1 Introduction

This is a general overview of the Agent Alpha programming language, which is a member of the family of Agent Oriented Programming languages. The document 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 $p \leftarrow a \wedge b \wedge c$..This set includes *nogoods*, which are clauses of the form $false \leftarrow a \wedge b \wedge c$.., and *facts*, which are clauses of the form $p \leftarrow$ (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 $<R_1,J_1,A,G_1, T_1,\Delta_1>$ be the state of an agent at time t_1, and $<R_2,J_2,A,G_2, T_2,\Delta_2>$ 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 $<R,J,A,G,T,\Delta>$ 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, $P\cup\sigma|\text{-}\ q$ iff $J\cup\sigma|\text{-}\ q$
- A is the intersection of R and atoms(Ab),
- For any subset δ of atoms(Ab) and any goal g in G, $P\cup\delta|\text{-}g$ iff $J\cup\delta|\text{-}g$

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 $<R,J,A,G,T,\Delta>$ is a sufficient instantiation of <P,Ab,G> and $<R,J,A,G,T,\Delta>$ 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.

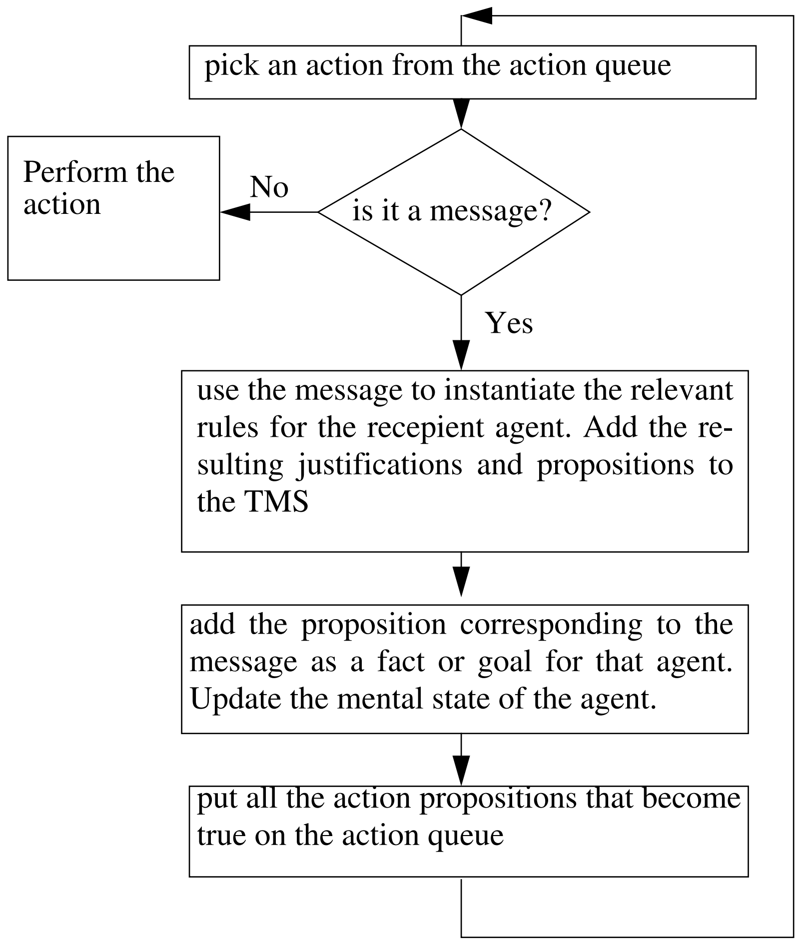
In general, as a result of adding a comply atom to the agent's goals, or adding an assimilate atom to the agent's facts, the agent's mental state will change. This happens in two stages:

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 $p \leftarrow q_1, q_2, \ldots q_k$ is an instantiation c in the context of D iff for some $d_1, d_2, .. d_n$, $D \mid\!- d_1, d_2, .. d_n$ and $p \leftarrow q_1, q_2, \ldots q_k, d_1, d_2, .. d_n$ 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), $B \cup D \cup \delta \mid\!- g$ iff $J_g \cup \delta \mid\!- g$

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 $\langle R,J,A,G,T,\Delta \rangle$, 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 occurring 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(display,*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.