptime,
        RgnHandle *mouseRgn)
{
        Boolean gotKbd;
        EventRecord testEvent;
        char key;

        gotKbd = GetOSEvent( keyDownMask, &testEvent);
        if (gotKbd) {
                key = testEvent.message & charCodeMask;
                if ( (key == '.') && (testEvent.modifiers & cmdKey) )
                        return (true);
        }
        return (false);
}
```

agent-actions.c - Action Execution Functions

```
#define FILE_NUM 51
/* Generic agent notification routines
 * All source code in this file was written by Max E. Metral
 * MIT Media Lab - 1993
 * Initial Apple Event code from bonura@apple
 */

#pragma load EUDORA_LOAD
#include "agent-events.h"
#include "agent-actions.h"
```

```
void DoReadAction(long int u_id)
{
	TOCHandle theToc;
	int sumnum;
	MyWindowPtr w;

	sumnum = FindMsgByUid(u_id, &theToc);
	w = GetAMessage(theToc, sumnum, nil, True);
	NotUsingWindow(w);
}

void DoRefileAction(long int u_id, UPtr name)
{
	TOCHandle fromtoc;
	int sumnum;

	sumnum = FindMsgByUid(u_id, &fromtoc);
	MoveMessage(fromtoc, sumnum, MyDirId, name, false);
	return;
}
```

main.c - Modification to the Main Source File

```
/**********************************************************************
 * NotifyNewMail - notify the user that new mail has arrived, via the
 * notification manager.
 **********************************************************************/
void NotifyNewMail(void)
{
	TOCHandle tocH;

	if (PrefIsSet(PREF_NEW_SOUND)) NewMailSound();

	/* Max M. 4/93 - changed to inform lisp of new mail */
	InformLispOfNewMail(tocH=GetInTOC());
	if (!PrefIsSet(PREF_NO_OPEN_IN) && (tocH))
	{
			short i=(*tocH)->count;
			while (i--) if ((*tocH)->sums[i].state!=UNREAD) break;
			i++;
			RedoTOC(tocH);
			ShowMyWindow((*tocH)->win);
			SelectBoxRange(tocH,i,i,False,-1,-1);
			ScrollIt((*tocH)->win,0,-INFINITY);
			SelectWindow((*tocH)->win);
#ifdef NEVER
		}
		else
		{
			UpdateMyWindow((*tocH)->win);
		  ScrollIt((*tocH)->win,0,-INFINITY);
		}
#endif
	}
	if (InBG && PrefIsSet(PREF_NEW_ALERT)||!PrefIsSet(PREF_NO_APPLE_FLASH))
	{
		if (MyNMRec) return;		/* already done */
		MyNMRec = New(struct NMRec);
		if (!MyNMRec) return;	/* couldn't allocate memory (bad) */
		WriteZero(MyNMRec,sizeof(*MyNMRec));
		MyNMRec->qType = nmType;
		MyNMRec->nmMark = 1;
		MyNMRec->nmRefCon = TickCount();
		if (!PrefIsSet(PREF_NO_APPLE_FLASH))
			MyNMRec->nmIcon = GetResource('SICN',FLAG_SICN);
		if (InBG && PrefIsSet(PREF_NEW_ALERT))
		{
			Str255 scratch;
			GetRString(scratch,NEW_MAIL);
			MyNMRec->nmStr = NuPtr(*scratch+1);
			if (MyNMRec->nmStr) PCopy(MyNMRec->nmStr,scratch);
		}
		if (NMInstall(MyNMRec))
		{
			DisposPtr(MyNMRec->nmStr);
			DisposPtr(MyNMRec);
			MyNMRec = nil;
		}
	}
	if (!InBG && PrefIsSet(PREF_NEW_ALERT))
	{
		AlertTicks=GetRLong(ALERT_TIMEOUT)*60+TickCount();
		(void) ReallyDoAnAlert(NEW_MAIL_ALRT,Note);
		AlertTicks = 0;
	}
}
```

message.c - Modifications to the Message Handling Routines

```
/************************************************************************
 * GetAMessage - grab a message
 ************************************************************************/
MyWindowPtr GetAMessage(TOCType **tocH,int sumNum,MyWindowPtr win, Boolean
showIt)
{
	MessHandle messH;
#ifdef DEBUG
	if (BUG6) DebugStr("\p;hc;g");
#endif
	if (!tocH || (*tocH)->count<=sumNum) return(nil);
	if (messH = (*tocH)->sums[sumNum].messH)
	{
		if (showIt)
		{
			if (!(*messH)->win->qWindow.visible)
				ShowMyWindow((*messH)->win);
			SelectWindow((*messH)->win);
```

```
                }
                UsingWindow((*messH)->win);
                return((*messH)->win);
        }
        else if ((*tocH)->which==OUT)
                return(OpenComp(tocH,sumNum,win,showIt));
        else {
                /* Max M. 4/93 - Tell Lisp that the user is about to read mail */
                MyWindowPtr retval = OpenMessage(tocH,sumNum,win,showIt);
                InformLispOfReadMail(tocH, sumNum);
                return(retval);
        }

        }
```