Using Software Agents for Facilitating Access to On-Line Resources

Kave Eshghi
External Research
HPL-93-82
September, 1993

software agents,
agent oriented
programming,
abduction,
distributed problem
solving

A new agent programming methodology, called Abductive Agent Oriented Programming, is introduced. This methodology builds on Shoham's Agent Oriented Programming paradigm and the abductive reasoning work in Logic Programming. The application of this paradigm to the development of semi-autonomous software agents is discussed.

Internal Accession Date Only

1 Introduction

In this document we discuss the Hewlett Packard Labs-Stanford University Agent Oriented Programming Project, its achievements and lessons, and the next step after this project.

The objective of the project was to investigate the application of the Agent Oriented Programming paradigm to the development of software agents, in particular in the office environment. We aimed to show, through a representative application, how this technology can be used to facilitate the design and implementation of intelligent software agents, and to provide a higher level metaphor for human computer interaction.

2 The Need for Software Agents

Computer systems are getting more complex. In networked, distributed environments, this complexity is the main barrier to the efficient and cost effective use of computerized resources by non-specialist users. As network based services spread and diversify, becoming available at home as well as in the office, this problem is going to get more acute. Some of the main causes of the complexity barrier faced by users are as follows:

2.1 Ignorance of the Existence of Resources

Often, users are unaware of the existence of resources which may be directly relevant to solving the problem at hand. Examples of such resources include:

- On-line internal databases containing corporate, business and other data,
- Communication facilities such as on-line fax, email, voicemail,
- Wire service reports, stock exchange data, specialized outside databases, on-line library search
- Mailing lists, electronic conferencing, and net-news facilities,
- Repositories of scientific papers, ftp sites, on-line journals
- Automatic system management tools (ninstall, update etc.) and facilities
- On-line services such as travel agents, banking facilities etc.

Often, faced with a problem the user will not use the more efficient on-line service, simply because he is not aware of its existence.

2.2 Lack of Specialized Knowledge

Even when users are aware of the existence of on-line resources which are relevant to their problem, they lack the expertise to access and use these resources. On-line services have grown in an ad-hoc fashion, and each one has been designed with its own, often idiosyncratic user interface. Thus an average user, who may use each one of these services only occasionally, is deterred from using them by the need to master the intricacies of each one from scratch. This problem is exacerbated by the lack of

reasonable on-line help facilities and obscure, hard to access documentation. For example, even though many users are aware of the existence of on-line library search facilities, the burden of learning to use this facility (among many other such facilities) is such that they often don't bother.

2.3 Using On-Line Services Is Hard Work

Accessing and using network based services is often a long winded process, involving logging on to remote computers, navigating through often hostile user interfaces, and searching for, manipulating and moving information manually. For example, suppose you are looking for an electronic copy of a paper by a given author. Here are the steps you need to take to get it:

1. Somehow, find the address of an ftp site that may contain this paper. Frequently, this involves wading through a lot of different possible sources of information (FAQ files, personal notes, pleading on the net-news etc.).
2. Having found a number of likely ftp sites, logging on to them, and searching through their directory structure.
3. Having located the paper on one of the sites, transferring it using ftp (if you are on a closed subnet machine, and the paper is on an external machine, this is a two stage process). Having got it on your machine, decompressing and printing it.

The effort involved, which is multiplied by the lack of specialized knowledge about the given service, is a real obstacle to the use of such services.

3 Centralized Repository of Knowledge about Network Based Resources

The obvious solution for bridging the knowledge gap with respect to the use of networked resources might seem to be to provide an on-line service which keeps track of all the resources on the network, and provides users with the information necessary for the use of these resources. While this would be a step forward from the current situation, the deployment of such a centralized repository has several problems of its own:

1. Computer networks and the resources maintained on them are dynamic, loosely managed entities. Keeping a centralized repository of knowledge creates difficult problems of management and control, and assumes a tightly organized structure for maintaining the faithfulness of the central repository.
2. The scope of computer networks exceeds any given organization and structure. It seems hardly possible to have a centralized knowledge base about all the resources available on the internet, for example.
3. A centralized knowledge base would become a bottleneck if it is to be used by all those who need access to the network-based resources.

4. At best, the centralized knowledge base will provide the information about how to access and use the resources; the user still has to do all the work in accessing and using the resource. Also, the user will have to cope with all the error conditions and exceptions that arise, since it is unlikely that the central knowledge base will have information about all the possible error situations.

4 Using Semi-Autonomous Agents As Resource Custodians

A radical solution to the complexity barrier presented by network based resources is the use of semi-autonomous, high level software agents. In this model, each resource on the network would have an agent as the custodian. The agent's knowledge base will contain information about the client resource, its internal language, its capabilities, and the requirements of its use. Thus each agent is a specialist about its client resource. The agent has a module in a low-level language tailored specifically as a mediator between the agent language and the resource's Application Programming Interface (API). Thus the agent can control the resource by driving it through its API. We call these types of agents application agents.

As well as application agents, there may be other agents which are not specifically associated with any application, but have knowledge about a range of resources and agents on the network. We call these the mediator agents.

Application agents and mediator agents know about other agents, and use the high-level agent oriented language for communication with other agents. Clearly, in order to avoid the problems with the centralized model, we cannot expect every agent to know about every other agent on the network. We compensate for the partiality of agents' knowledge by using a cascaded model of goal satisfaction, as explained below.

Each user has a personal agent. Personal agents are special cases of mediator agents, which are dedicated to one user. The user communicates with his agent using a 'humanized' version of the agent communication language, for example using forms, canned dialog formats, etc.

The user can communicate with his agent using the full range of message types provided by the agent communication language, including informing and requesting. When the user needs to solve a problem, he would initialize a dialog with his personal agent. This dialog would eventually lead to the establishment of a number of goals for the agent, which will then strive to satisfy them using the distributed problems solving model.

5 Distributed Problem Solving with Agents

Agents have goals which they will try to satisfy. Goals are either persistent goals built into the agent by the designer, or transitory goals resulting from requests made by other agents. For example, a persistent goal of the database agent is to maintain the integrity of the database contents; a transitory goal for the same agent can be to answer a specific query using the data base.

Transitory goals are communicated to agents through requests from other agents. In the distributed problem solving model advocated here, faced with a goal an agent can either

- Initiate a sequence of interactions with its client application which would satisfy the goal, or
- Make further requests to other agents (including mediator agents) which, if successful, will eventually lead to the satisfaction of the goal.

With this type of agent functionality, a community of agents will be able to satisfy most of the goals of any of its members without needing a centralized repository of information. This is achieved through the cascading and subdividing of requests until the original request is serviced through the cooperation of many agents.

For example, let us assume the user needs to locate a specific paper, and all he knows is the title and the name of the author. He can ask his personal agent to get a copy of this paper. The personal agent, which knows about the various ftp sites on the network and some of the mediator agents which might know about such sites, will get in touch with the relevant ftp and mediator agents. The ftp agents may not themselves have the requested paper on their local site, but they may know about other ftp sites which may have the paper. Also, some of the mediator agents may know about such ftp sites. Thus, in a cascading process, the original user agent will hopefully find an ftp site which has a copy of the requested paper. It can then open a dialog with that agent directly, arrange to transfer that paper to the user's machine, and get it printed on a local printer. All this can happen without the user having to supervise the process step by step.

6 Software Agents

Given the functionality outlined above, what are the necessary characteristics of the agent technology to be used in this context? Below we briefly outline some of the main features of the type of agent we have in mind:

1. Agents are knowledge based and goal driven. Their knowledge base allows them to maintain an explicit model of the aspects of the world which are relevant to their function, and their goal driven behavior ensures that they will behave in a way that will satisfy the persistent goals set for them by their designer and the transitory goals generated as a result of requests by other agents.
2. There is a high level notion of mental state attributable to agents. Elements of their mental state include beliefs, goals and decisions.
3. Agents are able to communicate with each other using a sufficiently high level language through which knowledge is transmitted and requests are made. Their mental state evolves as a result of receiving messages from outside in a coherent and well understood way. There is an underlying semantic framework relating the mental state of agents with the nature of the messages they receive.

4. They are capable of action, including transmission of messages to other agents and direct driving action with respect to their client applications. Their action follows transparently from their mental state.

A major focus of the project was the development of the kind of agent technology that can rise to this challenge.

7 Abductive Agent Oriented Programming

Abductive Agent Oriented Programming is a new agent programming framework developed during this project to facilitate the implementation of the type of agent functionality mentioned above. The language Agent Alpha, a programming language within this framework, has been implemented and used for the implementation of a prototypical distributed problem solving task, which we chose to be meeting scheduling. Abductive Agent Oriented Programming draws on two strands of work in artificial intelligence: Prof. Shoham's Agent Oriented Programming paradigm, and the abductive reasoning literature. Here, briefly, are the main outlines of these two strands:

7.1 Agent Oriented Programming

Agent Oriented Programming is a new programming paradigm developed by Prof. Shoham of Stanford University. It combines the main features of Object Oriented Programming with some of the latest results in theoretical artificial intelligence concerned with the nature of epistemic attitudes such as knowledge, belief, intention etc. Specifically, in Agent Oriented Programming

- Agents have mental state, which are propositional attitudes which can be attributed to them at a given time. These mental states include knowledge, belief and commitment. There is an underlying semantic framework based on epistemic logic which gives coherence to these mental states, and constrains the relationship between the various components of the mental state. For example, it is required that agents' beliefs should be consistent, that agents should believe that they will carry out their commitments etc.
- Agents communicate with each other through 'speech act' type primitives. Examples of such primitives are informing and requesting. There is a close relationship between the communication acts among agents and the evolution of their mental state. For example, in appropriate circumstances informing an agent of the fact X will lead that agent to believe X, requesting X from an agent will lead to that agent committing to X etc.
- Agents act on the basis of their mental state. If an agent is committed to doing X at time T, and this commitment persists until T, it will carry out X.

7.2 Abductive Reasoning Frameworks

Abduction is the process by which plausible explanations are found for observations. For example, suppose we know that b implies a and now observe a. Then b is a

plausible explanation for a, because assuming b will allow us to derive a using the rule b→a. Abduction has been used in diagnosis, story comprehension, planning etc.

Abduction is a form of non-monotonic reasoning, because hypotheses which are consistent with one state of a knowledge base may become inconsistent when new knowledge is added.

An abductive framework is a triple <P,Ab,I>where P is a Horn Clause program, Ab is a set of *abducible* predicates, and I is a set of integrity constraints. Given a set of goals G, a set Δ of ground atoms is a *solution* of G iff

- all the predicates in Δ are from Ab,
- For all g in G, $P \cup \Delta \mid -g$.
- $P \cup \Delta$ satisfies the integrity constraints

By definition, $P \cup \Delta$ satisfies the integrity constraints I if all sentences of I are true in the minimal model of $P \cup \Delta$. From this definition, it follows that Δ is a solution of G with respect to <P,Ab,I> iff $P \cup \Delta$ satisfies $I \cup G$

Example: Let the program P be

p←a

a←c

a←d

Let A= {c,d} and G={p}. Then Δ={c} is a solution of G, as is Δ={d}.

Abductive reasoning has mostly been explored in a static context: given an abductive framework, what is the solution for a given goal (this is the way abduction is used in diagnosis, for example). In the application of abductive reasoning to agent programming, we consider abductive reasoning from a dynamic point of view: how do the assumptions held by an agent change in time, as a result of interaction with other agents, and how is the notion of action integrated with the abductive point of view.

7.3 Using Abductive Frameworks As Agents

In Abductive Agent Oriented Programming, an agent is defined as an abductive framework <P,A,I>. In particular, the ground atomic sentences in I are called the goals of the agent. At each instant, the agent adopts a set of assumptions, Δ, which are ground atomic sentences whose predicate are in A. We call Δ the *decisions* of the agent. The mental state of the agent at each instant is defined by its abductive framework plus the set of decisions it has adopted. Its goals are the atomic sentences of I, its decisions are Δ, and its beliefs are the set of sentences provable from $P \cup \Delta$. We require that the agents mental state at each instant be coherent, in the sense that sense $P \cup \Delta$ should satisfy I. Furthermore, we require that the set of decisions be minimal with respect to coherence, i.e. there is no subset Δ' of Δ where the mental state arising from the adoption of Δ' alone is coherent.

Agents communicate with each other through the two primitives 'inform' and 'request'. Informing an agent of a fact F will result in the fact being added to the P

component of its mental state, and requesting an agent to do R will result in R being added to the agent's goals. Typically, to maintain coherence, the agent's decisions have to change after receiving a message. For example, when a message is received requesting R, the agent has to adopt sufficient decisions to make R true in the minimal model of $P \cup \Delta$. When a message is received informing F, inconsistency may result with the agent's current decisions, and as a result some decisions may have to be withdrawn, and others adopted to cater for the original goals which gave rise to the withdrawn decisions.

Certain predicates are defined as action predicates. When an atomic sentence with one of these predicates becomes true in the current mental state of the agent, the agent carries out the associated action. The two main action predicates are inform(X,Y) and request(X,Y). For example, if the sentence inform(tom, printer_down) becomes true in the current mental state of the agent, the message inform(tom, printer_down) is sent to the agent tom.

The main advantages of the abductive agent oriented programming approach is that it provides a natural integration the abductive reasoning approach with the main principles of agent oriented programming. In doing so, it enriches both fields:

- The work in abductive logic programming assumes a single, stationary abductive framework, and provides a basis for making assumptions which satisfy the constraints of this framework. Abductive agent oriented programming extends this paradigm by incorporating multiple, communicating agents, their states evolving in time according to a well defined dynamic between communication and internal state. This makes abductive logic programming a potent tool for such application areas as reactive planning agents, distributed problem solving etc.
- Agent oriented programming benefits from the abductive approach by inheriting the natural goal directed, clear semantics of abductive frameworks. The incorporation of logic programs plus integrity constraints gives agent oriented programming a high level of expressive power, and helps it inherit the rich body of results, algorithms and insights in the abductive logic programming field.

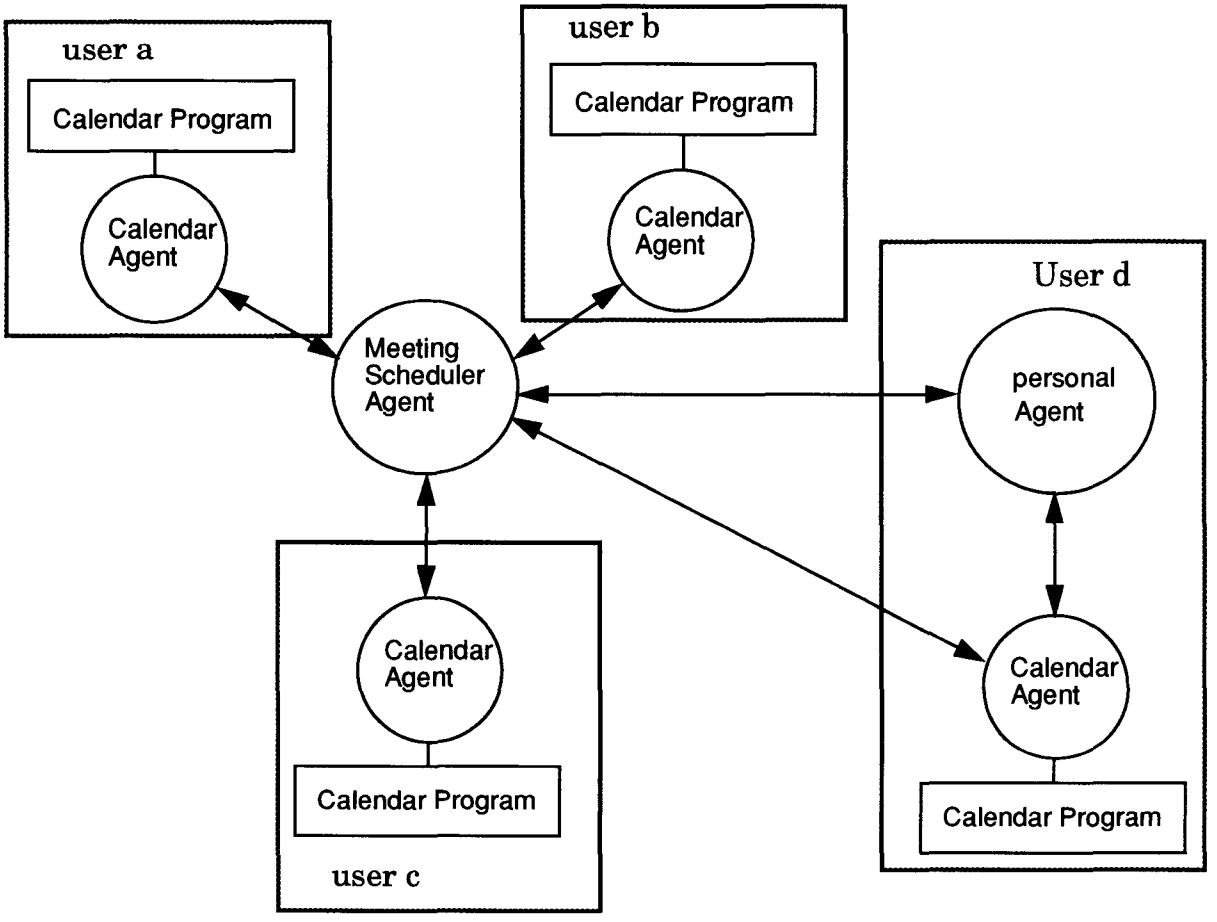
8 The Agent Alpha Language

In order to develop Abductive Agent Oriented Programming in a practical way, it was decided to implement an interpreter and associated communication and user interface modules for the pilot language Agent Alpha which embodies most of the principles of Abductive Agent Oriented Programming. Agent Alpha is a general purpose programming language for agent programming. It has built-in communication facilities for developing distributed problem solving applications, and incorporates the notions of mental state and communication discussed above. In order to keep track of the mental state of agents, a truth maintenance system is used which records the reasons for making decisions. The truth maintenance system ensures the

coherence of the agent's mental state, and deploys a minimal change strategy for revising agent's decisions when the need arises due to the receipt of new messages.

9 Meeting Scheduling as a Prototypical Application

In order to gauge the practicality and potency of Agent Alpha for developing software agents for distributed problem solving tasks, we decided to use meeting scheduling as a prototypical application. The top level model of the meeting scheduling system we built is as follows:



Each user on the network has a calendar program in which he keeps his engagements. The calendar program is managed by a calendar agent, which communicates with the calendar program through its API. There is a meeting scheduling agent which specializes in arranging meetings among the users on the network. When user d wants to organize a meeting among a number of other users, say a, b, c and d, he instructs his personal agent to do it for him, providing it with details such as the list

of participants in the meeting, suggested suitable times for the meeting, the duration of the meeting etc. The personal agent then gets in touch with the meeting scheduling agent, and makes a request for this meeting to be scheduled. The meeting scheduling agent contacts the calendar agents for a, b and c, requesting time periods that are suitable times for that user to hold the meeting. The calendar agent consults the calendar program, and gets a list of suitable times for this meeting, whereupon it replies to the meeting scheduling agent with the information. When the meeting scheduling agent receives a reply from each of the calendar agents, it tries to find a time which satisfies all the participants. If such a time can be found, then the it schedules the meeting, informing all the calendar agents, who then update the respective calendars.

If the first set of replies is such that there is no time which suits every participant, the scheduling agents enters a process of negotiation with the calendar agents, trying to find an acceptable time for the meeting. At the end of this process of negotiation, if a time is found that is acceptable to all participants, the scheduler will schedule the meeting. Otherwise, it gets back to the original requesting agent (in this case the personal agent for user d), and report its failure with the reasons for the failure.

9.1 Meeting Scheduling as a Testbed for Agent Alpha

As stated before, the reason for developing the meeting scheduling system was to exercise, using a practical application, those features of the Agent Alpha language which are important in building distributed problem solving agent networks. The following aspects of this application were of particular significance in this regard:

- Meeting scheduling is a typical distributed problem solving task, involving the coordination of a series of constraint satisfaction tasks by the different agents. As such, it presented a realistic challenge to be met by the Agent Alpha implementation.
- The meeting scheduling agent needs to solve its goals through a protracted process of negotiation with the calendar agents. This involves taking a process oriented view of goal satisfaction, keeping track of the asynchronous negotiation process, determining the next action to be performed in view of the current status of the various subgoals generated by the original goal and updating decisions depending on the progress of the negotiation process. The explicit nature of agent goals, beliefs and decisions and the transparent way in which these propositional attitudes interact with communication acts helped keep the structure of the agents simple and clear, which was a major design objective for Agent Alpha.
- The coupling of calendar agents to the calendar program is typical of the application agent-client application model mentioned earlier. This raises the issue of the low level interface to the calendar program, and the ability of the agent to drive the calendar application through this interface.

9.2 Meeting Scheduling in Practice

The first version of the meeting scheduling experiment assumed that all potential participants in the meeting had calendar programs and agents running. This proved to be an unrealistic assumption, so in the next version we dropped this assumption and relied on direct email to users who didn't have calendar agents, who then interacted with the scheduler agent directly through email, using forms. Technically, the experiment was a success; the agents performed as expected, and we used the system to schedule meetings among project members. Some interesting issues that arose in the practical application of the system were:

- In order to ensure that the negotiation process for determining a suitable time for meetings succeeds, it is necessary that all interactions among the participants go through the agents. In practice, the tendency for the human users to use informal direct communication among themselves can compromise the scheduling agent's ability to have all the necessary information for finding a suitable time.
- Although we tried hard to anticipate all the contingencies that can arise in the scheduling process, sometimes the dynamics of the process lead to situations in which the scheduling agent could not usefully proceed. This happened, for example, when people changed their mind about a meeting after they had given the agent their preferred free times.

Perhaps the biggest lesson of the meeting scheduling experiment was that successful distributed problem solving using current agent technology necessitates strictly predictable behavior on the part of all participant agents, including the handling of error and exception situations. With humans in the loop, this can be difficult to achieve.

10 The Achievements of the Project, and What Next

In this project, we concentrated on the development of an agent language and programming infrastructure for the development of agents. As such, we tried to provide one of the pre-requisites of building the network of agents that can collectively ease the difficulty of using network based on-line services.

This effort was successful, in the sense that we developed and implemented an agent programming language that was shown in practice to be a powerful tool for the development of semi-autonomous software agents. There is a lot more to be done to fulfill the vision of type of agent network which can remove the complexity barrier faced by ordinary users when trying to use on-line resources. Some of the main remaining tasks are:

- A larger, more ambitious experiment involving a network of application agents, mediator agents and user agents must be undertaken, to see if the kind of agent technology we developed can be scaled up to handle a large, heterogenous set of resources.

- Monitoring, debugging and other support software needs to be developed to ease the agent construction task.
- The human-agent interface needs to be considerably improved. This is a significant area of research, which was not attempted in a serious way by this project.

1 Appendix: The language Agent Alpha

This is a general overview of the Agent Alpha programming language, which is a member of the family of Agent Oriented Programming languages. The appendix is organized as follows: first we give a general overview of the main mechanisms and concepts in the language, and then we give a more detailed description of the language, including its syntax and primitives.

2 General Overview of Agent Alpha

2.1 Abduction

The semantics of agents in Agent Alpha is based on abduction. Abduction is the process by which plausible explanations are found for observations. For example, suppose we know the rule a←b and now observe a. Then b is a plausible explanation for a, because assuming b will allow us to derive a using the rule a←b.

In Agent Alpah, an abductive framework is a triple <P,Ab,G>where P is a Horn Clause program, Ab is a set of *abducible* predicates, and G is a set of ground atoms, called goals. A set Δ of ground atoms is a *solution* of <P,Ab,G> iff

- all the predicates in Δ are from Ab,
- $P \cup \Delta$ is consistent,
- For all g in G, $P \cup \Delta \vdash g$.

Example: Let the program P be

p←a

a←c

a←d

Let A= {c,d} and G={p}. Then Δ={c} satisfies <P,Ab,G>, as does Δ={d}

2.2 Agents As Abductive Frameworks

In Agent Alpha, each agent is essentially an abductive framework <P,Ab,G> where P is a set of Horn Clauses, Ab is a set of abducible predicates, and G is a set of goals. At each instant, the agent adopts a set of assumptions Δ which satisfies the abductive framework, as discussed above. Agents communicate through receiving and sending messages. There are two types of messages: inform messages, and request messages. Receiving a request message creates a new goal for the agent, i.e. a new atom is added to G. Receiving an inform message creates a new fact for the agent, i.e. a new atom is added to P. In general, receiving a message leads to a change in the set of adopted assumptions, because in general the old set of assumptions will no longer satisfy the new abductive framework.

The agent's mental state is defined as follows: its beliefs are the logical consequences of $P \cup \Delta$, its goals are G, and its decisions are Δ.

2.3 The Truth Maintenance System

Agent Alpha uses a truth maintenance system to keep track of mental state of agents. We define <R,J,A,G, T,Δ>, the *TMS-State* of the agent at a given time, as follows.

1. R is a set of propositions,
2. J is a set of propositional Horn clauses, called justifications. In general, justifications are clauses of the type p ←a∧b∧c..This set includes *nogoods*, which are clauses of the form false←a∧b∧c.., and *facts*, which are clauses of the form p← (i.e. conclusions without any conditions)
3. A is distinguished subset of R declared as abducible,
4. G is a distinguished subset of R declared as goals.
5. T is the subset of R all of whose elements have been assigned the value true,
6. Δ is the subset of A all of whose elements have been assigned the value true.

Agent Alpha relies on an underlying truth maintenance system to keep track of the TMS-states of agents, to maintain the consistency of their beliefs and to ensure that their goals are met in the current state. It is the truth maintenance system that assigns true or false to the propositions in R. The truth assignment satisfies two types of constraints: static constraints, and dynamic constraints.

2.4 Static Constraints on the TMS-State of Agents

1. The truth assignment to propositions is consistent with the justifications, i.e. there is no justification such that its antecedents are true and its consequent is false.
2. The truth assignment to non-abducible propositions is justified. This means that a non-abducible proposition is assigned true only if it is the consequent of a justification where all the antecedents are assigned true.
3. All the goals are assigned true (while satisfying condition 2 above)
4. The set of abducibles assigned true, i.e. Δ, is minimal in the following sense: if any member of Δ is assigned false (without assigning true to any other abducible) then there would be a goal which cannot be assigned true in a justified way.

The requirements 1 to 4 above can be summarized as follows: Δ is a minimal solution of the abductive framework <J,A,G>.

2.5 Dynamic Constraints on the TMS-State of Agents

When a new fact or goal is added to the state of an agent, the truth maintenance system ensures that a minimal number of assumptions is changed to accommodate the new situation. This can be expressed in the following way:

Let $\langle R_1,J_1,A,G_1, T_1,\Delta_1\rangle$ be the state of an agent at time t_1, and $\langle R_2,J_2,A,G_2, T_2,\Delta_2\rangle$ the state of the same agent after the addition of a fact or a goal. Roughly speaking, the following process describes the evolution of Δ_1 to Δ_2:

First, enough assumptions are added to Δ_1 such that there is a consistent subset of the resulting set of assumptions from which all the goals in G_2 are provable. Let us call this set Δ_{11}. Then enough assumptions are removed from Δ_{11} to make it a minimal set of assumptions such that the resultant truth assignment to propositions satisfies all the static constraints discussed in the previous section. The resulting set of abducibles is Δ_2.

3 The Need for Instantiation Algorithms

The TMS is a propositional device, and keeps track of the state of agents through propositional justifications. In Agent Alpha, agent programs are in general non propositional. To bridge the gap between the non-propositional rules in agent programs and propositional justifications used by the TMS, the Agent Alpha intherpreter uses various instantiation algorithms.

Let P be a set of non-propositional Horn Clauses. One way to generate propositional clauses from P is to instantiate all the variables in P to all possible ground terms in the language of P. This would create a set of fully ground Horn Clauses which is equivalent to P, and which is effectively propositional, because we can treat each $p(arg_1,arg_2,,arg_k)$ where $arg_1,arg_2,,arg_k$ are fully instantiated as a separate proposition.

The problem with this simple scheme is that the set of propositional clauses thus generated is infinite, because the number of ground terms is infinite. Thus we need a more selective approach to instantiation, to only generate those propositional clauses which are relevant to the task at hand.

Let L be a first order predicate language, and <P, Ab, G> an abductive framework where P, Ab and G are in L. Let ground(P) denote the set of ground clauses derived by instantiating the clauses in P over all the ground terms in L. Let atoms(Ab) denote the set of all ground atoms in L whose predicate occurs in Ab. Then the TMS-State <R,J,A,G,T,Δ> is called a *sufficient instantiation* of <P, Ab, G> iff

- For every clause $p \leftarrow q_1,q_2,..$ in J, there is a clause $p \leftarrow d_1,d_2,...q_1,q_2...$ in ground(P) where $d_1,d_2..$ are atoms which logically follow from P (we call these *direct* predicates, as discussed below)
- for any subset σ of R and any proposition q in R, $P \cup \sigma \vdash q$ iff $J \cup \sigma \vdash q$
- A is the intersection of R and atoms(Ab),
- For any subset δ of atoms(Ab) and any goal g in G, $P \cup \delta \vdash g$ iff $J \cup \delta \vdash g$

We call predicates whose definition do not in any way depend on abducibles to be *direct* predicates. For example, append(X,Y,Z), which computes the concatenation of two lists, can be such a predicate. The use of direct predicates ensures that we can keep the size of J managable, as the definition above excludes direct atoms from the requirement that they be provable from J. Without this exclusion, we would need an infinite number of clauses in J just to define append.

It is possible to prove that a set of assumptions Δ is a solution to the abductive problem <P,Ab,G> iff <R,J,A,G,T,Δ> is a sufficient instantiation of <P,Ab,G> and <R,J,A,G,T,Δ> satisfies the static constraints on TMS-States mentioned earlier.

The instantiation algorithm (discussed below) ensures that the set of propositions and justifications in TMS is such that the TMS-state for an all agents is always a sufficient instantiation of the agent's abductive framework.

4 Agent Communication

Agents communicate through two types of messages: request messages and inform messages. When the agent Agent2 is sent the message request(Agent2,R) by the agent Agent1, the atom comply(T, Agent1,R) is created, and added to the Agent2's goals (T is the time at which the message is received). When Agent2 is sent the message inform(Agent2,R), the atom assimilate(T,Agent1,R) is created, and added to the Agent2's program as a fact.

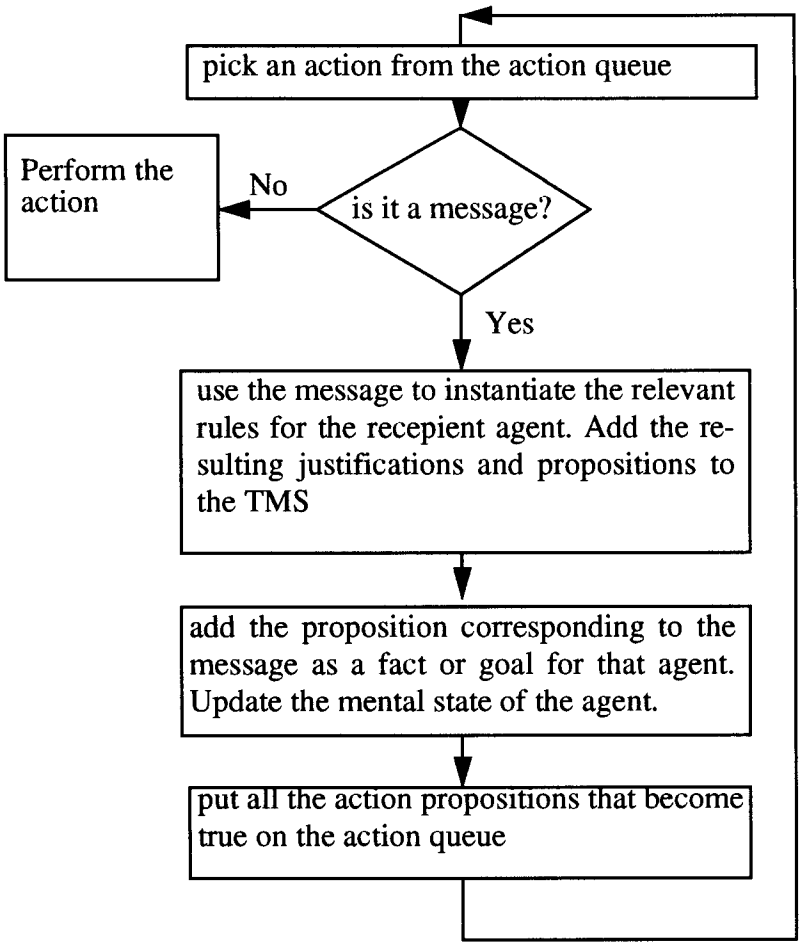
In general, as a result of adding a comply atom to the agent's goals, or adding an assimilate atom to the agent's facts, the agent's mental state will change. This happens in two stages:

1. Adding a new fact or goal in general means that the set of justifications stored by the TMS for this agent no is longer a sufficient instantiation of its rules. Thus, the instantiation algorithms are invoked to expand the set of justification the TMS holds for this agent.
2. Given the new set of justifications and the added fact or goal, the agent's mental state in general will no longer satisfy the static constraints mentioned earlier. Thus the TMS will revise the agent's mental state, arriving in a new state in the fashion described earlier.

If, in the agent's new mental state, a proposition of the form request(Agent,R) or inform(Agent,R) becomes true, a corresponding message is sent to Agent. Thus, in general, as a result of receiving a message, agents will send out other messages. This is how the mental state of a collection of agents communicating with each other evolves over time.

5 The Interpreter Cycle

The following diagram indicates the interpreter cycle for Agent Alpha.



Below we will describe the instatiation phase of the cycle in more detail.

6 The Instantiation Algorithm

As mentioned previously, the purpose of the instantiation algorithm is to make sure that the justifications held by the TMS are a sufficient instantiation of the agent rules with respect to the current facts and goals. This means that when, as a result of the receipt of a message, the facts or goals of the agent are expanded, it is necessary to instantiate some more rules.

For efficiency reasons, in Agent Alpha there are three types of rules: forward, backward, and direct. This is one among a number of syntactic restrictions which are used to make the instantiation process computationally efficient. Backward rules are written in the form *head<-body*. where *head* is an atom and *body* is a list of atoms. Forward rules are written as *body->head*. Direct rules are written as head:-body.

Below we describe, in some detail, how the instantiation algorithm works when a new goal, or a new fact, are added to the agent's abductive framework.

Terminology: Let D be a set of Horn Clauses and c a Horn Clause. We say that the ground clause $p \leftarrow q_1,q_2,...q_k$ is an instantiation c in the context of D iff for some $d_1,d_2,..d_n$, D |- $d_1,d_2,..d_n$ and $p \leftarrow q_1,q_2,...q_k,d_1,d_2,..d_n$ is an instantiation of c.

6.1 Instantiation of backward rules

Backward instantiation is started when a new goal is added to the agent's set of goals. Let B be the set of backward rules in the agent's program, D the set of direct rules, and atoms(A) the set of all ground atoms with an abducible predicate. Let g be the goal just added. The purpose of the backward instantiation algorithm is to find a set of justifications J_g where

- Each member of J_g is an instantiation of a rule in B in the context of D,
- For all subsets δ of atoms(A), $B \cup D \cup \delta$|-g iff $J_g \cup \delta$|-g

The following is the algorithm used for backchaining from a proposition p. Initially, p would be the added goal, g.

1. Find all the backward clauses which have an atom which would match p at the head. For each such clause, e.g. $q \leftarrow q_1,q_2,..$ do the following:
2. Match p with the head of the clause.
3. Evaluate all the direct predicates in the body of the clause. All the solutions to the direct predicates are evaluated, and for each solution, a different copy of the instantiated clause is generated.
4. By this stage, all the variables in clause should be instantiated. For each instantiated copy of the clause, create proposition corresponding to the non-direct atoms in the body. Add the justifications corresponding to the instantiated copies of the clause to the agent's set of justifications. Backchain from all the non-abducible propositions just created.

Example: Let us consider a rule such as

```
p(X,Y)<-increment(X,Z),r(Z,Y).
```

where increment is a direct predicate. Now suppose we want to back-chain from the proposition p(5,a). What happens is this: the head of the rule (i.e. p(X,Y)) is matched with the proposition that is being backchained. This will half-instantiate the clause, in this case to p(5,a)<-increment(5,Z), r(Z,a). Then increment(5,Z) is evaluated. Let us assume that increment is defined by the single clause

```
increment(X,Y):- add([X,1],Y).
```

where add is a system predicate which computes the sum of a list of integers. Thus the effect of evaluating increment(5,Z) is that Z is bound to 6. Now the fully instatinated clause is

```
p(5,a)<-increment(5,6), r(6,a)
```

Now we create a proposition corresponding to r(6,a) and add the justification p(5,a)<- r(6,a) to the truth maintenance system. Then we backchain from r(6,a).

Note: by the time the evaluation of the direct predicates is finished, the clause should be fully instantiated. *It is the responsibility of the programmer to ensure this, the interpreter does not check it.* Serious errors will result if some unbound variables are left in the clause following the evaluation of direct predicates.

6.2 The Forward Chaining Algorithm

The forward chaining algorithm is invoked whenever a proposition is added to the agent's set of propositions. Let the current TMS-State of the agent be <R,J,A,G, T,Δ>, i.e. R is the current set of propositions and J the current set of justifications. Let F be the set of forward rules of the agent, and D its direct rules. What we require from the forward chaining algorithm is to maintain the following constraints:

- Every justification in J is an instantiation of a clause in F in the context of D
- for all subsets σ of R and any proposition q in R, $F \cup D \cup \sigma \vdash p$ iff $J \cup \sigma \vdash q$.

In general, when a new proposition is added to R, the second constraint will not be satisfied any more, which is why we invoke the forward chaining algorithm to maintain it. When the algorithm is finished, the constraint will be satisfied.

Let p be the proposition which is to be added to R. The following is the broad outline of the forward chaining algorithm.

1. Add p to R.
2. Find all the forward clauses in the agent program which have an atom in their body which matches p. For each such clause,
3. Match p with the atom in the clause with which it can match. This will half-instantiate the clause.
4. Evaluate all the other atoms in the body of the clause left to right: those with direct predicates are evaluated by running the corresponding direct rules, and the non-direct atoms are matched against the already existing propositions in R. All possible solutions to the direct predicates, and all possible matches with existing propositions are considered, thus this step in general leads to many different instantiated copies of the clause.
5. At the end of this process,for each copy of each clause, all the variables in the clause should be bound. Add the justification corresponding to the instantiated clause to J. If the instantiated head atom is not in R, add it to the set of propositions and forward chain from it.

Notice: unlike backward clauses, the non-direct atoms in the body of the clause are only matched against existing propositions. No new proposition is created to correspond to the atoms in the body of the clause. If more than one proposition mathces a given atom, then for each matching proposition, a new copy of the clause is considered.

Notice: At the end of the instantiation process, no variables should be left in the head of the clause. *It is the responsibility of the programmer to ensure this, the interpreter does not check it.* Serious errors will result if there is a variable left in the head of the clause at the end of the instantiation process.

Example: Let the current set of propositions be {p(a,5),p(b,6),p(c,6)} and the forward clause in question be s(X),increment(X,Y),p(Z,Y)->r(Z). Let's assume that the proposition being forward chained is s(5). Then the following is the instantiation process for this clause:

- match s(X) in the clause with s(5). This will half-instantiate the clause to s(5),increment(5,Y),p(Z,Y)->r(Z)
- evaluate increment(5,Y). The only solution found is increment(5,6). This instantiates Y to 6. Now the clause is s(5),increment(5,6),p(Z,6)->r(Z)
- match p(Z,6) with all the existing propositions with which it matches. This would create the two ground clauses s(5),increment(5,6),p(b,6)->r(b) and s(5),increment(5,6),p(c,6)->r(c)
- create the propostions r(b) and r(c), and add the justifications s(5),p(b,6)->r(b) and s(5),p(c,6)->r(c) to the set of justifications for this agent.

6.3 Evaluation of direct predicates

Direct predicates are evaluated in the context of instantiation of forward or backward predicates, as discussed above. For every direct predicate, there are a set of direct rules which define the predicate. A direct rule is a clause in pure Prolog (i.e. Horn clause with negation as failure). The sense in which direct rules define predicates is the same as that employed in logic programming.

The only point worth mentioning about the use of direct rules in Agent Alpha is that when a non-ground direct atom is evaluated with respect to its definition, an all-solution breadth firsts strategy is employed for evaluation, rather than the usual one-solution depth first strategy used in Prolog. So, for example, if we define append as

```
append([],X,X).

append([X|Y],Z,[X|U]):- append(Y,Z,U).
```

and then try to evaluate the direct atom append(U,V,[a,b,c]) with respect to this definition, we will get all of the following evaluations:

```
append([],[a,b,c],[a,b,c])

append([a],[b,c],[a,b,c])

append([a,b],[c],[a,b,c])

append([a,b,c],[],[a,b,c])
```

So when writing direct rules, care should be taken that unwanted solutions are not logical consequences of the definition, because they will always be generated, unlike Prolog where one may get away with having unwanted logical consequences of the rules which are never generated because of the one-solution, depth first strategy. (Of course, in Prolog, too, these unwanted solutions will be generated on backtracking, but if the programmer is certain that backtracking will not occur, he does not have to worry about this.)

7 Syntax of Agent Alpha

In Agent Alpha, Agent programs are collections of declarations and rules. The following is a small agent program:

```
agent(travel_agent).
abducible([go_by_rail,go_by_bus]).
recursive([]).
comply(T,Agent,organize_trip(Travellers,Destination))<-
                        organize(Travellers,Destination).

organize(Travellers,Destination)<-
                        go_by_rail(Travellers,Destination).

organize(Travellers,Destination)<-
                        go_by_bus(Travellers,Destination).

                        go_by_rail(Travellers,Destination)->
buy_rail_ticket(Travellers,Destination).

                        go_by_bus(Travellers,Destination)->
buy_bus_ticket(Travellers,Destination).

                        buy_rail_ticket(Travellers,Destination)->
request(station_master, get_ticket(Travellers,Destination)).

                        buy_bus_ticket(Travellers,Destination)->
request(bus_company,get_ticket(Travellers,Destination)).

                        assimilate(T, station_master,ticket_reserved(Travellers,Destination)),
                        member(Agent,Travellers) ->
inform(Agent,ticket_reserved(Destination)

member(X,[X|Y]).
member(X,[Y|Z]):- member(X,Z).
```

7.1 Declarations

The first line in the agent program is the declaration of the agent name.

agent_name(*name).*

where *name* is the name of the agent.

The second line in the agent program is the list of abducible predicates.

abducible([*predicate1, predicate2,...*]).

declares *predicate1, predicate2,...* to be abducible.

The third line in the agent program is the list of recursive predicates.

recursive([*predicate1, predicate2,...*]).

declares *predicate1, predicate2,...* to be recursive.

7.2 Rules

The syntax of agent rules is very close to the syntax of Prolog. We use *constant*, *variable* and *term* in the same sense as that employed in Prolog. We use *atom* to refer to a predicate applied to a number of arguments.

In Agent Alpha, there are three kinds of rules: backward rules, forward rules, and direct rules.

7.2.1 Backward Rules

A backward rule is a rule of the type

head<-body1, body2,..bodyn.

where *head, body1, body2,..bodyn* are atoms. There are syntactic restrictions on variables in backward rules, these restrictions are explained in the section on the instantiation of backward rules.

Example:

```
comply(T,Agent,organize_trip(Destination))<- organize(Agent,Destination).
```

is a backward rule.

Recursive rules are a special kind of backward rule. A predicate which is declared to be recursive should only occur at the head of a pair of backward rules following the pattern below:

```
recursive_p([],arg1,arg2,...).
recursive_p([X|Y],arg1,arg2,...)<-
        p(X,arg1,arg2,...),
        recursive_p(Y,arg1,arg2,...).
```

where recursive_p is the predicate defined to be recursive, and *arg1,arg2,...* are terms. *X* and *Y* should be variables. p can be an arbitrary predicate.

7.2.2 Forward Rules

A forward rule is a rule of the type

body1, body2,..bodyn->head.

where *head, body1, body2,..bodyn* are atoms. There are syntactic restrictions on variables occuring in forward rules, these are explained in the section on instantiation of forward rules.

Example:

```
assimilate(T, station_master,ticket_reserved(Travellers,Destination)),
```

```
                    member(Agent,Travellers) ->
inform(Agent,ticket_reserved(Destination)
```

is a forward rule.

When the head of the forward rule is false, then that rule is an integrity constraint. For example,

```
happy(x),sad(x)->false.
```

means that nobody can be happy and sad at the same time.

7.2.3 Direct Rules

A direct rule is a rule of the type

head:-body1, body2,..bodyn.

where *head, body1, body2,..bodyn* are atoms, or an assertion of the type

head.

A predicate which appears at the head of a direct rule is implicitly declared to be a direct predicate.

Example: the following two rules are direct rules

```
member(X,[X|Y]).
member(X,[Y|Z]):- member(X,Z).
```

8 Terms in Agent Alpha

Agent Alpha terms are a subset of those in Prolog, with the exception of strings and tuples. Below we give a brief description of the primitive and compound terms in Agent Alpha.

8.1 Primitive terms

8.1.1 Constants

A constant is an alphanumeric identifier starting with a lower case letter. tom is a constant.

8.1.2 Variables

A variable is an alphanumeric identifier starting with an upper case letter. Tom is a variable.

8.1.3 Integers

An integer is an unsigned sequence of digits. 723 is an integer.

8.1.4 Strings

A string is a sequence of characters starting with " and ending with ". "hello world" is a string.

8.2 Compound Terms

Functions and lists have exactly the same syntax as in Prolog. The only extra term type in Agent Alpha is the tuple. A tuple is a term $<t_1,t_2...t_k>$ where $t_1,t_2...t_k$ are terms.

9 System Predicates

System predicates are direct predicates which are predefined by the system, i.e. their definition does not need to be included in the agent program. In general, system predicates have the following format: p(Input,Output) where input is the input to the predicate, and the result is returned in Output. Thus the first argument of system predicates should be bound when the system predicate is called. The system call will succeed if Output can be bound to the result of the computation. If the second argument is not a free variable at call time, then the call will succeed if the result can be unified with the second argument, and will fail otherwise.

The following are the system predicates currently supported by Agent Alpha.

9.1 term_to_string(Term,String)

Will create a string from a term.

Example: term_to_string(f(a,b,c),X) will bind X to "f(a,b,c)"

9.2 concatenate([String1,String2,...],String)

Will concatenate String1,String2,... and return the result in String.

Example: concatenate(["hello ","world"],X) will bind X to "hello world"

9.3 current_time(now,X)

Will return the current time (in Unix Standard Time) in X

9.4 add([N1,N2,...],X)

Binds X to N1+N2+..

Example: add([5,6,7],X) will bind X to 18

9.5 times([N1,N2,...],X)

Bind X to N1× N2×,..

Example: times([2,3,4],X) will bind X to 24

9.6 subtract([N1,N2,N3...],X)

Binds X to N - N2 - N3...

Example: subtract([10,5,3],X) binds X to 2

9.7 divide([N1,N2,N3...],X)

Binds X to N/(N2×N3..) using integer division.

Example: divide([10,5,2],X) binds X to 1

9.8 less([N1,N2],X)

Binds X to yes if N1<N2, to no otherwise.

10 Action Propositions

Action propositions are those which, when they become true in the agent't current state, will lead to some action. The main action propositions are inform and request. Normally, these would occur at the head of forward clauses, and become true as a consequence of the antecedents of the rule becoming true. Below is the list of action predicates currently supported in Agent Alpha.

10.1 inform(*agent, inform_message*)

When this proposition becomes true, *agent* is informed of *inform_message.*

10.2 request(*agent,request_message*)

When this proposition becomes true, *agent* is requested to do *request_message*

10.3 x_window(inform,*display_message,reference*)

display_message must be instantiated to a text string, and *reference* is an instantiated term which is used for identifying the response to this message.When this proposition becomes true, a window will pop up containing the message *display_message.* When the user presses the OK button in this window, an inform(*agent,reference*) message will be sent to *agent*, where *agent* is the name of the agent which initiated the display action.

10.4 x_window(query,*query_message,reference*)

When this proposition becomes true, a window will pop up containing the message *query_message,* and two buttons labelled yes and no. When the user presses one of these buttons in this window, an inform(*agent,<reference,reply>*) message will be sent to the agent which initiated the display action. *reply* will be either yes or no, depending on which button the user pressed.

10.5 x_window(dialog, *dialog_message,reference*)

When this proposition becomes true, a window will pop up containing the message *dialog_message,* and a text entry field, which can be used for typing the reply to the message. When the OK button in this window is pressed, the message inform(*agent,<reference,reply_message>*) is sent to the agent which initiated this action, where *reply_message* is the text the user entered in the text entry field of the window.

11 Loading Agents: the file .AOPagents

At start up time, the interpreter reads a file named .AOPagents to find out which agent files it should load up. Each agent should be in a separate file, and the names of these files should be in .AOPagents one file per line. The first three lines of this file should be

x_manager.ag

agent_init.ag

tms_spy.ag

These are agents which should always be loaded, as they are used by the system itself. The names of other agent files should follow these lines.

12 Resolving Agent Names: the file .AOPalias

When a message is to be sent to a local agent, i.e. one which is loaded by the same interpreter, then it is sufficient to simply put the agent's name as the first argument of the inform or request message. For example, inform(tom,hello) would send the message hello to an agent named tom loaded in the same interpreter. When messages are to be sent to agents residing on other machines, then the agent name should be in the following format: agent@alias, where alias is the name of the interpreter on which the agent resides. Each interpreter is associated with a specific user name on a machine (thus there should not be more than one interpreter per user name on any machine). The file .AOPalias contains the information about which alias is associated with a given user name on a machine. For example, let the .AOPalias file contain the following lines:

alias1 kave hplav.hpl.hp.com

alias2 kave hplke.hpl.hp.com

alias3 steven hplsr.hpl.hp.com

Then sending the message inform(tom@alias1,hello) will send the message hello to the agent named tom running on the interpreter belonging to the user kave on the machine hplav.hpl.hp.com

When messages are received from agents on other machines, the field corresponding to the name of the sending agent will be agent@alias, where alias is the name of the interpreter on which the sending agent resides. (This assumes that both interpreters use the same .AOPalias file)

In order for the name resolution mechanism to work smoothly, it is very useful (though not imperative) for all the interptreters trying to communicate with each other to have the same .AOPalias file.

Example:

There is an interpreter running under the user name steven on the machine hplsr.hpl.hp.com. There is another interpreter running under the user name kave on the machine hplav.hpl.hp.com. Both of these interpreters use the above alias file. Thus the alias of the first interpreter is alias3, and the alias of the second interpreter is alias1.

Now suppose there is an agent named mary on the first interpreter (alias3), and an agent named tom on the second interpreter (alias1). If mary sends the message inform(tom@alias1,hello) to tom, tom will receive the message

assimilate(78181881,mary@alias3,hello)

where 78181881(say) is the time the message was received.

13 Sending a Message to an Agent: the .AOPcommand file

When the interpreter is started up, it brings up a window which can be used for sending messages to the agents directly. To do this, the user types the message in the text field provided, and presses return. To avoid typing these message repeatedly in debugging situations, the often used messages can be put in the .AOPcommand file. This file will be read by the interpreter when it is started up. Then the text field for sending messages can pre-filled by clicking on any of the messages in the appropriate window.