Imperial College of Science, Technology and Medicine

University of London

Department of Computing

KNOWLEDGE ASSIMILATION IN MULTI-AGENT SYSTEMS.

by

Jacinto Alfonso Dávila Quintero

Submitted in partial fulfilment of the requirements for the MSc degree in Engineering of the University of London and for the Diploma of Imperial College of Science, Technology and Medicine.

September 1994

Gracias Señor por la oportunidad..

ACKNOWLEDGEMENTS

I would like to thank my supervisor Prof. Robert A. Kowalski for the valuable support and encouragement he has given me throughout this project.

I would like also to thank Prof. Frank McCabe for his support and attention and for providing the basic software for building this project; Yuji Takada and Francesca Toni for their friendly attention and professional support and Jans Wielemaker who gently accepted my queries and helped me to debug the intricate code of SWI-PROLOG.

Special thanks to Sylvana Zappacosta who always provided technical and academic support and extra-academic advice during this year at Imperial College.

Finally but mainly, I would like to thank to my wife, Liliana, my fellow girl in this venture.

CONTENTS

Index

Table of figures

ABSTRACT

This work is concerned with logic-based Multi-Agents Systems. We have investigated multi-agent systems as *open* knowledge bases, with knowledge and beliefs represented in a logical formalism. We show how to model agent brains as logic programs using *forward-backward* representations and *search by layers*. Our agents perform *knowledge assimilation,* observing inputs and adapting themselves to a changing environment. They also can use *abduction* to reason about actions and perform outputs to the environment. Our agents keep an explicit representation of goals which is used in conjunction with observations to guide the behaviour of those agents in the world. All this is accomplished within a *logic programming framework*.

We have modelled agents using PROLOG and APRIL as the building tools and simulate a world populated by this sort of agents. A set of experiments has been carried out with the PROLOG-APRIL test-bed and the obtained data has been used to analyse the effect of *bounded reasoning* in the performance of the agents. We suggest that other logic-based tools as the *Event Calculus* and *Metalogic* can be use to tackle the problems of planning and cooperation in Multi-Agent Systems.

INTRODUCTION

I.1.Logic-based Multi-Agents Systems.

This work is concerned with logic-based Multi-Agents Systems. Multi-Agents is an important category of Distributed Artificial Intelligence Systems (DAI), whose main feature is that the distributed entities are autonomous (self-controlled). An agent is an aggregate of knowledge, practical skills, and its own single locus of control and intentions [Bond, Gasser; 1988]. With independent entities interacting in the same system, a clear need exists for coordinating their activities in pursue of goals seen as individuals interacting goals or as global goals. The other universally accepted category in DAI, Distributed Problem Solving (DPS), considers how to split a problem into a collection of "nodes" [Bond, Gasser; 1988] that can work separately with less complex sub-problems. DPS systems use the pre-obtained "problem architecture" to coordinate the solving process and integrate the solutions. However, unlike DPS, in Multi-Agents Systems the task of coordination can be quite difficult. In more interesting systems (realistic *open systems* [Hewitt; 1985]), it may be the case that there is not possibility for global control, globally consistent knowledge, global shared goals or universal success criteria and that the system representation is incomplete. Under some or all of those conditions, the agents *need to reason about the process of coordination among themselves* [Bond, Gasser; 1988].

According to the previous picture an agent could be a rather complex entity. We are talking about systems that have explicit representations of goals and an internal representation of the world that probably includes models of the other agents. The agents use their knowledge bases for reasoning and planning their next actions. The agents are, therefore, strictly rational agents with a well-defined logical inference machine.

Some researchers have argued that this approach for system construction, using logic-based agents, is "entirely unrealistic" [Steels; 1990] because: 1) the technological complexity required is too high, 2) there is no way of extracting logical description of the world from the current sensing devices and 3) Logic-based AI faces several theoretical problems (frame problem, reasoning about time and space, and modelling the non-monotonic dynamic of the world). Steel [Steels; 1990] proposed an apparent alternative to the logic-based approach: the behaviour based approach in which, he uses the subsumption architecture of Brooks [Brooks; 1986] to get emergent functionality from very simple *reactive* pattern.

The aim of the current work is to show that reactive multi-agent systems can be modelled as logic-based agents with bounded resources. We employ a Logic Programming framework to implement agents with limited reasoning capability, which perform efficiently but still use a sound inference mechanism for reasoning about themselves and the external world and planning their actions. As part of a Logic Programming framework, ours system can employ representations that overcome the problems of reasoning about time and the frame problem. Our proposal addresses the key issue of adaptation whereby the agents can cope with the dynamic of the world. We implement for each agent an adaptable locus of control algorithm, recently presented [Kowalski; 1994b] which interleaves planning and execution.

Possibly the more important by-product of this work is that, by implementing efficient agent in a logic programming framework, we let open the possibility for developing multi-purpose Multi-Agents Systems. Systems in which the agents may reason about cooperating strategies, may efficiently use communication and shared resources and may accomplish realistic autonomy. All this without having to lose performance or other practical capabilities.

This document is organised as follow. The last section of the Introduction presents the system used as the test-bed: APRIL and the APRIL-PROLOG interface. Chapter 1 presents the logic-programming implementation of the "brain" of our agents. We start by giving account of previous research in logic-base Multi-Agents Systems and located our work in this stream. In Chapter 2, we present our implementation and compare the Steel's [Steels; 1990] proposal to our, showing how the subsumption architecture can be seem as a resolution search strategy and how to model *forward-backward* representations. Chapter 3 describes our experiments and presents the results. We conclude in the next part and show possible extensions to this work and guidelines for the rest of this research project. The appendices contain the source codes of our implementation and present the details of the APRIL-PROLOG interface developed as part of this work.

I.2 April: A language and a Test-bed for making experiments with Multi-Agents Systems.

APRIL, *Agent PRocess Interaction Language*, is a programming language and a test-bed for building distributed artificial intelligence applications [McCabe; 1993]. The language is process-oriented. Programmers can write multi-process applications in a high level approach that combine the thread-based program structure, with a C-UNIX like syntax. A typical APRIL file contains several procedures including the "main" procedure, in complete analogy to c files (the procedures are similar to c functions). Some of these procedures can be, depending on the programmer's design, invoked as separated process by "fork" calls (the threads of some multiprogramming environments). However, the implementation details of the communications channels among process are completely transparent for the programmers.

Apart from those features that can make the language attractive for writing distributed applications, APRIL is also a symbolic language. It has a powerful data structuring and expression handling features and pattern-matching. It is a strongly typed language, with facilities for user-defined types but only one built-in complex structure (the tuple). The language can be its own meta-language and even program codes can be interchange among processes (as messages) and incorporated to their code at run time. The environment includes a macro-processing sub-language whereby the original language can be transformed and extended to fit user necessities or wishes. APRIL is an object-oriented language thank to its macro-facilities [McCabe; 1994].

In spite of all said, APRIL is not "*a multi-agent applications language*" [McCabe; 1993]. it does not oblige any particular internal agent's architecture (knowledge and/or belief representation and control mechanisms, as in Shoham's AGENT0 [Shoham; 1991]). We believe what makes APRIL so handsome for multi-agent's implementations are its facilities for modelling *interactions* among process and, consequently, among agents. For the sake of "*briefness*" let us present an example of an APRIL program that models a real human conversation. The example is not particularly enlightening, but it is a real human talk (it is a common joke-game among grandparents and their grandchildren in Venezuela) and has the advantage that is supposed to last forever. One of the *agents* (the grandfather) starts by saying: "*Do you want me to tell you the story of the bald cock?*" . The other agent (the grandson) answers (specially if it's the first time he plays) "*Yes!*". Then the first agent uses this answers to reply: "*It is not that **'Yes!'** but if you want me to tell you the story of the bald cock?*". From there on, the first agent keeps using the second's responses to reply and if the second agent is patient enough (the grandparents always are) that chat becomes a never-ending sequence with the following dialogue structure:

Agent1: Do you want me to tell you the story of the bald cock?
Agent2: <*any answer*>
Agent1: It isn't that '<*any answer*>' but if you want me to tell you the story of the

bald cock.

Our two agents can be simulated by the following APRIL program. This program has two procedures (representing one agent each) and the main procedure for setting up the chat. All of them can be store in the same text file (but the file structure of APRIL is currently changing to a modular structure, to cope with more comprehensive developments).

```
/* ___________________________________________ agent2 */
agent2(){
 while true do {
   DoYouWantMeToTellYouTheBaldCockStory => {
     symbol?answer := genAnswer(yes_No);
     answer >> replyto
   }
 | quit => break
 }
};

/* ___________________________________________ agent1 */
agent1(handle?agent2){
 DoYouWantMeToTellYouTheBaldCockStory >> agent2;
 while true do {
   quit => break
 | any?answer => {
     DoYouWantMeToTellYouTheBaldCockStory >> replyto
   }
 }
};

genAnswer(symbol?basicAnswer) -> symbol?valof{
 symbol?basename := gensym();
                  /* generate a symbol like Noname01 */
                  /* split it into "Noname" and "01" */
 [symbol,symbol]?splitname := split(basename, 7);
                  /* take the "01" part into counter */
 [symbol,symbol?counter] = splitname;
                /* concatenate argument with counter */
 symbol?answer := catenate( basicAnswer, counter );
 valis answer        /* Return the just built answer */
} ;
```

```
/* ____________________________________ main process  */
main(any[]?ar){
 if |ar| < 2 then {
   writef(stdout,"usage: april exam1 <agent2name>
<agent1name>\n",[]);
   exit(1);
 };

 handle?grandson := handle?ar[1];
 handle?grandfather := handle?ar[2];

 /* the processes representing the agents are created */
 grandson names agent2() ;              /* fork call */
 grandfather names agent1(grandson) ;    /* fork call */

 { any?Msg => relax
 | timeout 2 secs => relax }; /*simulat. last 2
seconds*/

 /*The main process inform the agents the game is over
*/
 quit >> grandfather;
 quit >> grandson;

};
```

The first part of the file contains the procedure that models `agent2`. Its structure is very similar to a C-function. There is a "while" control structure, which as usual, is a loop in the execution flow. However, inside the loop there is an unusual structure: several statements separated by "|" (*{ st1* | *st2* | .. | *stn }*). This is the choice operator and the sentences it separates are *guarded statements (G => S)*: statements preceded by tests that allows or disallows their execution. There are two types of guarded statement in APRIL: semantics guards and message guards. In this case, those are message guards. Whenever a message is received by agent1, this message is compared with the set of guards inside the while. If the message *matches* one of the guards, this one succeeds and its associated sentences are executed. If no message matches any guard (or if there is no message at all) the process stays suspended.

Only two types of message can be "accepted" by `agent2` (for simplicity instead of the answer "*it isn't that*" we keep using the first question). `DoYouWantMeToTellYouTheBaldCockStory` is one (rather long) pattern, and `quit` is the other. Whenever a message containing `quit` is received, the sentence `break` is executed and the process gives up the while loop, ending the execution afterwards. Observe that the `agent2` process does not care who (which process) sent a message. However, there are ways of checking the message's source using the guard test. The other important things to note in the code are the declaring statements with the form *type?var* (which in this code always appears combined with the assignment, but this is not compulsory). A variable can be declared anywhere in the code (as in C) and its scope extends until the next ending curly brace "}". There are 7 basic types in APRIL: `symbol, handle, number, integer, real, logical` and `any`. Our example only employs the first two. The last one (`any`) is used when no type can be anticipated. As we said before, only one built-in, compound type exists in APRIL: the tuple. Nevertheless, users can define its own types, even using tuples of tuples nested to an arbitrary depth.

Procedure `agent1` should be understandable at this stage. The structure `genAnswer` is a function, the only kind of APRIL subroutine that return values. Hence, unlike procedures, functions have types (`*functioname*(*argdecl*) -> *type*?valof{ .. valis *value* }`). The `main` procedure is explained by the comments. The function `names` create a new agent with the name given as first parameter. We obtain those names from the arguments given when the APRIL test-bed is called to run this program. It worth noting that an APRIL process (in this case, there is one process in each agent) is not a machine (operating system) process. The APRIL *executive* implements its own non-preemptive process manager that executes the code pre-compiled (by the APRIL compiler).

By adding some write instructions (see appendix A) we can get the following *trace* of our simple multi-agents' application:

```
MAIN(april#19555):Starting the simulation

AGENT1(Grandpa):Do you want me to tell you the story of
the bald cock?..

AGENT2(Grandson):yes_No0

AGENT1(Grandpa):It isn`t that yes_No0, but if you want
me to tell you the story of the bald cock..

AGENT2(Grandson):yes_No1
```

Logic Programming Agents

```
AGENT1(Grandpa):It isn`t that yes_No1, but if you want
me to tell you the story of the bald cock..

AGENT2(Grandson):yes_No2

AGENT1(Grandpa):It isn`t that yes_No2, but if you want
me to tell you the story of the bald cock..
.
.
.
AGENT1(Grandpa):It isn`t that yes_No423, but if you want
me to tell you the story of the bald cock..

AGENT2(Grandson):yes_No424

MAIN(april#19555):Ending the simulation
```

APRIL is not only a language. It is a distributed platform that includes a compiler, a program administrative (an interpreter of pre-compiled APRIL programs), a nameserver (for managing the addresses of agents located anywhere in the network) and also that provides facilities for visualisation through a special agent called DIALOX. DIALOX is seen, from the point of view of other APRIL agents, as another agent that can receive messages related to visualisation tasks.

In this project we use APRIL for building the agents of the application and other especial agents representing the *worlds* where the former *live*. We use DIALOX for displaying the activities of those agents in those worlds (as elemental computing animations). Finally, we use PROLOG and all its inference machinery for implementing the *brains* of the intelligent agents. For the point of view of APRIL agents, PROLOG is, as DIALOX, one more agent with which they can interact.

APRIL is a system under development. The compiler is been migrated from PROLOG to C, seeking efficiency for bigger applications. DIALOX is in its first versions. As part of this MSc project, we develop an APRIL-PROLOG interface that allows PROLOG programs to perform as APRIL agents. The interface has been developed on top of a TCP/IP (UDP) communications platform[1] that has been used for DIALOX - APRIL communications. The details of our application are presented later on, but the general architecture is summarised in figure I.1. Every rectangle is an operating-system process acting as an APRIL agent. The arrows represent

1This interface was provided by Prof. F. McCabe.

interchange of APRIL messages. The titles in bold letters show the language used to write each program. DIALOX and apnameserver are independent agents. The nameserver (apnameserver) is consulted by the other agents when they need to find out the real location (addresses) of other agents. There is no logical bound for the value K (the number of simulated agent).

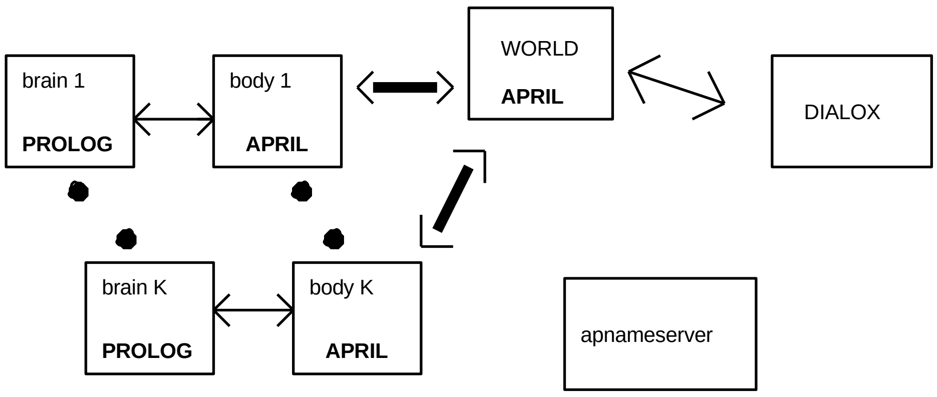


Figure I.1 APRIL-PROLOG test-bed.

This project has been developed in a UNIX platform. The APRIL system runs in SunOS machines with TCP/IP as the networking environment. DIALOX employs X11/R6 servers and compatible versions. The PROLOG system employed is SWI-PROLOG [Wielemaker; 1989]. The PROLOG-APRIL interface is a set of C-programs incorporated to the UNIX (SunOS) SWI-PROLOG. The interface employs the signalling facilities of UNIX and therefore, it is not completely portable to other operating systems, unless these provided a signal management similar to UNIX's. The PROLOG programs try to preserve Edinburgh compatibility (however, we had to use the particular I/O tools of SWI-PROLOG). Some of them were developed and tested in the DOS-WINDOWS SWI-PROLOG.

CHAPTER 1

LOGIC PROGRAMMING AGENTS

1.1. From reactive agent to deliberate agents.

There have been numerous attempts to model agents either as reactive or as deliberate entities. In this section we summarise one attempt that tries to cover the gap between them. In [Genesereth, Nilsson; 1988] Genesereth and Nilsson have proposed 5 types of agent architecture that could be seen as corresponding to ordered steps in a progression between reactive and cognitive agents. Starting with The Tropistic Agent, "*whose activity at any moment is determined entirely by its environment at that moment*" [Genesereth, Nilsson; 1988] they define several basic concepts: partition, sensory function, effectory function and action function. All those ideas, together with the traditional notions of *state* and *action* allow this authors to characterise the Tropistic Agents as a 6-tuple:

<States, Partial descrip, Actions, sensory function, effectory function, action function>.

Observe that this conceptualisation does not take into account the idea of an internal state in the agent (memory). The action function simply relates actions with situations in which the agent might be. These relations are probably *hard-wired* in the agents' structure. The agent does not need to record any global or permanent information. Indeed, the agent does not distinguish perfectly all the states of the world. It only recognises what they call partitions: set of states in which certain feature or property holds. We prefer name those sets *partial descriptions* as they are incomplete descriptions of states in the world stored inside the agent. This model fits perfectly with the generally accepted conception of what it is a reactive agent. Moreover, the term *Tropism* has been used in other works on Reactive Agents (See for instance [Feber, Drogoul; 1991]).

The notion of internal state appears in the following type or agent: *The Hysteretic Agent*. The internal state is used as an additional parameter for defining the next action to be performed by the agent.

However, the next architecture presented by [Genesereth, Nilsson; 1988] substituted the original proposal for an internal state (which is too much detailed according to them), with the notion of database. This could be seen as rising the abstraction level of the model. From this point on, the internal state of the agent contains a set of sentences in predicate calculus conforming a database. The content of this database (and the non-predefined method of consulting and searching) determines the next action to be performed. This is a *Knowledge-Level Agent* Architecture.

The Stepped Knowledge-Level Agent is an monotonic agent in the sense that new sentences can be added to the database (programme), but no sentences are ever removed. To accomplish this, the agents need and additional structure as part of their internal state: a counter of the cycles of the agent's operation.

Finally, *The Deliberate Agent* "*prescribes the use of some automated inference method in deriving the sentences that indicate the required action on each cycle*" [Genesereth, Nilsson; 1988]. Of course, the inferences are made upon the sentences in the agent's database which, in turn, is updated to include observations and past actions' records. These authors also present an *alternative characterisation* of a deliberate agent as a programme in imperative style ([Genesereth, Nilsson; 1988], fig. 13.9):

```
Procedure  CD(DB)
Begin      CYCLE <- 1,
Tag        OBS <- OBSERVE(CYCLE),
           DB <- APPEND([ T(OBS, Ext(CYCLE) ]),DB),
           ACT <- FIND( k, Must(CYCLE)=k, DB),
           EXECUTE(ACT),
           DB <- APPEND([ Act(CYCLE)=ACT], DB),
           CYCLE <- CYCLE+1,
           GOTO Tag
End
```

1.2. From deliberate agents to reactive agents.

The work just discussed covers the gap between reactive and deliberate agents but does not consider the opposite direction: how can a deliberate agent be transform to a reactive agent. This has to be done in order to assert that both agents general architectures are transmutable into each other. Previous works have avoid the idea and, instead, have tried to combine both architectures. The InteRRaP Project [Muller, Pischel; 1993]) have considered the use of *pattern of behaviour*, a special version of production rules in which the action component is a set of pre-compiled abstract low level actions (a plan) for executing *routine* behaviours. *This routine behaviour does not require deep* *reflection or planning* [Muller, Pischel; 1993]. In the InteRRaP project the agent structure combines behaviour-base components with plan-based components, acknowledging the fact that efficient agents still require a deliberate component. They insist in seeing its reactive components (pattern of behaviour) as procedural knowledge, in contrast to the knowledge that can be *represented in a declarative manner*.

Our proposal shares with the InteRRaP project the belief that a complete autonomous agent should somehow combine reactive behaviour with *deep* well-founded reasoning. However, we believe that all this can be provided inside the framework of a logical representation. We believe that the reactive behaviour *is as logical as possible*. An agent needs to decide the next action it will execute. However, probably due to some environmental constraint, it does not have enough resource to perform a *deep thinking* for getting a well-founded solution. Therefore, it has to consider only those alternatives that it can superficially explore in the little time or with the small space of memory it has. Of course, if the situation prescribes a *routine behaviour*, the agent could capitalise over previous reasoning achievements that, after been accomplished several times, were stored as pre-compiled (partial evaluated) solutions. We believe that partial evaluation is the key for integrating learning and the development of skills into our agents' setting, but we do not take this issue any further in the present work. Moreover, there are other alternatives for *helping* the agent to make a sensible decision as we will show later on.

The key strategy we use to obtain reactive performance is by limiting the resources the agent has to perform its reasoning activities (time, space of memory or speed of processing). We propose to extend the Genesereth&Nilsson's conceptualisation of *Deliberate Agents* to incorporate the notion of ***resource allocation function***. Our intention is to prove that a Cognitive Agent (Genesereth&Nilsson's Deliberate Agent) can be regarded as a Reactive Agent at certain moments of its *life*, when it has no enough resources to make deductions with a database. The basic limiting resource is *time*. Therefore we devise a function that, at any cycle in the agent's life, assigns limits of time to every internal activity (observing, updating database, reasoning and acting). The time allocation function maps the database (D), the agent's cycle (C), and the agent's internal condition (IC) to a time amount (N):

$$\text{resource\_alloc: } D \times C \times IC \text{ ---> } N$$

Consequently, the new Deliberate Agent with bounded time can be define as a 9-tuple as follows: D is an arbitrary set of predicate calculus databases (one for each cycle), S is a set of external states, T is a set of partial descriptions of states in S, A is a set of actions and N is a natural number indicating amounts of time to be employed in an internal action. In addition, *see* in the sensory function from S x Ns into T, *do* is a effectory function from A x S x Ne into S, *database* is an update function from D x C x T x Nt into the new D, *action* is a function from **D** x C x T x Nt into A and *resource_alloc* is the just defined resource allocation function that produces Ns, Ne and Nt, different time limits for each internal action or living phase (per cycle).

< D, S, T, A, *see, do, database, action, resource_alloc* >

Is worth noting that the parameter N (time resource) can have different interpretations. Considered in absolute terms, N limits the amount of time dedicated to each *phase* of the agent's life (Nt: time for thinking, Ns: time for observing and so on). However, if the cycle is considered as the real time reference (a cycle is the unit of time), N could be regarded as *speed*. Hence, an agent with a bigger Nt thinks *faster* than other with a lower Nt. In general when Ns is big enough, it can *plan ahead* as a *full cognitive agent*. On the other hand, if Nt is small, the agent can perform only a limited amount of reasoning. It might decide its next action but this is not sure. In the case of small Nt, the agent would be doing its action and its consequence observation *sooner* and therefore, its interaction with its environment is more similar to that of the *reactive agents*.

For the sake of simplicity, in the rest of the world we will consider Ns and Ne as being equal to 1. So, in a typical cycle of life, an agent performs an observation, reasons as long as its Nt allows, and then executes an action, if it could choose one. The action can fail or succeed and the cycle start again with other observation. This is the basic adaptable locus of control algorithm, which has been presented in a more general framework elsewhere [Kowalski; 1994b] and will be examined in the following section.

For completing our *discussion* about cognitive and reactive agent, there is something else to be said. In the Genesereth&Nilsson's proposal, the Tropistic agent does not have an internal representation of its state. Its *action function* is a map between partitions (partial descriptions of states obtained by specific sensors) and actions. However, one could see this *map* as part of an abstract database which, among other more complex knowledge structures, comprises *perception-action* rules implemented in hardware. The rest of the database may be implemented by other means (firmware, different software). In any case, the hardware portion of the database is by nature more efficient, and thus, even with a low N (few resources), is likely to be consulted during the process of deciding the next action. This imposes a *layered structure* that priorized the access to the hardware low-level layer, against the access to the rest of the database. In the *Tropistic Agent* the rest of the database does not exist. The perception-action rules are hard-wired in. Nonetheless, the knowledge relating perception with actions is still there. We are not getting rid of the representation. We are once again in the eternal dilemma of the programmer: to write programs in *assembler*, (or by wiring transistors) dealing with an awful amount of details but with full control and efficiency, or write programs in some high-level programming language, without so many details, but with the nuisance of sometimes inappropriate abstractions. In either case the programme will be there in one and/or other representation.

1.3. Knowledge Assimilation and the adaptable locus of control.

In [Kowalski; 1979], Kowalski proposed the use of assimilation for dealing with changes and updates in information systems and databases. In that context an information system was a *collection of assumptions expressed in logic* (the database), *together with a proof procedure and maintenance procedures*. The set of procedures deals with cases when 1) the new information is already implied by the database, 2) the new data implies existing data, 2) the new information is independent from the current database or 4) is inconsistent with it [Kowalski; 1979]. The proof procedure is used, not only to answer queries posted to the database, but also to perform the assimilation of inputs. Hence, its outcomes are an answer to the query and a new database that has assimilated the changes.

Recently, this very same idea has been reintroduced as part of a computational approach to logic called 'CL' (computational logic [Kowalski; 1994a]) which is aimed to understand autonomous intelligent agents and human reasoning. In this context, "***knowledge assimilation** provides a syntactic and pragmatic alternative to* **model theory** *as an account of the relationship between language an experience.[..] the experience takes the form of an inescapable stream of input sentences, which needs to be assimilated into a constantly changing knowledge base*" ([Kowalski; 1994a], pg. 46). Computational logic has very attractive features for supporting the work in multi-agent systems. Firstly, it is based in the clausal and logic programming forms of logic, which have proved useful for building a wide range of computing applications. In the second place, it provides an alternative that gets over the inconveniences of model theory, closed world assumptions and possible worlds' semantics of logic, seen by Hewitt [Hewitt; 1985] as inadequate to model open system's dynamics. Indeed, knowledge assimilation, by tackling the need of processing an almost continuous stream of input sentences, is fulfilling the requirement *to allow for the open-ended incremental introduction of new beliefs and* *objects*, also pointed out by Hewitt [Hewitt; 1985] as an essential need in open systems.

Also, the knowledge assimilation idea in Computational Logic has already been put to work. In [Kowalski; 1994b], a proposal that preceded and inspired our project, Kowalski presents an initial attempt to specify the *main cycle of a rational, active agent*. We called it ***the adaptable locus of control algorithm*** and may be sketched as a logic program as follow:

```
cycle(KB, Goals, T ) <-
      observe( Input, T ),
      assimilate( Input, KB, NewKB, Goals, NextGoals,
                  [T+1, T+N]),
      execute( NewKB, Goals, NextGoals, NewGoals,
               T + N + 1 ),
      cycle( NewKB, NewGoals, T + N + 2 ).
```

It worth noting that this logic program, in its procedural interpretation, is equivalent to the Genesereth&Nilsson's imperative program for modelling deliberate agents. The main difference is that the logical semantics of the former is provided by the CL framework. The locus of control could be explained as follow:

The `observe` condition obtains the input at time T. This could be regarded as an access to an abstract database (which in this case is the environment) as is done by some "query-the-user" facilities in expert systems (in which case, the database is assumed to be partially in the user's brain, partially inside the expert system).

The `assimilate` call combines the maintenance procedures for updating the database with the proof procedure that reduces the current goals to a set of new goals. The possibility of seeing an action as a goal, a search space as a hierarchy of goals, and a set of sub-goals as a plan provides a powerful, practical framework for modelling agents. As we shall see later on, an assimilation call, which is not allowed to last more than N reduction steps, obtains a more detailed, possibly executable, set of goals, from the previous set. The N refers to the number of inference steps in the bounded reasoning proof procedure. An ***executable goal*** is an *abduced* term that represents a *low-level atomic action* that could be executed by the *body* of the agent as a unit. A ***partial plan*** is a logical conjunction of executable goals with high-level (still to be reduced) goals. A ***full plan*** is a conjunction of executable goals only. Every ***set of goals*** (`Goals`, `NextGoals` and `NewGoals` in the previous programme) is, in general, a *disjunction* of goals. These goals can be conjunctions of non-executable sub-goals, partial plans or full plans.

The logical status of the `execute` condition is under investigation. In our system what *the executive* does is: the brain checks the set of goals received. If this set contains a partial plan, pick the plan, take the first (low-level atomic) action (scheduled for this time) and try to execute it. That is, the brain of the agent sends an order to its body to execute this action. The body tries to execute the action. Whether it fails or not, it will send and acknowledge (fail or succeed) message back to its brain. If the brain receives a succeed message, it *commits* to the current plan being executed, which will be considered for extension in the next cycle. That means that the alternatives to this plan at this stage, are forgotten. It is easy to see the reason. If the action succeeds, the agent (body and brain, of course) is now in a new situation in the world. The alternatives to its plan at this stage, are choices for the previous situation and have nothing to do with the new next action. On the other hand, if the action fails, the brain will receive a failed message and will throw away the current plan being executed, preparing one of its alternative plans to be considered in the next cycle. It is the responsibility of the brain's designer to ensure that there is always an alternative plan, or a way to obtain one. Finally, If there is no partial or full plan in the current set, the only action the body should execute (and that is always executed *attached* to the other actions) is sensing its environment. In this case, the set of goals will not be modified.

In order to give a logical interpretation to this process, Kowalski [Kowalski; 1994b] suggests regarding it as 1) using abduction to add to the knowledge base the assertion: the agent is trying to perform this action at this time, and 2) making this assertion available to the environment as an input to it. This would be equivalent to consider the world one more agent, an explanation which is not fully investigated yet. However, this could be similar to the message-passing semantics proposed by Hewitt [Hewitt; 1985], where *the meaning of a message is determined by how it affects the recipients*. Under Kowalski's interpretation there is a symmetry between the agent and the environment. The agent needs to assimilate inputs from the environment (observations) possibly rejecting the ones which are incompatible with is beliefs (an optical illusion would fall in this category, for instance). On the other hand, the environment needs to assimilate the outputs from the agent (its actions) which are regarded as input to the world. Infeasible actions (actions that can not be executed) could be seen as inputs rejected by the environment because they "are inconsistent with the *environment's beliefs*"[2].

The last condition in the locus of control algorithm is the `cycle` itself, which recursively invokes the next iteration cycle at a new time, with a new database and a new set of goals.

2This explanation is due to Prof. R. Kowalski.

We regard the `cycle` as an **adaptable locus of control algorithm** because the agent can iteratively update its beliefs and model of the world while is acting in this world. This continuous update or better said, assimilation, is the base of a rational, practical, adaptable behaviour. The agent is continuously *learning* about the world and changing its mind accordingly.

CHAPTER 2

SIMULATING AGENTS in PROLOG and APRIL

2.1. Modelling agents in the PROLOG-APRIL TEST-BED.

This project has consisted essentially of implementing models of multi-agent systems, using APRIL and PROLOG as building tools. The purpose of such models is to simulate agents' performances in a social context. As Ferber and Drogoul in [Feber, Drogoul; 1991] we depart from the traditional simulation techniques, based on mathematical and stochastic models, in the belief that these techniques are very limited to simulate societies. Specially in the case of societies of autonomous intelligent agents it is essential for researchers to *see* the specific behaviour of individuals: the actions they perform. This is missing in the traditional simulation framework where actions are only considered through their very final quantitative effects or by measuring their probability to happen. In our case, due to the fact that we are proposing an approach for specifying the behaviour of autonomous agents (for implementing artificial agents, indeed), it seems natural to try to related a program to an individual. Afterwards, we can simulate a world populated of interacting individuals of that sort. Of course, we still can collect statistics of these individuals and use them to infer features of the real world. We can also do a detailed analysis of the individual behaviour and its relationship with the global social behaviour.

Some people (see again [Feber, Drogoul; 1991]) argue that the classical stochastic simulation is an exercise of reductionism at macro-level, which cut off micro-elements and complex subtle components that affect the world being simulated. This eventually extends the gap between the simulation model and the real world. In contrast, the multi-agents' approach can capture the interactions among individuals, from which a global behaviour of the population emerges. We are still sceptical about the applicability of this approach to the modelling and simulation of arbitrary natural societies, but it is certainly useful and effective (and more faithful to the world than the classical simulation alone) for simulating open systems and systems composed of intelligent agents and human beings as operators.

As we said in the introduction, we use the services of the APRIL platform. The agents, whatever they are in the real world, are devices as APRIL agents together with PROLOG process whenever they require reasoning capabilities. The world, for instance, is models as a pure APRIL agent, not because it would be controversial to assign reasoning capabilities to the world, but because the database management capabilities we need from the world, can be implemented totally in APRIL. Therefore, in our applications the world is a central database with a manager. The other agents interact with this manager every time they want to execute an action.

An APRIL agent can consist of several APRIL processes active in parallel and synchronising themselves through the interchange of messages.

2.2. The path-finding and warehouse applications.

This project was somehow inspired by the work done in the InteRRaP project of DFKI [Muller, Pischel; 1993] Their approach is totally different to our, but the motivations are the same. We both address the issues of reactive agents versus deliberative agents, the world seen as a central agent and efficient, adaptable, search-control algorithms. Consequently, we decide to share with them also one application example, which is becoming a classical example. This project is aimed at implementing the warehouse simulation: A set of robots which work in a loading dock warehouse as forklifts, loading and unloading boxes from a truck to some shelves. The example is interesting because provides a framework where communication, cooperation and reasoning about other agents are all needed. However, the implementation of the test-bed, including the locus of control algorithm, and the extension made to PROLOG, proved to be complex enough to consume more than the time available for this first stage of the project.

We had to restrict the work to the implementation of agents who can *find their way* in a changing environment. They do not talk each other yet (hence, they will not cooperate) and the only actions they can perform are turning left or right or moving forward. However, they can see (with some very limited sensors) the world and therefore, they can decide how to avoid permanent obstacles and other agents in their way toward their goal places. Also, the object level knowledge required by the path-finders is particular suitable for a layered arrangement as we explain in the next section. We show experiments of this sort in the next chapter. The rest of this section is devoted to describing the world agent, the robots and their brains.

2.2.1. The World

As in the InteRRaP [Muller, Pischel; 1993] loading dock example, we employ a raster-based representation of the physical world. An array of cells *covers* the environment, each cell *representing* a significant portion of it. The sizes of the cells are adjusted in such a way that a robot occupies just one cell. The cells in turn are represented by a list of attributes that describe their content. Thus, we end with a simple database (one flat table) describing the world as follow:

Pos X	PosY	objec	type	status	bitmap
0	0	floor	floor	empty	floor
0	1	robot1	struc	free	rb_north
0	2	floor	floor	empty	floor
1	0	floor	floor	empty	floor
1	1	shelf	struc	empty	emp_shel
1	2	floor	floor	empty	floor
2	0	floor	floor	empty	floor
2	1	floor	floor	empty	floor
2	2	floor	floor	empty	floor

The world corresponding to this table is show in figure 2.1:

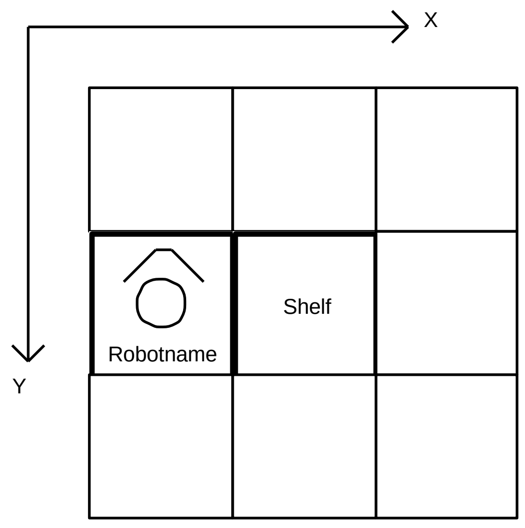


Figure 2.1. A rasterized world.

Note that our representation of the world is more complex than that. We devise an (APRIL) agent that manages the database, receives and serves action request by the agent, calculates and transmits to every robot the portion of the world inside its *range of perception* and triggers the animation. This APRIL agent is, in turn, constituted by two APRIL processes: a top process that performs all the mentioned activities in a centralised fashion and an auxiliary process that controls the animation with DIALOX. The following code is an outline of the first process written in APRIL:

```
world(handle?name,handle?interface){
 world_model?real_world := world_model?initial_world();

 /* The world main control cycle */
 repeat {
   [act, m_forward,
    integer?Cposx, integer?Cposy, symbol?Looking_To] =>
{
      real_world := moving_forward(real_world, Cposx,
                 Cposy, Looking_To,replyto,interface);
   }

 | [act, t_right,
    integer?Cposx,integer?Cposy, symbol?Looking_To] => {
      real_world := turning_right(real_world, Cposx,
                 Cposy,Looking_To, replyto, interface);
   }

 | [act, t_left,
    integer?Cposx, integer?Cposy, symbol?Looking_To] =>
{
      real_world := turning_left(real_world, Cposx,
                  Cposy,Looking_To, replyto, interface);
   }

/* This actions are not implemented in the current
version
 | [act, take,
    integer?Cposx, integer?Cposy, symbol?Looking_To] =>
{
      relax
   }

  | [act, put,
     integer?Cposx, integer?Cposy, symbol?Looking_To] =>
{
       relax
   }*/
  | [act, birth,
      integer?Cposx, integer?Cposy, symbol?Looking_To] =>
{
        real_world := being_born(real_world, Cposx,Cposy,
                     Looking_To, replyto, interface);
    }

  | [act, look,
      integer?Cposx, integer?Cposy, symbol?Looking_To] =>
{

      world_model?cp := world_model?(real_world^/[Cposx,
Cposy,symbol, symbol, symbol, symbol]);

      [[any,any,any,any,any,symbol?bm]] = cp;

      /* Build the new vision field of this robot */
       world_model?new_scene:=vision_field(real_world,
                              Cposx,Cposy,bm);
      /* acknowledge execution */
      [ok, Cposx, Cposy, new_scene] >> replyto ;
    }

  | any?Msg  => writef(stdout,
                          "Strange messagge %p\n",[Msg])
  } until end ;

};  /* of the world */
```

Briefly explained, this is a typical agent's control loop with message-guard tests, which allows the system to discriminate among the action-messages it receives, and to activate the database maintenance routines accordingly. These routines (APRIL functions most of them) perform the actions of the agents in the world and generate the responses for the *bodies* of the agents. The *succeed* responses always include an update of the *robot's vision of the world* (but only the current field of vision). The range of perception of the robots is limited in this first version to one cell (the front cell). However, this could be changed by modifying the `vision field` function. The `look` action is a motionless action. It does not modify the world database but only allows the agents to have a look at it. All actions are atomic in the sense that only one is executed at the time. The order of execution is established by the order of arrival of the messages.

This simulated world brings about the *structural constraints* in the robots' behaviour. The forklifts can move neither across structures (including themselves, of course) nor beyond the limits of the world. Despite this supervisory proficiency, the role of the world agent is still passive. The world does not affect the robot agents or itself by executing actions or generating events. Furthermore, the world agent does not influence the update of the robot's *vision of the world*: if one agent changes the database, the others will only know about it when they sense the world. The world will not inform those changes by its own agency.

The simulated world was originally designed to model the warehouse. Nevertheless, by changing the content of the file `world.db` (see appendix A) which stores the initial specification of the world (as is read by the `initial_world` function), any other *rasterized* world can be simulated. In fact, our experiments are made in a more simple world: the `patio`, which allows the robots more freedom of movement. To get an animation display in agreement with the world being simulated, the auxiliary process also needs to be modified, but the rest of the world agent stays the same. The appendices (See appendix D) also include some typical snapshots picked from the images of the animation.

2.2.2. Robots.

These are the active, deliberate agents of this application. The *forklifts* (we prefer to call them robots) are, as expected, more complex than the world agent, but also more than a typical APRIL agent. A robot agent is composed of two APRIL processes and one operating-system (UNIX) process, which runs a special version of SWI-PROLOG we call `plagent` (Prolog-Agent). The `plagent` executes a PROLOG program that controls the overall behaviour of the robot and that we call the robot's *brain*. The brain structure will be discussed in the next section, so let us concentrate on the other two (APRIL) process. The `robot` process, apart from being the locus of control of the APRIL semi-agent, works as the interface between the

(APRIL) body and the (PROLOG) brain. The other process is the `body`, and its structure is this:

```
body(handle?name,handle?worldname){
 repeat {
   [execute, symbol?nact,
           [integer?px,integer?py,
            symbol?dir, symbol?st, any?wm]] => {
      /* send execution request to the world */
      [act, nact, px, py, dir] >> worldname ;
      /* wait answer from the world */
      {  [ok, integer?npx,
           integer?npy, world_model?new_scene] => {
            symbol?ndir := newdir(dir,nact) ;
            /*action succeed*/
            [ok,[npx, npy, ndir, st, new_scene]] >> name
         }
      |  [failed, action?failed_act] => {
            [failed,failed_act] >> name
         }
      }
    }
 |  any?Msg => writef(stdout,"strange message %p",[Msg])
 } until die ;
};
```

At this stage should be clear that, in the simulated world, all the actions are accomplished by interchange of messages that transmit the actions, the results and information about the new state of the world. It is important to note, however, that the robot agent is responsible for deducing and keeping track of its new state as a consequence of those actions. The world sends back position status (x, y) of the robot for the sake of efficiency (otherwise, these calculations should be repeated by the robot itself). Other attributes of its current state (the direction the agent is looking to, obtained by the `ndir` function, for instance) are locally deduced. It is always possible to exploit (and to study) the trade-off between using the world central database for storing agents' attributes or to locate those descriptions inside each agent.

This specification of the robot agent as two separate APRIL processes is pursued, not only for its instructional utility, but also because it allows a more straight-forward implementation of the communication module. The `ear-mouth` module, to handle human-like conversations and debates with other robots, should run separated from the rest of the body.

2.2.3. The brain of the robots.

As we said in the introduction, we could implement agents only in APRIL (even with similar deliberate capabilities). The reason for using PROLOG is to show that an agent's *brain* can be completely specified (and implemented) in logic and that a theorem-prover can be as useful as specially designed programs for controlling reactive autonomous agents. Thus, we sacrifice the probable efficiency of a full APRIL implementation to get the possibility of employing a language that can be used to describe the *object level knowledge* about the world, together with the *meta-level knowledge saying how to think and* ***assimilate***. Consequently, the brain of the robot is a PROLOG program that contains: 1) an image of the world database, probably partial and out of date, accessed through the predicate `cell/6`; 2) a set of clauses establishing the relations among goals and actions in a particular domain; 3) the code of the meta-interpreter that implement a sound proof procedure for reducing goals to actions and 4) the locus of control algorithm presented in chapter 1. We must say that, once again, the clause form of logic proved useful for specifying an *information system*, in the sense of [Kowalski; 1979] (Chapter 13), but which can be regarded now as an ***open system***.

The first and last parts of the brain have already been explained. The second is going to be detailed in the next section. The part 3, we present the interpreter of PROLOG that we use to process the object level clauses written in PROLOG syntax. This interpreter is itself written in PROLOG which is why we have to call it a PROLOG meta-interpreter. This meta-interpreter is the implementation of a *bounded SLDNF with abduction proof procedure* (see [Hogger; 1990] for further details about SLDNF). The *bounded* adjective refers to the fact that the meta-interpreter can perform the reduction in the goals' hierarchy until some predefined extend. This is accomplished by providing the `resolve` predicate, the top level procedure of the meta-interpreter, equivalent to Kowalski's `demonstrate` [Kowalski; 1979] and Shoham's `meta` predicates [Shoham; 1994], with an extra argument controlling the *depth* of the search.

To say that the meta-interpreter implements SLDNF resolution is not enough to describe it. SLDNF does not enforce a particular computation rule or search rule. For both of these we use the PROLOG rules. Our meta-interpreter selects terms in clauses from left to right. Also, clauses are considered in the order in with they appear in the object-level database (text order). However, unlike other meta-interpreters, we do not use the implicit PROLOG backtracking. We build a list of the alternative *branches of the search tree* and use this list to select the next branch to be extended. The reason for this explicit list is twofold: 1) We need to keep a record of the current state of the reduction because, due to the depth constraint, we will have to *suspend* the resolution, probably before the current branch get completed; 2) The control of the resolution is going to be transferred to another procedure: the *executive*. This executive continues the resolution process by reducing an executable goal (defined in Chapter 1), if there exists one in the branch being extended. The function of the executive can be seen as testing the validity of an abduced predicate (the executable goal: `do`) and asserting its truth-value according to its effect in the world. If the action succeeds, the truth-value of `do` term is true and therefore, the current branch (partial or full plan, defined in Chapter 1) can be further extended. If the action fails, the `do` term is false, and this falsifies the current branch (conjunction) being extended. In this case, the executive has to rule out the current branch and *backtrack* to an alternative branch. Thus, the backtracking can be effected either by the normal `resolve` (when some condition does not hold) or by the executive (when an action fails). Observe that the executive is invoked by the *locus of control*. In addition, between the main cycle and the meta-interpreter stands the *assimilation procedure*. Immediately below, we show the actual codes of the `assimilate` and `resolve` procedures. Observe that the list of alternative goals (or-list) is represent as a ordinary PROLOG list (between square brackets) whereas a conjunction of sub-goals is represented as a PROLOG and-list and therefore, ends with `true`. Once again, we preferred legibility instead of efficiency in our code.

```
assimilate( Input, Gs, NewGoals, T, N, Tn ) :-
 update_world_model( Input ),
 resolve( N, Gs, NewGoals ),
 Tn is T + N.


resolve( 0, Goals, Goals ).              /* Base cases */
resolve( _, [(true)], [(true)] ).

resolve( N, [true|Rest], NGoals ) :-
 NNext is N - 1,
 resolve( NNext, Rest, NGoals ).
resolve( N, Goals, NGoals ) :-
                        /* demostrating upon the current KB */
 Goals = [FirstAlt|RemAlt],
 demo( FirstAlt, NewSet ), /* G <-> body1 or body2 .. */
 append( NewSet, RemAlt, NextGoals ),
                                          /* depth first search */
 NNext is N - 1,
 resolve( NNext, NextGoals, NGoals ).

resolve( N, [_|RemAlt], NGoals ) :-   /* backtracking */
 NNext is N - 1,
 resolve( NNext, RemAlt, NGoals ).


demo( true, [(true)] ).

demo( (true, R), NewGoals ) :- !, demo( (R), NewGoals ).

demo( (not G, R), [(R)] ) :- not demo( G, _ ). /* NF */

demo( (G, R), [(R)] ) :-
 predicate_property(G, built_in), !, G.
                                      /* Builtin predicates */

demo( (G, R), NewList ) :-               /* Abducibles */
 abducible( G ), !,
 push( G, [(R)], NewList).


demo( (G, R), NewList ) :-  /* Object level clauses */
 findall( BB,
         (clause( G, Body ),and_append( Body, R, BB ) ),
          NewList ),
 ( NewList = [] -> ( fail, ! ); true ).
```

Note that we do employ abduction (fourth `demo` clause), but it is not *constructive abduction*. The meta-interpreter decides if a particular term can be posted as a *suggestion* (abductive explanation) to be tested by the executive, but it does not generate that *suggestion*. It is responsibility of the programmer-designer of the object level database, to be sure that the meta-interpreter receives a grounded abducible term. This provision is similar to that used to avoid *floundering* in programs with negation as failure and thus most logic programmers are used to it.

In our system, the use of negation as failure (third `demo` clause) is justified by the need to commit to actions, making quick decisions with the available information. Thus, the robot will, for instance, decide to move to the next cell if, either its model of the world says that it is an *accessible* cell or there is no information about it. The `prohibited` predicate, a condition that is part of the object-level database (see appendix B), tests whether a cell is occupied by one structure and therefore it is not accessible. In addition the robot assumed that a cell is *accessible* whenever it can not be prove that it is *prohibited* (the cell is `not prohibited` as far as the robot knows. This is the close world assumption of negation as failure). If this assumption is wrong, that will be corroborated at execution time when the assimilation procedure will make the adjustment needed to the current set of goals and to the database.

We believe this is an appropriated model of intelligent decision-making. After all, we human beings make decisions in the same way, probably changing our minds when we come to realise the world. We also believe we address the objections of Hewitt [Hewitt; 1985] to the use of the close world assumption in open systems, by allowing our ***creature*** [..] to *cope appropriately and in timely fashion with changes in its dynamic environment*, as Brooks has required [Brooks; 1991]. In the next sections, we will review the other requirements for artificial creatures that Brooks has established, to see whether or not our robots satisfy them.

2.3. Backward-forward representations.

In this section we discuss the second part of the *brain information system*: the set of clauses establishing the relations among goals and actions in some domain. We try to answer the question: how can specify knowledge relate to a particular domain, in such a way that an agent can employ this knowledge to guide its actions?. In other words: *How to program an intelligent agent?*. Obviously, we are not the first to make this question or to suggest an answer.

The need to act, probably within some period of time, has always been a major problem for the designer of intelligent agents. Arguably, this could be the reason for the success of condition-action production rule systems or more recently patterns-with-associations [Newell, Young, Polk; 1993] were the knowledge is pre-compiled to allow more *flexible contingent response*. In this sort of system, a set of rules contains the knowledge relating conditions in the world with actions to be performed. Whenever all the conditions in a rule hold, the rule *fires* and so, *triggers* the execution of its related action, which can produce changes in the world or inside the agent itself. These changes can make other conditions hold (or not hold). If several rules fire, the system has the extra work of picking one, probably appealing to some meta-level information. Thus, production-rules can be used to model reactive intelligence.

Although we do not deny the appropriateness of production rules, we believe, following Kowalski [Kowalski; 1994b], that they correspond to normal logic programs whose goals have been put out of sight by pre-compilation. We take this believe a little further and propose to use logic programs to specify the behaviour of the agents by describing the full hierarchy that relates goals and conditions with actions. Some implementation tricks are required to obtain the required efficiency. However, those are only implementation issues, as solvable in the production rules framework as with the logic programming approach. In exchange, the programmer gets the greater expressiveness of logic programming with its declarative and procedural readings.

What we call a hierarchy of goals is simply an abstraction of the SLD-tree. The root of the tree is a high-level goal. The intermediate nodes are sub-goals. The leaves of the tree are either low-level conditions or actions (executable-goals). The system maintains a list of conjunctions, as the one described in Chapter 1, to store the current *set of leaves of the SLD-tree*. During the process of reduction, this list is updated accordingly. To illustrate the resolution process, we immediately include a fragment of a typical evaluation, showing the evolution of the list in PROLOG-like notation (uppercase letters represents variables). This example employs the rules involving `go` goals. Note that, for simplicity, only one number denotes a geographical situation (instead of the normal X-Y representation). The lowercase letter beside the number indicates the direction toward which the agent is looking:

```
[(go(3n,5e))]
↓
[(gradient_steps(3n, Z, 5e), go(Z, 5e) ) ;
(avoid_steps(3n, Z), go(Z, 5e)) ]
↓
[(atom(A, 3n, Z), closer(3n, Z, 5e),
 not prohibited(Z), do(A), go(Z, 5e)) ;
(avoid_steps(3n, Z), go(Z, 5e))]
↓
[(closer(3n, 0n, 5e), not prohibited(0n),
 do(m_forward), go(0n, 5e)) ;
(closer(3n, 3e, 5e), not prohibited(3e),
 do(t_right), go(3e, 5e)) ;
(closer(3n, 3w, 5e), not prohibited(3w),
 do(t_left), go(3w, 5e)) ;
(avoid_steps(3n, Z), go(Z, 5e))]
↓
.
.
↓
[(not prohibited(3e), do(t_right), go(3e, 5e)) ;
(closer(3n, 3w, 5e), not prohibited(3w), do(t_left),
 go(3w, 5e)) ;
(avoid_steps(3n, Z), go(Z, 5e))]
↓
[(do(t_right), go(3e, 5e)) ;
(closer(3n, 3w, 5e), not prohited(3w), do(t_left),
 go(3w, 5e)) ;
(avoid_steps(3n, Z), go(Z, 5e))]
```

Looking top-down, the first list contains only one single-term goal: `go(3n, 5e)`, which should be read as the agent having the intention to go from cell 3 (where it stands looking to the North) to cell 5 (where it will be looking to the East). This single-term goal is replaced during *the first step of reduction* by two alternatives set of sub-goals (separated by semi-colon). Note that each one of those set of sub-goals is a conjunction of terms (the aforementioned *and-list*). The terms `gradient_steps/3` (from steps in the gradient direction) and `avoid_steps/2` (from steps for obstacle avoidance) represent different strategies for *generating* the next action of the agent. The next resolution step (next list) substitutes the first sub-goal in the first set (`the gradient_step/3` term. This is *depth-first* search) with its definition. Within this definition one can observe several kinds of terms. The `atom/3` term, at the beginning, in a *generator term* because it generates and action that could be executed in the current situation (`3n`) and instantiates the variables accordingly (`A` with the action, `Z` with the next situation). The `closer/3` *condition* tests those situations generated by atom to decide whether each one of them (see next list) is closer to the final situation than the current situation. Indeed, `closer/3` implements our notion of gradient field (see the source-codes in appendix B). The `not prohibited/1` sub-goal was presented in the previous section. It is a condition for checking that a particular situation, generated by `atom/3` and tested for closeness by `closer/3`, is indeed *accessible* (the agent can walk through it). Finally in the definition appears the `do/1` term. This is the *executable goal* previously described (Chapter 1). It is a term that will be *offered* to the meta-interpreter for abduction. Observe that the variable `A` (specific low-level action) in `do/1` is instantiated by the precursor `atom/3` term. Eventually, a `do/1` term appears as the first term in the first list (see last list in the example) ready to be executed (*the executive* also works in a depth-first fashion). What happens next depend on the *sort of brain* involved (reactive or planner as will be explained in Chapter 3). We provide a more comprehensive explanation on the structure of these domain-specific rules in the following section.

Although the details about what it is a generator term, a condition or an abducible term are important, the fundamental idea of this representation is the use of what we call *situated goals* to perform a ***forward search within a backward reasoning scheme***. Almost every sub-goal term (excepting only the abducible terms and the non-logical terms used in normal PROLOG programming) carries a description of the current situation, which may be updated to a new situation when the sub-goal implies the execution of an action (recall in the previous example `go(3n,5e)` means the agent is now in `3n` and when the goal is accomplished will be in `5e`). Note that what we call *situation* is not necessarily a complete description of the current state of the robot. In this case what it is relevant to the goal `go/2` is the geographical condition of the agent (coordinates X and Y and direction toward which it is looking). It does not matter, for instance, whether or not the robot is carrying a box.

This knowledge database (goal hierarchy) is designed in such a way that the high-level goals are defined in terms of movements from a *well defined* initial situation toward a final situation. It could be seen as the forward reasoning of condition action rules implemented in a backward reasoning framework. We insist on the point because it has been mistakenly believed (specially in some non-academic circles) that the backward reasoning proof procedure implemented in PROLOG can not be use for forward reasoning. Our programs show that it can be used. Moreover, we are not the first to adopt such representations (see for instance the DCG rules in [Covington; 1994]).

Let use a classical example to show the universality of this representation. A program for building towers in the *world block* might include the following clauses (in PROLOG syntax):

```
build(InitialTower, FinalTower) :-
      put_block_on(InitialTower, NextTower),
      build(NextTower, FinalTower).

build(InitialTower, FinalTower) :-
      take_block_from(InitialTower, NextTower),
      build(NextTower, FinalTower).
```

The predicate `build/2` represents the action of building the tower described by `FinalTower`, starting with a tower (that may be a null tower, i.e. the empty table) described by `InitialTower`. How efficient (optimal) is the program depend on the actual implementation of the procedures `put_block_on/2` and `take_block_from/2`, which represents the actions of putting a block on top of the current tower and taking a block from the top of it, respectively. Of course, the process ends when `InitialTower` is equal to `FinalTower`. The key point is to discompose the problem is such a way that the situation description progress toward the solution while the reasoning is accomplished in backward manner.

2.4. Search by layers.

It is worth noting that in the brain programs, as in normal PROLOG programming, *the branching points of the search tree* can be arranged in such a way that during the search, the agent explores first the most promising branches. Furthermore, the branches could be used to represent *layers of control* in the sense first proposed by Brooks [Brooks; 1986] and applied by Steels [Steels; 1990], as we will show later.

In that last reference, one can find a proposal for building reactive agent that advocates the use of pattern of behaviour against explicit (logical) knowledge representation. Steels employs Brooks' subsumption architecture [Brooks; 1986] in order to model and simulate reactive robots in an application very similar to our path-finder example. A fragment of the Steels' robots subsumption architecture is shown in figure 2.2

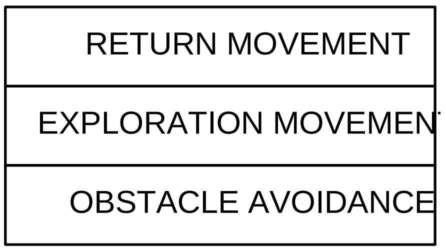


Figure 2.2. Steel's agent architecture.

Steels identify layers with behaviours, which in turn are defined as follow [Steels; 1990]:

Return movement behaviour:

If I am in return mode I chose the direction of the highest gradient.

Exploration movement behaviour:

If I am in exploration mode I chose the direction with the lowest gradient.

Obstacle avoidance behaviour:

If I sense an obstacle in front, I make a random turn.

It is not difficult to argue that these definitions are ordinary condition-action rules. Steels insists that "*there is no sophisticated control strategy or internal reasoning of any sort"* behind this architecture. The layers run in parallel. The only important point about control is that lower-level edicts take precedence over upper-level ones. That is, whenever there is an obstacle at the front, is the obstacle avoidance layer which dictates the actions to take. There are not further details in the reference to explain how to resolve other conflicts. However, some other rules are added to guide the action of the robot by determining its *next mode of operation*:

Mode determination:

If I am in exploration mode and I sense no lower concentration than the concentration in the cell on which I am located, I put myself in return mode.

If I am in return mode and I arrive at my destination, *I put myself in exploration mode*.

If I am holding a sample, I am in return mode.

Let us present the logic programming specification of our path-finder robot in order to compare it with the Steels' architecture. A path-finder robot "*lives"* to find a way of going somewhere. Consequently, one top level goal of our robot may be logically stated by the predicate: `go(InitialSituation, FinalSituation)`. We fulfil one of the Brooks' requirements for artificial creatures: "*A creature should do something in the world; it should have some purpose in being*" [Brooks; 1991]. We link this overall goal with more *every-day* goals and actions by means of the following PROLOG program:

```
go( InitialSit, FinalSit ) :-
       gradient_step(InitialSit, IntermSit, FinalSit), go(IntermSit, FinalSit).
go( InitialSit, FinalSit ) :-
       avoidance_steps(InitialSit, IntermSit), go(IntermSit, FinalSit).
```

Where the arguments of the `go/2` predicate are descriptions of an initial situation and a final situation, the `gradient_step/3` clause is a subgoal forcing the agent to make a step in the direction of certain gradient field and the `avoidance_steps/2` dictates a *sequence* of actions to avoid obstacles.

We claim that this program generates in our simulated robots the same behaviour that Steel's reports from his. The first `go/2` clause is equivalent to the return and exploration movement layer together because, once a final situation is established, the agent always tries to follow the shortest way (the direction with the maximum gradient) to its destination. The second one is the logic programming representation of the obstacle avoidance layer. According to the PROLOG search rule we use, the first clause is considered first. If there is an obstacle to be avoided, the conditions behind the first clause will not hold. As a consequence, the search will continue with the next clause that specified the correct behaviour in that situation. This is the principle of *search by layers*.

It is important to remember that the clauses can be ruled out either at *thinking time* or at execution time. Therefore, even if `resolve` mistakenly *uses* the `gradient` clause, the executive will force backtracking afterwards. Certainly, this logic programming representation does not allow parallel exploration of clauses. Nevertheless, nothing apart from the need for choosing only one final action (that also exits in the Steel's architecture), excludes parallel search throughout the alternative branches. This is possible because our locus of control keeps all the alternatives to particular goal at any moment. The set of goals, the *list of and-lists*, used for controlling backtracking is, precisely, a set of disjunctive sub-goals (an *or-list*) which are alternatives to the current goal. This could be seen as having multiple goals at every stage. "*Depending on the circumstances it find itself in,* [the robot] *will change which particular goals it is actively pursuing*" [Brooks; 1991], another of the requirements established by Brooks and that we have been enumerating.

As a final point in our presentation of the path-finder architecture, let us present the full definition of the `avoidance_steps` sub-goal, which will allow us to highlight some important aspects of our representation:

```
avoidance_steps( CurrentSit, NextSit ) :-
 change_dir( CurrentSit, NewSit ),
 avoidance_step2( NewSit, NextSit ).



avoidance_step2( CurrentSit, NextSit ) :-
 move_forward( CurrentSit, NextSit ).

avoidance_step2( CurrentSit, NexSit ) :-
 avoidance_steps( CurrentSit, NexSit )



change_dir( (X, Y, north), (X, Y, east), T ) :-
 do(t_right, T).
change_dir( (X, Y, south), (X, Y, west), T ) :-
 do(t_right, T).
change_dir( (X, Y, east), (X, Y, south), T ) :-
 do(t_right, T).
change_dir( (X, Y, west), (X, Y, north), T ) :-
 do(t_right, T).



move_forward( (X, Y, north), (X, Ny, north) ) :-
 Ny is Y - 1, not prohibited( (X, Ny, north) ),
 do(m_forward).
move_forward( (X, Y, south), (X, Ny, south) ) :-
 Ny is Y + 1, not prohibited( (X, Ny, south) ),
 do(m_forward).
move_forward( (X, Y, east), (Nx, Y, east) ) :-
 Nx is X + 1, not prohibited( (Nx, Y, east) ),
 do(m_forward).
```

```
move_forward( (X, Y, west), (Nx, Y, west) ) :-
 Nx is X - 1, not prohibited( (Nx, Y, west) ),
 do(m_forward).
```

The `avoidance_steps` rule states that the actions the robot will take consist of a change of direction (`change_dir`) and a second action determined by `avoidance_step2` rules The next action, established by `avoidance_step2`, is either move forward (`move_forward`) or try a new change of direction, deduced through a recursive call to the `avoidance_steps` rule. Observe that when the robot is surrounded by other structures it keeps changing directions until it get and open space (which can never happen).

We also include the definitions of `change_dir` and `move_forward`. It is important to note that every action is related to a situation of the robot (the situation with respect to its goal of going somewhere). Thus, the code is carefully designed to guarantee that the robot always knows where it is. Every atomic action (proposed for abduction by the `do` predicate) should be linked to a situation in such a way that if the action succeeds or fails, the robot will be able to deduce its next situation. The fundamental point is that an over-simplified rule, as the following (presumably closer to the equivalent condition-action rule), would not be correct:

```
avoidance_steps( CurrentSit, NewSit ) :-
 /* follow the wall */
 change_dir_move_forward( Turn1, CurrentSit, NewSit ),
 not prohibited( NewSit ),
 do( Turn1 ),
 do( m_forward ).
```

The first term `change_dir_move_forward/2` generate the actual direction for turning and calculate the new situation. This new situation is tested by `prohibited/1` and the two `do/1` terms are presented to the executive. Consider what happens when the `do(m_forward)` fails. The robot could not change its situation (it is still at `CurrentSit`). However the argument of the next term in the goal conjunction is pointing out to the new situation (`NewSit`). The robot will eventually lose track of its situation *with respect to its goal*.

We are conscious of the identification of actions and goals we impose upon our representation. This is not, however, an actual constraint. One could say, for instance, that the goal of a path-finder is not to go from one point to another but, instead, to be at certain place (at certain time). This possibility is not excluded in the representation. An appropriate clause for capturing the idea, could be added to hierarchy, raising its level as follow:

```
at( FinalSit ) :- current_sit(InitSit),
                         go(InitSit, FinalSit).
```

However, the designer needs to be alert in order to prevent tragic errors. For example, one could be tempted to exploit the easy definition of recursive procedures in clausal form, in order to program a permanent or long lasting behaviour into the robot. Something like this, following the same idea in the previous clause:

```
move_around( SomeSit ) :-
       current_sit(InitSit),
       go(InitSit, SomeSit), move_around( InitSit ).
```

This *rule* is intended to cause the robot to go from one place to another indefinitely. Unfortunately, in brains with *planning* capabilities (see next chapter for the definition) and a description of the `current_sit` restricted to the present time, this rule is misleading. To see the reason, consider what happens when the resolver has reduced all the sub-goals up to and including `go(InitSit, SomeSit)`. In the next reduction it will select the clause itself, which carries a call to the `current_sit` predicate. This call, however, refers to a situation in which the robot will be later, not the one recorded in its *current mental context*. The basic problem is that the database in the robot's brain does not describe the world and its dynamic, it simulates the world. We require a way of *predicting* situations of the robot, reasoning about the fact that it is going to achieve certain goals. We need the robot to project itself into the future.

2.5. Time and events representation in logic programming agents

Although this weakness of the representation, as we have described it so far, could be discouraging for the logic programmer, it can provide the practical context to introduce a powerful tool employed in more sophisticated agents: time management [Shoham; 1994] and/or event representation [Kowalski, Sergot; 1986]. We have been looking after the rationalism for including time representations in agent's brains. At first sight, it seems to be that, unless other external agencies (human operators or users, for instance) mandate the use of time, for example by limiting the periods of time for achieving goals, such a necessity does not arise by its own in these agents. Even some practical exercises of cooperation can be solved without an explicit account of time (i.e. *to past through the narrow corridor, you go, he goes and I go* in that order). If the forklifts want *to unload the truck as soon as possible*, they can simply negotiate and distribute the tasks and pick the plan with the minimal number of atomic actions. This could be done even if the goal is *to unload the truck before lunch time*. Some *more intelligent* agent (maybe a human) could evaluate the robots' global plan, to see whether or not it can be accomplished before the lunch gets cool.

However, as we said in the previous section, sometimes (for example in narrative understanding) the agent needs to deal with goals that are not equivalent to actions (e.g. to *go to*) but, instead, to situations (e.g. to be at *some place*, at *some time,* to have *something*, during *some period*). We must admit that our current agent model does not permit the agent to distinguish its current situation from previous and future ones. In other words, our agent can not reason about what the situation was sometime ago or will be in the future. As in conventional databases, the agent updates its beliefs about the world in the same order (and also almost at same time delayed only by the speed of perception) as the corresponding events occur in the real world. That is why we say the agent's brain *simulates* the world. As the robot only represents the present moment, it can not accept new information about the past or the future available from the environment or from other agents. Hence, the agent is missing part of the world's experiences which, otherwise, could be available for a more accurate deduction of actions and goals.

We human beings do that *situation analysis* in order to learn from past experiences and to decide the better way for behaving in the future in pursue of some goal. We also use that capability of self-location in the space-time for synchronising ourselves with nature and with other beings (eat breakfast just after dawn, lunch at 12.00 and dinner when she gets home, then rest between 11.00 p.m. and 5.00 a.m.). There is an important advantage in using a watch or an event notifier. We do not need to worry about the overwhelming amount of detail about actions, order of actions or events of other beings and nature that, in addition, happens in parallel to our own actions. Our own organism is a set of processes running in parallel. We do not follow them explicitly. We simply *meet them* at certain points in time with as much precision as the state of the art in time-keeping allows. We believe all these capabilities could be provided to artificial, intelligent agents within the logic programming framework.

CHAPTER 3

EXPERIMENTS WITH THE PROLOG-APRIL TESTBED

3.1. Comments on multi-agent simulations.

As we said at the beginning of Chapter 2, part of the motivation to simulate an agents' society in this non-traditional simulation framework, comes from the opportunity to model *micro-behaviour* (behaviours of individuals) and to measure its impact on global behaviour. This last, however, is not a straightforward task with a test-bed like ours. Repetition of experiments, for instance, is fundamental in an environment where real randomness is used instead of the pseudo-random computer generated sequences. Due to the fact that our agents' messages almost always travel throughout a real network, we deal with real random delays. In such conditions, any conclusion made from the results is at risk of being mere speculation, unless the supporting data can be proved to be representative enough.

In contrast to the call for more testing it is the quantity of information to be analysed. The amount of details that can be recorded, already overwhelming in macro simulations, is even greater in micro-simulation. There is also, the additional problem that some of the statistics need to be obtained by individuals and post-processed into global hypothesis afterwards. This post-processing can imply loss of grades of freedom in the final statistics and, consequently, less confidence. We bore all those influences in mind during the tests, and also encourage the readers to consider them when they come to interpret the results.

3.2. Description of experiments with the path-finders.

Taking the simplistic path-finder agents (and the `patio` world) as *guinea-pigs,* we concentrated attention on two types of experiments: 1) measures of real machine performance of our agents and 2) measures of adaptability of the agents to environmental conditions. The first set of experiments is aimed to compare the running requirements (time and memory space) of two models of brains: the reactive brain and the planner-brain. The second set obtains data to evaluate the effect of *deep thinking* (recall the parameter N used to bind the depth of search) into the behaviour of an agent that has to interact with a changing environment. We *build* a changing environment by setting goals to the agents that make them interfere each other.

Before showing the details of each group of experiments, some concepts need explanation. What we call a **reactive brain** is the program described in Chapter 2. The important point to be noted is that, in the reactive brain, *the resolver* reduces the goals until it gets the first `do` predicate pointing an action to be executed. It does not try to reduce the other goals *after* the `do`, so there is no further planning. The control is transferred to *the executive* which either executes the action, confirming the current conjunction of goals for further refinement, or backtracks to another conjunction. A reactive brain may or may not decide an action to be executed, depending on how deep the parameter N allows it to get. On the other hand, unlike the reactive, the **planner brain** does worry about sub-goals after the `do`. In fact, the planner brain tries to obtain as `do` predicates as possible given some N, getting in some cases to build several *full plans* in the sense explained in Chapter 1. Various modifications have been made to the program shown in Chapter 2, to cope with the planning-ahead capabilities of this brain. Some of them are: 1) changes in the `demo` clause devoted to abducibles, allowing the process to skip over previous obtained `do` terms; 2) changes in the `resolve` clauses to skip over the full-plans obtained in previous cycles, if any exits. The planner brain also may not decide an action to be executed, but this is unlikely to happen because it capitalises on reductions made in previous cycles. Certainly, planning can demand enormous spaces of memory. However, once again, the depth-parameter N can be use to control and to restrict resources. The source-codes of both brain programs are included in Appendix B.

3.3. Redesign of experiments

As soon as we started to carry out the experiment described in the previous section, several issues arose. The first thing we realised is that *adaptability*, how the agents integrate themselves into the changing environment and *performance*, how well they accomplish their goals in that environment, could be correlate and, therefore, we need to test them with the same set of experiments. On the other hand, during the first experiments, our first model of the planner brain showed itself to be awfully inefficient. One run of one robot alone, with N set to 70, took 1 hour and a half, with the reference robot *living* only for 50 cycles (See the experiments' specification in the next paragraphs, for comparison) whereas the reactive-brain agent could make the same work in 4 minutes. A careful analysis of the planner brain structure let us realised that, in the original design, we had missed or misunderstood one important aspect of a long-lasting agent's brain.

In the planner brain we employ abduction for building ordered sequences of *executable goals* (`do` terms) which can be regarded as detailed plans. One valid sequence, in our path-finder application could be (as before, we use a simplified notation): `do(m_forward), do(m_forward), do(t_right), do(m_forward), go(0e, 4n)`. This sequence is a *partial plan* (see previous Chapter for definition) that the agent can use to go from some initial situation to the situation `4n`. The basic problem in our first model of planner is that the agent keeps extending these sequences *indefinitely*. At every cycle, the brain skips over the previously abduced `do` terms and tries to solve the other high-level goals. Considering that the robots have permanent goal hierarchies, most of the time recursively defined, this turns out to be a never-ending extension process. It is always possible to extend the robot's plans, even with a very low N. The agent still shows an acceptable performance for low N (N = 10..20 in our application), because the abduced terms are generated almost at the same speed as they are consumed by the executive. Therefore, there is no memory overflow, as it is the case when N is big (N > 30 in our application). Our system does not explode, but the garbage collection is so intense that dismisses any hope of efficiency. As if that is not enough, that scheme does not seem to be a realistic model. We human beings, for instance, do not go into details about those plans whose execution is too far in the future. We can specify the constituents of a plan until certain point and then *complete* it with more abstract goals (i.e. in the morning: get up, put on your clothes, brush your teeth and then go to work). How far to go specifying plans in that way is an application dependent issue, yet it is not carried on forever.

Following the previous line of reasoning, we decided to modify the planner brain schema to limit the length of the detailed part of the plans. The strategy is simple: in the same way we use a parameter to limit the depth of the search (***bounded reasoning***), another parameter may be used to control the size of the set of abduced terms at any time (***bounded abducibility***). Both parameters should be adjusted by the designer in agreement with the available resources and the application domain constraints, including the need for quick response. In pursue of this, we implement bounded abducibility altering the code by adding some special procedures and predicates to the knowledge base, namely `abd_count/1, init_abd_limit/1, up_abd, down_abd, reinit_abd`. The final effect is that the brain can plan ahead until it gets certain number of abduced terms. We set this number to 10 in our experiments. That was enough for our robots to build a set of steps for taking itself from the initial situation to the extreme situation. Observe that this is not a complete plan. The goal of the agents in the experiments is recursively defined as follow:

```
go_around( So, Sf ) :- go( So, Sf ), go_around( Sf, So ).
```

Therefore, a *full plan* for this goal is a never-ending sequence of abducible terms. The attempt to generate such plan causes the overflow of memory we want to avoid with bounded abducibility.

It is really worth noting that the reactive model of brain becomes a special case of a planner brain, whose abducibility parameter has been set to 1. This implies an even smoother transition between reactive and planning ahead planners in this framework. Accordingly, we also decided to include in the experiments, tests comparing the behaviour brought about by the pure-reactive brain and the planner brain. The Appendix B contains the source-codes of those brain schemes.

The unified set of experiments obtains information about performance, failures and hesitancy of the agents. The overall purpose is to evaluate the effect of bounded reasoning (N parameter) on the behaviour of individual agents and on the whole group. At the same time, we survey the behaviour of pure-reactive and planner robot's brains, following the guidelines discussed in the previous paragraphs. In all the experiments the hardware of the test-bed is configured as follow: 2 SparcStation ELC, with 16 MB RAM memory each one, connected via nfs with a Disk Server. The `plagents` (SWI-PROLOG) use 2 MB of RAM memory for the global stack, 4 for the local stack, 2 for the trail stack and 1 for the arguments' stack (those are the maximum values permitted). One machine executed the DIALOX server and the `patio world`. The other machine executes the robots' processes. Recall that a robot is composed of one `plagent` process and two APRIL processes. Another machine runs the APRIL nameserver. The experiments are:

- **Experiment ONE**: For every test (for N = 10, 20, 30, 40, 50, 60, 70, 100 and 200), for every brain scheme (reactive, planner), simulate a robot alone in the patio world, going from initial situation to extreme situation. Record the number of **failure steps per trip per robot**. That data will be used to estimate the average of failures per trip and to correlate the number of failures with N. It is worth noting that, in our simple patio and warehouse worlds, the only action that can fail is the *move forward* action. Turning left or right is not restricted. Thus, what we are trying to determined is *how frequently a robot is blocked by its fellows* in its attempt to move forward. Also record, **the number of observations the agent performs per trip**. These values are used to estimate the level of hesitancy of the agent, that is, *how frequently the agent does not decide the next action to* *be executed*. The agents' insecurities are due to the fact that the agent may not reason long enough, thus hesitancy is a function of the parameter N. Finally, record **the time the reference robot (ja) takes to**

reach 200 cycles of *life*. All the agents stop at the same time and, therefore, these time values can be used to estimates the average performance of the agents.

Description of agent ja:

Robot name: ja

Initial situation: X=0 ; Y=5; Looking toward = north.

Extreme situation: X=4; Y=3; Looking toward = east.

High-level goal: permanently go between initial situation and extreme situation, following the shortest path.

Length of minimal path: 5 step between initial situation and extreme situation.

- **Experiment TWO**: Repeat experiment ONE with two robots (ja, jb). The simulation runs until the reference robot reaches 200 cycles.

Description of agent jb:

Robot name: jb

Initial situation: X=4; Y=5; Looking toward = north.

Extreme situation: X=0; Y=3; Looking toward = east.

High-level goal: permanently go between initial situation and extreme situation, following an optimal path.

Length of minimal path: 5 step between initial situation and extreme situation.

- **Experiment THREE**: Repeat experiment ONE with three robots (ja, jb, jc). Again the experiment last until the reference robots reaches 200 cycles.

Description of agent jc:

Robot name: jc

Initial situation: X=0 ; Y=1; Looking toward = south.

Extreme situation: X=3; Y=6; Looking toward = north.

High-level goal: permanently go between initial situation and extreme situation, following an optimal path.

Length of minimal path: 8 step between initial situation and extreme situation.

- **Experiment FOUR**: Repeat experiment ONE with four robots (ja, jb, jc, jd). Stop when ja reaches 200 internal cycles.

  Description of agent jd:

  Robot name: jd

  Initial situation: X=4 ; Y=1; Looking toward = south.

  Extreme situation: X=0; Y=6; Looking toward = north.

  High-level goal: permanently go between initial situation and extreme situation, following an optimal path.

  Length of minimal path: 8 step between initial situation and extreme situation.

The reason for using the numbers 10, 20, 30, 40, 50, 60, 70, 100 and 200 is this: In the elementary path-finding domain, described by the object level rules, a robot alone can decide most of the time its next action in one cycle if its N has been set to some value between 20 and 30. The value is not unique and universal because some action can need more thinking to be decided (i.e. move forward is more straightforward than turn right or left). In Any case 50 is always enough to reach the first `do` predicate, while 10 is always insufficient for this purpose. The scale allows a more systematic approach. The values 100 and 200 are included to compare the performance of the pure-reactive and the planner scheme *for big N*.

We talk about optimal and minimal path because the object-level rules are designed for allowing the robot to follow a shortest path (may be more than one, remember that the robot does not walk over diagonal lines) to its destination. Following the `gradient_step` clause, the agent always chooses a cell that is closer to its destination. This is the criteria for optimality within the program. What we call minimal path is the one with the minimum number of steps for going from initial situation to extreme situation.

It is also important to note that the apparently arbitrary routes the robots follow have been calculated to generate interaction. The selected configuration tries to balance the trade-off between interaction caused by sharing one specific (pre-defined) resource (as in normal distributed systems) and interaction caused by accidental coincidence of goals (attempts to share an arbitrary resource in the world). It is difficult to control this trade-off, so we do not claim this configuration is the best in any sense. The default routes the robots follow when there are no obstructions are shown in figure 3.1. Note that `ja` and `jb` use the same routes for going and coming, whereas `jc` and `jb` have circular routes.

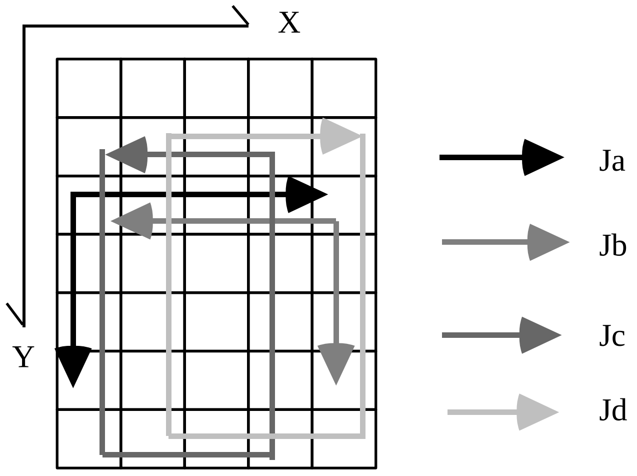


Figure 3.1 Routes of agents.

3.4. Results.

3.4.1. Performance of the agents.

For easing the burden of analising the data from the experiments, we compile the information into statistics that represents the features of the groups of agent we wanted to measure in the simulations. The first of those statistics is the Average time to make a Trip (AVT): an estimation of the time an agent would take to make a trip. A lower AVT indicates that the agent is more efficient in achieving its goal or, in other word, that it performs better. Observe that here we use time instead of, for example, the inference step counter, as the measure of performance per trip. We discard the inference step counter because it is not a uniform unity. From one step of inference to another, the number of clauses and term/clauses involved may change and therefore, we end having more than one independent variable. That is not the case with absolute time. AVT is defined as follow:

$$AVT = \frac{\sum_{j}^{R} TTj}{R} \qquad TTj = \frac{TIME}{TRIPSj}$$

Where R is the number of robot agents in the simulation, TIME is the total duration of the simulation and TRIPS*j* are the number of trips made by agent *j* during the whole simulation.

Figure 3.2 shows a plot of the calculated AVT versus the parameter N for both planner and pure-reactive brains (left and right, respectively).

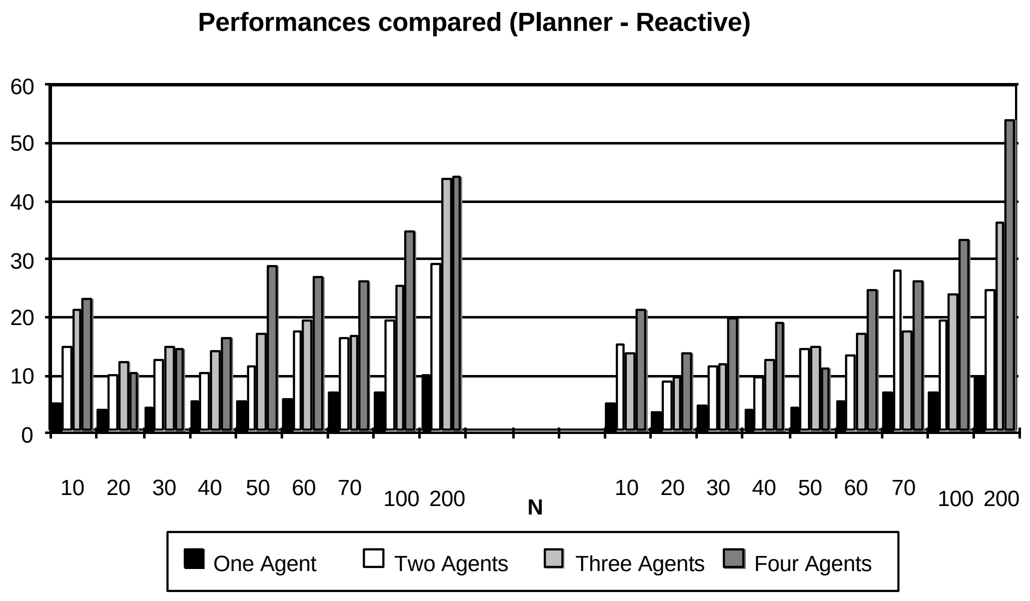


Figure 3.2

Reading the groups of columns separately, one can see that the efficiency consistently drops back as the number of agents increases. Clearly with more agents in the environment, the world becomes more unpredictable and therefore, the agents waste more time deciding and trying actions which eventually fail. Considering the whole graphs, one may note that the efficiency start at some intermediate level and then goes to maximal values for N between 20 and 40. After this point, it decreases monotonicly. There exists a point of optimal performance associated with the *depth* of the knowledge base. Recall that for values of N around 30 the agent almost always gets to the leaves of the decision tree in the current knowledge base, in one cycle.

The graphs show that there is a slightly difference in favour of the planner brain, which could be explained by the fact that this brain can execute more inference step per cycle. The reactive brain stops as soon as it gets to first executable goal (do(Action)). This implies wasting some inference steps still allowed by the remainder of parameter N. Whereas, the planner may keep working in those circumstances, going deeper in the SLD-tree. At the end the net effect is that the planner brain decides, in average, sooner than the reactive brain. However, the important point to observe is that planning does not help to much to avoid the fall of efficiency in this context. It can be note that, ignoring random disturbances, the graphs are very close one to another. Generally speaking, one would expect that an agent that can plan up to

10 actions ahead in one cycle, would have a better performance than one that only decides its next action. However, clearly that is not the case with the current mechanism for planning applied to this domain. We can mention two notable weaknesses of the current planner brain, in order to explain that: 1) because there is the possibility of evasive actions, any plan can always be extended although the plan being followed may not be optimal (for instance, when facing a wall the robot turns right and keeps following the wall even though turning left would take it to its destination in fewer steps), and 2) The robot never evaluates the situation in which will be in the future due to the execution of certain plan, which also could be used to select a better plan. In both cases, there is no overall analysis to select the best among several solutions. The planner brain (and also the reactive brain) commits to a solution *built into* the knowledge base for this domain. A solution that may not be the best. It seems that other capabilities, as projection into the future and meta reasoning about planning, are required in pursue of the optimal performance.

On the other hand, the characteristics of the `patio` world with few regular structures to take care of (and from which to learn), cancel out the advantages of looking ahead. The `patio` world is well suited for representing an environment where the agents are responsible for change, but it is not complex enough for showing the advantages of planning. A more *constrained* environment (like the warehouse) where the fixed obstacles influence the choice of the best route, could be more promising in that sense. Of course, there are still other aspects to be take into consideration. A larger range of perception, for instance, can improve the *quality* of knowledge (how well represent the world) by allowing the agent to learn more about the real world in less time. Both types of agent can take advantage of a wider perception but it is the planner which will exploit the *fresh* knowledge to a greater extend.

As a final remark, let us say that absolute efficient seems to depend heavily on the memory management policy used by the `plagent`s. However, in this work we limit ourselves to maintain the same garbage collection scheme through all the tests and to make observations on relative efficiency. We do not pretend to employ this results to evaluate any of the tools in the test-bed (namely SWI-PROLOG and the APRIL system).

3.4.2. Quality of decision-making

The second statistic we have calculated is the Average number of failures per trip per agent (AVF). This indicator could also be regarded as a measure of performance. However, we are getting rid of the time as the reference and attempting to measure efficiency due only to correct decision making. If the world model in the agent's brain is a faithful representation of the real world, the robot will avoid failures on its attempts to act. The AVF is defined below:

$$AVF = \frac{\sum_{j}^{R} AFj}{R} \qquad AFj = \frac{NFj}{NTj}$$

Where R is the number of agents, AF*j* is the average number of failures of agent *j*, NF*j* is the total number of failures of agent *j* and NT*j* is the total number of trips of agent *j*.

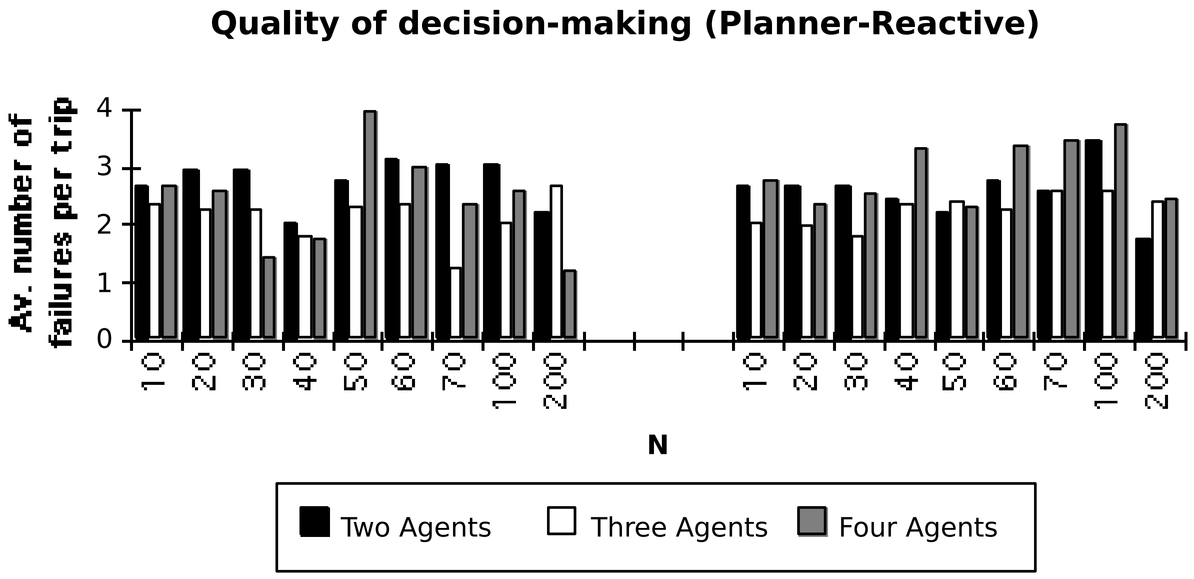


Figure 3.3

Unfortunately, there is no clear tendency in the data, as one can see from both graphs. It seems that either this statistic is inadequate for capturing the information or the experiments are too limited. We still believe the AVF could be used to analysed the quality of decision making. Probably, we need to involved a bigger group of agents to make arise the effect of higher entropy in the system, that suggests more failures with more agents. We already obtain a bigger total number of failures with more agents, but also more trips per agent. In any case, we will not make any conclusion about this topic.

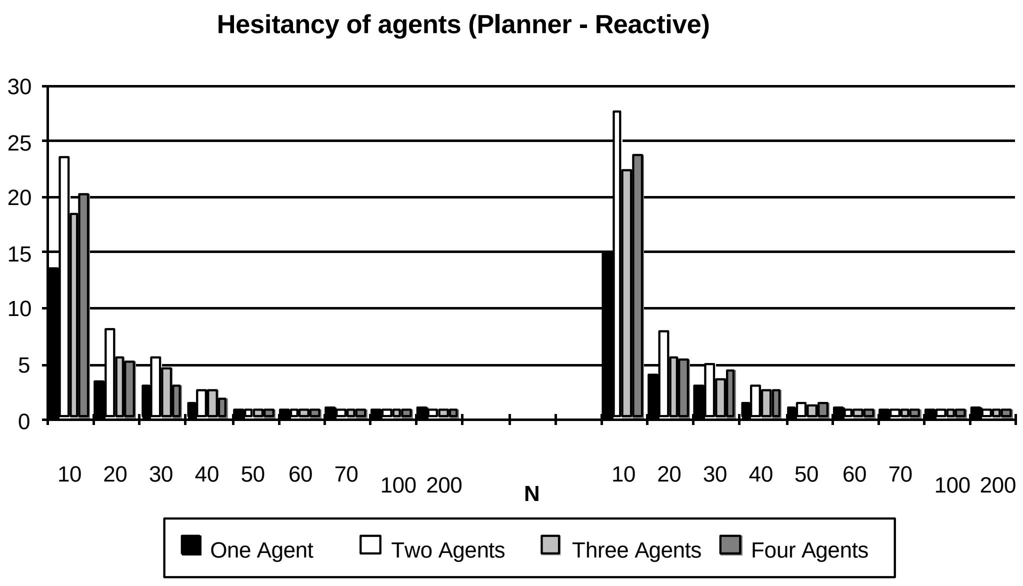
3.4.3. Hesitancy

The final statistic we considered is intended to show the relationship between the hesitancy of the agent and bounded reasoning. What we call hesitancy is the behaviour of a robot that can not decide the next action to be executed. In those situations (cycles) when this happens, the robot activity is limited to an observation of its environment. The robot executes no action to change the world or itself. Of course, this is closely related to the depth up to which the agent is allow to reason. A superficial exploration of its alternatives for acting (namely, its hierarchy of goals), caused by a low N, is likely to end in hesitancy. In other words, hesitancy is what the designer or controller of the brain should expect if he/she does not allocate to the robot enough resources to reasoning.

We try to capture the hesitancy of the agents by measuring the Average number of observations per trip (AVO). It is important to clarify the distinction between the kind of explicit observation we record and the almost continuous stream of observations the agent receives. The locus of control algorithms implies an observation after every action is executed. In our system, every time an action is executed, the world sends back an update of the new range of perception of the agent, due to its new situation. This is what we call an *implicit observation* and it always occurs. Observe that implicit observations are linked to actions that change the world. We talk about *explicit observations* either 1) when the brain does not decide an action to be execute, in which case it order the brain only to look, or 2) when the hierarchy of goals contains the executable goal: `do(look)`, that we use to mark particular points within the plans (in this case at the end of a trip between initial situation and extreme situation). Indeed, the agents are enforced to make an (explicit) observation every time that they get to the end of one trip. Any additional (explicit) observation during the trip can be regarded as hesitancy. We defined AVO as follow:

$$AVO = \frac{\sum_{j}^{R} OTj}{R} \qquad OTj = \frac{NOj}{NTj}$$

Where again, R is the number of agents, OT*j* is the average number of observations per trip in agent *j*, N0*j* is the total number of observations made by agent *j* and NT*j* is the total number of trips by agent *j*. Figure 3.4 shows the graphs of AVO versus N.



Fig

ure 3.4

This time the message is clear. In the current hierarchy of goals, N set to a value greater than 50 is always enough to get to the leaf of any branch and, hence, to make a decision about the next action in one cycle. The other interesting point to note here is that, for very low N, the planner brain dictates a slightly less hesitant profile of activity than the pure-reactive brain, even in a changing environment (several agents). This is probably the only advantage of what we call planning in this context: to keep reducing the goals after the next action has been decided. While the reactive agent may stop with a list of goal similar to (see chapter 2 for a preliminary explanation of this example):

```
[(do(t_right), go(3e, 5e)) ; .. other alternatives.. ]
```

the planner brain might end with:

```
[(do(t_right), atom(A, 3e, Z), closer(3e, Z, 5e),
 not prohibited(Z), do(A), go(Z, 5e)) ;
.. other alternatives.. ]
```

or even with:

```
[(do(t_right), do(m_forward), go(4e, 5e)) ;
.. other alternatives.. ]
```

where the action `t_right` and `m_forward` are decided in one cycle. One of them (`t_right`) will be consumed immediately by the executive. The other (`m_forward`) is going to be used in the next cycle, even if N is too small to get a new executable goal in that cycle.

Thus, the agent can try actions decided in previous cycles when the value of N allows to reach the leaves of the hierarchy of goals. This may be the case when the branches are shorter (let say "rules with few conditions *fire"*, using the condition action rules' analogy).

Currently the value of the parameter N is the same throughout the robot's *life*. In a more sophisticated agent, the parameter N may be let to change during its life in order to reflect the *mood* of the agent (in a hurry, excited, calm, inert). Of course, the practical purpose of a changing N is to allow the agent a more dynamic adjustment to the available resources (time and memory space).

Nevertheless, it is important to note that a less hesitant agent is not necessarily a more efficient one. The time wasted by an agent that could not decide its next action, can be equivalent to the time spent by the same agent testing more conditions and building a plan within one cycle. More reasoning per cycle aimed to build a plan implies more memory consumption and in practice, waste of time in memory management. Consequently, the remarks about planning and efficiency made in section "performance of the agents" are perfectly compatible with the results in this section.

3.4.4. Other results

Apart from the formal, objective results presented so far, other discoveries were made during the tests. First of all, and in spite of our efforts to provide the agent's brain with all the knowledge needed to make optimal decisions (pre-compiled set of rules), there are still situations where groups of path-finder agents get stuck. We did not fix them because that would have implied repeating most of the tests and we lacked the time. Thus, we continued working with the anomalous rules until the end. However this allowed us to confirm that those dangerous situations were unlikely to occur, even thought under more restricted testing conditions (without the randomness of the network) they do arise.

This discovery was due to collateral observations and thus we can not jump into any conclusion yet. Nevertheless, we wonder whether this effect of randomness, which allowed our system to behave well even with incomplete knowledge guiding the agents, could be linked with the *emergent behaviour* reported in other works (see [Steels; 1990] and [Feber, Drogoul; 1991]).

There are, we have to say, problems concerning the `plagent` program (SWI-PROLOG with the PROLOG-APRIL interface). In several occasions during the tests, the PROLOG brains stopped working after a call to the garbage collector. The system works reasonably well most of the time, but we believe the `plagent` still requires more testing and debugging.

CONCLUSIONS AND GUIDELINES FOR FURTHER RESEARCH

Conclusions

This work has included a survey of one of the basic topics that has motivated research in Distributed Artificial Intelligence and Multi-Agents Systems in the last years: the use of logic for modelling autonomous agents. We contribute to the debate about reactive agent and deliberate agent by saying that the former can be regarded as a special case of the later. One and the other can be modelled as logic programs. Taking a proposal by Genesereth and Nilsson [Genesereth, Nilsson; 1988] and that provided by [Kowalski; 1994b] as references, we present an alternative model for a **deliberate agent with bounded reasoning**. The reasoning is bounded because the agent has limited resources (time and space of memory) to perform deductions and to select its next actions. We analyse Steel's proposal [Steels; 1990] which advocated the use of reactive systems and show that we can get similar behaviour and performance, with logic-based deliberate agents. Indeed, we show how to accommodate a domain knowledge base, with goals and rules written as terms and clauses, in such a way that rules of behaviour in specific situations are represented by separated *layers of knowledge*. Thus, an ordinary computation rule can be used to perform a ***search by layers*** that, we believe, is equivalent to what is accomplished by a subsumption architecture [Brooks; 1991]. We also explain how to design programs that implement a ***forward search of solutions while using a backward reasoning proof procedure***.

We address the criticism of Hewitt [Hewitt; 1985] to the use of logic and closed world assumptions in open systems by devising the **adaptable locus of control algorithm**. This locus of control, used as the main control loop in an agent brain, intermixes observation, assimilation (knowledge updates, resolution and planning) and execution. This allows the agent to respond quickly to contingencies in the environment using its recently updated model of the world, but still following its own goals and purposes of life. We believe our artificial *creatures* conform to almost all the requirements set by Brooks [Brooks; 1991] because 1) they can cope opportunistically with changes in the environment; 2) they maintain a set of multiples goals and, depending on the circumstances, can choose which particular goal to

pursue and 3) The creatures have some high level goal for living, something to do in the world. We do not claim to address the fourth requirement of Brooks: *The creature must be robust with respect to its environment* because our project is limited to the simulations of multi-agent environments and thus, we are still far away from concrete implementation of, for example, real robots. Moreover, our model of brain can not cope yet with unexpected changes in the agent situation produced, for example, by accidents or natural events. Nevertheless, we believe that even Brook's fourth requirement can be fulfilled with logic-based representations.

We also highlight some of the limitations of the representation adopted in this work. Our robots can not evaluate their current situation in the world and project themselves into the future or the past. We believe that this in an important feature in autonomous intelligent agent. Furthermore, we suggest that these capabilities can be provided to agents modelled as logic programs.

All the main discussions in this work are supported by experiments carried on logic programming agents. We devised models of agents using APRIL and PROLOG as the development languages. An APRIL-PROLOG test-bed is configured to make experiments tracing the performance and other features of logic programming agents. We simulated a world where agents (robots) interact by performing actions. The very same test-bed is ultimately aimed at supporting simulated contexts where agents communicate and cooperate in pursue of social goals. However, in this project, the test-bed has been employed to evaluate the impact of the bounded reasoning and to compare the performance of pure-reactive brains and models of brains with planning capabilities.

Several results arose from the experiment:

1) The parameter N, which is used to bound the reasoning by limiting the depth of search in a knowledge base, proved to be very influential in the efficiency of the agent. Normally, to obtain the best performance, the value of N should be such that allow the theorem-prover in the agent's brain to get to the leaves of its hierarchy of goals. This allows the agent to decided its next action in one cycle. However, a greater value may decrease the efficiency.
2) We discovered that, apart from the parameter N limiting the depth of search, other parameter seems to be required in agents with planning capabilities. We observed that a planner brain, that permanently tries to refine its plans until its lower level details, pays the cost of a formidable plan description with performance. For the sake of more realistic agents, we need to limit the extension and *level of detail* of plans built by the planner brain. As the low-level details of a plan in this context (specific actions to be executed) depend on the

abduction capabilities of our logic programming brains, we call the new control feature **bounded abducibility**. Once we introduce this concept in our formalism, the pure-reactive brain, a model of brain that concentrates its reasoning in obtaining only the next action to be executed, becomes a special case of a planner brain, where the reasoning may generate not only the next action, but a sequence of actions to be executed (a plan). The former is simply a planner brain whose *abducibility control parameter* has been set to 1. Therefore, our representation allows a smooth transition between reactive agents and planner agents. A logic programming agent can be *tuned*, not only to accommodate the amount of resources needed for reasoning, but also to control plan construction.

3) Finally, the simulations have shown no clear advantage in using a planner brain instead of a pure-reactive brain. We speculate that this may be due to the fact that our simulated contexts are too simplistic and the effect of the bounded reasoning and mainly, the dynamic of the environment, limit the advantages of planning. Perhaps a more complex environment, with less unpredictable dynamic, could show the relevance of building and following plans.

Recalling what was said in the introduction, probably the main by-product of this work is the evidence supporting the fact that a logical representation can be used to model and implement intelligent autonomous agent. The exciting point is that we already solve some of the main problems in designing Multi-Agent System and we have not exploited but a very limited set of the representations and programming strategies available within Logic Programming.

Extensions and further research

The immediate way for extending this work is by completing the warehouse example. It is not difficult to modify the robots' bodies to include the capabilities for taking, carrying and leaving boxes. Moreover, it does not require much programming effort to extend the object-level goals' hierarchy in the robot's brain in order to give account of goals such as: `find_truck`, `find_boxes`, `pick_a_box`, `carry_box_to_shelves`, `find_box_place`, `leave_box`. We would expect that the changes in the world agent and database, needed to incorporate (and to visualise) those more complex robots' activities, would require a careful design and may require more time. In addition a systematic set of experiment should be carried out in the test-bed, in order to ensure that the agents can cope with the interactions and that deadlock situations are avoided. Of course, first, the incompleteness of the current knowledge base (due to insufficient rules of behaviour and optimality criteria) already reported in this document, should be attacked. Also, some changes to the range of perception of the robots could be attempted by, for instance, allowing the agent *to see* across a larger area, but with obstacles hiding the part of the picture behind them.

Once those changes have been made and tested, the next stage of this research project may be the addition of communication and cooperation capabilities to the logic-based agents. A large amount of effort has been given to the issues of cooperation and communication within Distributed Artificial Intelligence (see [Bond, Gasser; 1988]). Thus, we would expect this part to be more time-consuming and to have many concepts and approaches to take into consideration. One has to bear in mind the motivation for including communication and cooperation in a Multi-Agent System: **coordination**. Coordination has been cited as possibly the *key research problem in DAI* [Gasser; 1992]. Once one decides, as we have done in this work, to build agents that perform autonomously in the world, using partial knowledge of the environment to make decisions (*the devils of partiality and autonomy* as pointed out by Jennings [Jennings; 1994]), one has to provide the way for *programming* groups of agents for the efficient pursuing of relevant goals (that is, to make all of them do something useful in a constrained environment). In [Gasser; 1992] Les Gasser also discusses a classification of approaches to coordination based on the predictability and reactiveness of the mechanisms employed: *organisation, exchange of meta-level information, multi-agent planning and explicit analysis with overall synchronisation*.

We believe that most, if not all, of those kinds of mechanisms for coordination can be embedded into logic-based agents. This would allow a dynamic self-adjustment of an agent to its *social environment* according to its *historic situation*. However, this discussion may be overwhelmingly complex. Therefore, we suggest a step-by-step approach, starting by employing only one coordination mechanism. *Meta-level information exchange* that implies agents sending each other control level information about their current priorities and focuses [Gasser; 1992], seems to be a promising approach for reasons that will be elaborated upon later. Besides, other aspects could also be included in a logic-based approach to coordination. The agent's reasons for cooperating (*benevolence, altruism and self-interest* as were classified in [Connah, Wavish; 1990]) may prove to be adjustable in the logical design.

In any case, to tackle the problem of cooperation requires a formal approach. In a preliminary exploration, we have found an interesting framework in the work by Jennings [Jennings; 1994] that establishes theoretical conditions for cooperation. Jennings distinguishes between *identical and parallel goals* and between *accidental coordination and cooperation*. If, for example, two path-finder robots want to go to the same place we say they have identical goals. However, if both of them have the goal "be at cell 1 at time t", they have parallel goals. The former kind of situation is suitable for cooperation because the agents may decide to work as a team and, for example, share information about routes to that place. The second kind may generate conflict because both robots will compete for getting the resource, in this case the cell. On the other hand, accidental coordination occurs when, for instance, two or more path-finder pick different routes toward their destinations and, therefore, help each other (by leaving their ways clear) without previous agreement. *That is not cooperation* [Jennings; 1994].

An important point to be noted in Jennings' project is its attachment to implementation. The theoretical rules support a DAI *shell* (GRATE [Jennings; 1994]) which has been used to integrate pre-existing knowledge-based system as agents in industrial distributed applications. The *often-neglected* use of DAI technology to integrate pre-existing standalone systems [Jennings, Wittig; 1992] entails an attitude that, we believe, may be useful to research: it needs to tackle immediately the implementation of models and designs, which may help to highlight relevant trade-offs, assumptions and other details. In addition, integrating pre-existing systems often entails filling, with artificial agents, the gap normally occupied by human operators. Building systems that work in human being's roles and cooperate with people is an initiative that has already reported success (see for instance, [Maes; 1994]). Summarising this second point, we suggest adopting two methodological guidelines for the next stages of research: 1) commitment to implementation and testing of programs as realistic as possible and 2) human beings' behaviours as *desiderata* of flexibility and practical capabilities. Our work so far has been guided by those ideas.

Although the previous paragraphs delineate a rather wide set of steps to continue this work, we believe that an even wider group of ideas needs to be considered. As we said in the conclusions, this work only involves a sub-set of concepts and tools of Logic Programming: meta-interpreters, search by layer, forward-backward representations, *decompilation* (restoration of goal-orientedness) of condition action rules and modelling of rules of behaviour as logic programming clauses. Logic Programming, within the semantic framework of Computational Logic 'CL' [Kowalski; 1994a], still has a large set of features and tools to be applied. Two of them are likely to be introduced in the next stages of this research in logic-based agents: 1) *Event Calculus* [Kowalski, Sergot; 1986] to cope with efficient temporal representation that would provide the agents the desired projection into the past and the future, while avoiding the *frame problem* , and 2) *Metalogic*.

The prefix *meta* is usually employed in a broad diversity of interpretations, some of them questionable. Thus, first attaching ourselves to the definitions of metalogic investigated by Kowalski ([Kowalski; 1979], [Kowalski; 1994a], [Kowalski; 1994c][3]) we claim that *meta-representation* may play an important role by allowing agents to model and to reason about other agents. Agents in Multi-Agents

3In that reference Kowalski defines metalogic as a set of *theories that "talk" about sentences, theories and other linguistics objects.*

systems normally keep models of other agents (or *acquaintances*, see [Jennings, Wittig; 1992] and [Gasser, Braganza, Herman; 1988]). Those models of others are used to [Jennings; 1994] 1) focus activities of the group and reducing communication overhead, 2) guide the transmission of predictive information and 3) enable self-reflection. The first two of these options conforms to the so-called meta-level information exchange strategy for cooperation. What we suggest is that metalogic may be employed by logic-based agents to guide all those three internal activities. In addition, metalogic might provide a solution to the *ramification problem* in planning and even to the *omniscience problem* of conventional modal representations [Kowalski; 1994a]. That is why we proposed to follow the mentioned mechanism for cooperation: for exploring the implementation issues of metalogic that already has been used to solve problems of highly complex, rational interactions as *the wise man puzzle* [Kowalski; 1994c].

REFERENCES

[Brooks; 1986] Rodney Brooks. "A robust layered control system for a mobile robot". *IEEE Journal of Robotics and Automation,* 2: pg. 14-23, 1986.

[Brooks; 1991] Rodney Brooks. "Intelligence without representation". *Artificial Intelligence*. 1991. Vol 47. pg. 139-159.

[Bond, Gasser; 1988] Alan H. Bond and Les Gasser (Ed). *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann Publishers. San Mateo. Ca. USA. 1988.

[Connah, Wavish; 1990] David Connah and Peter Wavish. "An experiment in cooperation". In *Decentralized A. I.; Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World*. Cambridge, England. August 1989. Y. Demazeau and J.-P. Müller (Eds.). North-Holland. 1990.

[Covington; 1994] Michael Covington. *Natural Language Processing for Prolog Programmers*. Prentice Hall. 1994.

[Ferber, Drogoul; 1991] J. Feber and A. Drogoul. "Using Reactive Multi-Agent Systems in Simulation and Problem Solving". *Distributed Artificial Intelligence: Theory and Praxis*. Avouris and Les Gassers Eds. Kluwer Academic Publishers. The Netherlands. 1991

[Gasser; 1992] Les Gasser. "DAI Approaches to coordination". *In Distributed Artificial Intelligence: Theory and Praxis*. N. Avouris and Les Gasser Eds. Kluwer Academic Pub. 1992. pg. 31-51.

[Gasser, Braganza, Herman; 1988] Les Gasser, Carl Braganza and Nava Herman. "MACE: A Flexible Testbed for Distributed AI Research". In *Distributed Artificial Intelligence*. M. N. Huhns Ed. Pitman Publishing. pg 119-153.

[Genesereth, Nilsson; 1988] Michael R. Genesereth and Nils J. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kauffman Publishers. CA. 1988.

[Hewitt; 1985] Carl Hewitt. "The Challenge of Open Systems". *BYTE*. Vol 10. April 1985. pg.123.

[Hogger; 1990] Christopher Hogger. *Essentials of Logic Programming*. Oxford University Press, 1990.

[Jennings; 1994] Nick Jennings. *Cooperation in Industrial Multi-Agent Systems*. World Scientific Pub. 1994.

[Jennings, Wittig; 1992] N. Jennings and T. Wittig. "ARCHON: Theory and Practice". In *Distributed Artificial Intelligence: Theory and Praxis*. N. Avouris and Les Gasser Eds. Kluwer Academic Pub. 1992. pg. 179-195.

[Kowalski; 1979] Robert Kowalski. *Logic for Problem Solving*. North-Holland. Elsevier Science Publishing Co. Inc. 1979.

[Kowalski; 1994a] Robert Kowalski. "Logic Without Model Theory". In *What is a Logical System*. Oxford University Press. D. Gabbay Ed. 1994. To appear

[Kowalski; 1994b] Robert Kowalski. "Reconciling reactive with rational agents". Internal Report. Imperial College. July. 1994.

[Kowalski; 1994c] Robert Kowalski. "Why metalogic?". In Knowledge Representation: non-monotonic reasoning and metareasoning. Foundations for Advanced Information Technology. Course's Notes. Dept. of Computing. Imperial College. London. March. 1994.

[Kowalski, Sergot; 1986] Robert Kowalski and Marek Sergot. "A Logic-based Calculus of Events". *New Generation Computing* 4 (1986). pg. 67-95. OHMSHA, LTD and Springer-Verlag.

[Maes; 1994] Pattie Maes. "Agents that reduce work and information overload". In *Communications of the ACM*. Volume 37. No. 7. July 1994. pg 31-40.

[McCabe;1993] F.G. McCabe. *April - agent process interaction language*. Internal report, Dept. of Computing. Imperial College, London. 1993.

[McCabe;1994] F.G. McCabe. "Object Oriented Programming with April". In Distributed Artificial Intelligence. Foundations for Advanced Information Technology. Course's Notes. Dept. of Computing. Imperial College. London. January. 1994.

[Muller, Pischel; 1993] Jörg P. Muller and Markus Pischel. *The Agent Architecture InteRRaP: Concept and Application*. German Research Center for Artificial Intelligence (DFKI). Report. 1993.

[Newell, Young, Polk; 1993] Allen Newell, Richard Young and Thad Polk. "The Approach Through Symbols". In *The Simulation of the Human Intelligence*. D. Broadbent Ed.; Blackwell Pub.; 1993.

[Shoham; 1991]. Yoav Shoham. "AGENT0: A simple agent language and its interpreter". In *Proceedings Ninth National Conference on Artificial Intelligence. AAAI-91*. July 14-19. 1991. Vol II. pg. 704-709.

[Shoham, 1994] Yoav Shoham. *Artificial Intelligence Techniques in Prolog.* Morgan Kaufmann Publishers. Ca. 1994.

[Steels; 1990] Luc Steels. "Cooperation between distributed agents through self-organisation". In *Decentralized A.I.; Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World*. Cambridge, England. August 1989. Y. Demazeau and J.-P. Müller (Eds.). North-Holland. 1990.

[Wielemaker; 1989] Jan Wielemaker. *SWI-PROLOG 1.6. Reference Manual*. Dept. of Social Science Informatics (SWI). Amsterdam. The Netherlands. 1989.

APPENDIX A. AGENTS IN APRIL

This appendix shows the source-codes of:

- **`robot.ap`**. The APRIL program that models the forklift/path-finder agent. Within this code the reader will find the specification of the processes that simulate this kind of agent.

- **`patio.ap`**. The APRIL program that represents the patio during the simulations.

- **`dk_world.ap`**. This code should replace the `Worldprocess` procedure (see the `patio.ap` code) when one wants to perform animated simulations of the warehouse.

- **`common.ah`**. This is header file, used for declaring some functions and macros employed by the APRIL programs.

- **`world.db.patio`**. It shows the data stored in the file `world.db` every time that a simulation of the patio world is carried out.

- **`world.db.warehouse`**. This is the data describing the warehouse for the APRIL program. Observe that simply by using the data in this file as the initial data in the file `world.db`, the system will simulate the warehouse. No recompilation of the APRIL code is required. Unfortunately, to display the warehouse in the DIALOX animation still (in this version) requires changes in the code and recompiling.

- **`exam1.ap`**. This files contains the example used in the section I.2 of the Introduction to explain the syntax of APRIL. The example include the tracing call to the routine in `common.ah`.

```
/* robot.ap

 A robot in the patio (or in the warehouse) is simulated by
 an agent whose structure and behaviour is modeled by this program.

  Last modified: 30 - 7 - 94.

  Knowledge assimilation in multiagents system.

  Jacinto Davila. Imperial College.

*/

#include "common.ah";

/* _______________________________________________data structures */
/*
  world_model ::= [[posx, posx, object, type, status, bitmap]]
  arms_towards E { north, south, west, east }
  mental_context ::= [ posx, posy, arms_toward, robot_status,
                    current_world_model ]
*/
listofsteps ::= [action] ;
act_list ::= [action,listofsteps] ;
world_item ::= [coord, coord, symbol, symbol, symbol, symbol] ;
world_model ::= [world_item] ;
mental_context ::= [ coord, coord, direccion, rb_status, world_model ] ;

/* ______________________________________________________newdir */
newdir(symbol?dir,action?ac) -> symbol?valof{
 { ac = m_forward || ac = look => valis dir
 | ac = t_back =>
   { dir = north => valis south
   | dir = south => valis north
   | dir = west => valis east
   | dir = east => valis west
   }
 | ac = t_right =>
   { dir = north => valis east
   | dir = south => valis west
   | dir = west => valis north
   | dir = east => valis south
   }
 | ac = t_left =>
   { dir = north => valis west
   | dir = south => valis east
   | dir = west => valis south
   | dir = east => valis north
   }
 }
};

/* ______________________________________________________________ body */
```

```
body(handle?name,handle?worldname){

trace("BODY","activated..",[]);
```

```
 repeat {
   [execute,symbol?nact,
      [integer?px,integer?py,symbol?dir,symbol?st,any?wm]] => {
      /* send execution intension to the world */
      [act, nact, px, py, dir] >> worldname ;
               /* Predefined world agent */
      /* Answer from the world */
      {  [ok, integer?npx, integer?npy, world_model?new_scene] => {
          /* The action succeed.
                          Now the mc should be update accordingly */
          symbol?ndir := newdir(dir,nact) ;
          [ok,[npx, npy, ndir, st, new_scene]] >> name
        }
      |  [failed,action?failed_act,integer?npx,integer?npy,
                  symbol?Look] => {
          [failed,failed_act] >> name
        }
      }
    }

 |  any?Msg => trace("BODY","strange message %p",[Msg])

 } until die ;
};


/* __________________________________________________process_plan */
process_plan(listofsteps?LowlevelPlan) -> act_list?valof{
  if (|LowlevelPlan| == 0) then
    valis act_list?[nothing,LowLevelPlan]
  else {
    symbol?Item := symbol?LowlevelPlan[0] ;
    integer?rest := |LowlevelPlan|-1;
    /* symbol[]?t := back(LowlevelPlan,rest); */
    /* [symbol?Item] = h ; */
    valis act_list?[Item,LowlevelPlan]
  }
};


/* __________________________________________________________robot */
robot(handle?name, handle?worldname, integer?PX,
              integer?PY, symbol?Dir){
 /* handle?mybrain := fork(brain(name)); April Brain lobotomized */
 handle?mybrain := handle?catenate(brainof_,symbol?name);
 handle?mybody := fork(body(name,worldname));
 mental_context?mc := mental_context?[] ;

 [act,birth,PX,PY,Dir] >> worldname ;
 /* Registering with the world */

 [ok, integer?X,integer?Y, world_model?firstsight ] => {
   wake_up >> mybrain ;           /* Registering with its brain */
   mc := [X,Y,Dir,free,firstsight] ;
 };

 listofsteps?current_plan := listofsteps?[] ;
 action?next_action := nothing;
 number?counter := 1;
```

```
 while true do {

trace("ROBOT","Assimilation-Execution Cycle number %d\n",[counter]);

   [observe,mc] >> mybrain;                /* Initializing cycle */
   while true do {
     [any,listofsteps?Plan] => { /* never receives an empty plan */
       next_action := action?head(Plan) ;
       current_plan := listofsteps?extract(Plan);
       [execute, next_action, mc] >> mybody ;
       while true do {
         [ok,any[]?Nmc] => {
            mc := mental_context?Nmc; /* Updating mental context */
            succeed >> mybrain ;             /* Action succeed   */
            go => relax ;    /* Sincronizing. Should'n be needed */
            break;
         }
       | [failed, action?fail_action] => {
            fail >> mybrain ;                /* Action failed    */
            go => relax ;    /* Sincronizing. Should'n be needed */
            break;
         }
       | any?Msg => trace("ROBOT",
                        "Wrong msg %p while waiting for ok or failed\n",[Msg])
       };
       break;
     }
   | any?Msg => trace("ROBOT",
                         "Wrong msg %p receive while waiting for plan",[Msg])
   };
   counter +:= 1;
 };
 die >> [mybrain, mybody] ;
};


/* ******************************************************************** Main */
main(any[]?ar){
 if |ar| < 5 then {

   trace("ROBOT","usage: april robot
             <robotname> <worldname> <coordX> <coordY> <dir>\n",[]);

   exit(1) ; /* names of agent and world should be provided */
 } ;
 handle?ar[1] names robot(handle?ar[1],
                        handle?ar[2],
                       integer?ar[3],integer?ar[4],symbol?ar[5]) ;
 handle?rb := handle?ar[1] ;
 while true do {
    any?Msg => Msg >>> rb ; /* Any message goes to robot */
 }
```

```
};
```

```
/*  patio.ap


   This file contains the code of two process.The "world" process which represents the environment in
simulation, and a subsidiary process called WorldProcess which triggers and controls the visualisation with
Dialox.

  The file also contains the especification of the data structures used for the "world". It worth noticing that the
world database is simply a description of the cells which conform the environment. Each cells has attributes
attached to it, to indicate which "things" it is. For instance, the "free" cells througth which the robots can walk,
are "floor_" cells. Each robot, in turn, occupies a cells which is then regarded as "full". (see file world.db)

  Version: 1.0   Robots` range of perception: the front cell

  Last modified: 30 - 7 - 94

  Jacinto Davila. FAIT 94. Imperial College.
*/

#include "common.ah";
/* ______________________________________________________ macros */
#macro maxx -> 500;
#macro maxy -> 700;
#macro sqmax -> 100; /* square size */
/* _______________________________________________________bitmaps */
#macro rob_east(?N,?X,?Y) -> [group,
                                        [text, X+sqmax/4, Y+sqmax/2, N ],
                       [line,X,Y,X,Y+sqmax],
                       [line,X,Y+sqmax,X+sqmax,Y+sqmax],
                       [line,X,Y,X+sqmax,Y],
                       [line,X+sqmax,Y,X+sqmax,Y+sqmax],
                       [ellipse,X+sqmax/4,Y+sqmax/2,40,40],
                                        [line,X+3*sqmax/4,Y+sqmax/4,
                                                       X+sqmax,Y+sqmax/2],
                                        [line,X+sqmax,Y+sqmax/2,
                                                 X+3*sqmax/4,Y+3*(sqmax/4)]];

#macro rob_west(?N,?X,?Y) -> [group,
                                        [text, X+3*sqmax/4, Y+sqmax/2, N ],
                       [line,X,Y,X,Y+sqmax],
                       [line,X,Y+sqmax,X+sqmax,Y+sqmax],
                       [line,X,Y,X+sqmax,Y],
                       [line,X+sqmax,Y,X+sqmax,Y+sqmax],
                       [ellipse,X+3*sqmax/4,Y+sqmax/2,40,40],
                       [line,X+sqmax/4,Y+sqmax/4,X,Y+sqmax/2],
                                        [line,X,Y+sqmax/2,X+sqmax/4,
                                                       Y+3*(sqmax/4)]] ;

#macro rob_south(?N,?X,?Y) -> [group,
                    [text, X+sqmax/2, Y+sqmax/4, N ],
[line,X,Y,X,Y+sqmax],
[line,X,Y+sqmax,X+sqmax,Y+sqmax],
[line,X,Y,X+sqmax,Y],
                        [line,X+sqmax,Y,X+sqmax,Y+sqmax],
                        [ellipse,X+sqmax/2,Y+sqmax/4,40,40],
                        [line,X+sqmax/4,Y+3*sqmax/4,
                                                    X+sqmax/2,Y+sqmax],
                        [line,X+sqmax/2,Y+sqmax,X+3*sqmax/4,
                                                    Y+3*(sqmax/4)]
                   ] ;

#macro rob_north(?N,?X,?Y) -> [group,
                                             [text, X+sqmax/2, Y+3*sqmax/4, N],
                         [line,X,Y,X,Y+sqmax],
                         [line,X,Y+sqmax,X+sqmax,Y+sqmax],
                         [line,X,Y,X+sqmax,Y],
                         [line,X+sqmax,Y,X+sqmax,Y+sqmax],
                         [ellipse,X+sqmax/2,
                                                          Y+3*sqmax/4,40,40],
                         [line,X+sqmax/4,Y+sqmax/4,
                                                          X+sqmax/2,Y],
                         [line,X+sqmax/2,Y,X+3*sqmax/4,
                                                          Y+sqmax/4]] ;

/* _______________________________________________ data structures */
/* Data structures:
  world_item ::= [posx,posy,object,type,status,bitmap]
  world_model ::= [world_model_item]
  arms_towards E { north, south, west, east }
  mental_context ::= [ posx, posy, arms_toward
                  current_goal,
                  current_plan,
                  current_world_model ]
  acquantaince_model ::= [ name,
                     posx, posy, arms_toward,
                     his_current_goal,
                     his_current_plan ]
*/

action ::= symbol ;
listofsteps ::= [action] ;
goal ::= symbol ;
coord ::= integer ;
direccion ::= north | south | west | east ;
world_item ::= [coord, coord, symbol, symbol, symbol, symbol] ;
world_model ::= [world_item] ;
mental_context ::= [ coord, coord, direccion,
                goal,
                symbol,
                world_model ] ;
acquantaince_model ::= [ handle,
                   coord, coord, direccion,
                   goal,
                   symbol ] ;
```

```
sensing_scope ::= [[integer,integer]] ;


/* ************************************************************/
main(any[]?arg){

 handle?dialox := handle?DialoX;
 handle?wd := handle?robot_world ;

 if |arg| > 0 then wd := handle?arg[1] ;
 if |arg| > 1 then dialox := handle?arg[2] ;

 handle?video := fork WorldProcess(me, dialox) ;

 wd names world(wd, video) ;

 repeat {
   any?Msg => { Msg >>> wd }
   /* Any message goes to world controler */
 } until end ;
 end >> wd ;
};

/* ____________________________________________________________ */

 /* Some auxiliary functions and procedures */
turn_back(symbol?bm) -> symbol?valof{
      { bm = rb_box_north => valis rb_box_south
      | bm = rb_box_south => valis rb_box_north
      | bm = rb_box_west => valis rb_box_east
      | bm = rb_box_east => valis rb_box_west
      | bm = rb_north => valis rb_south
      | bm = rb_south => valis rb_north
      | bm = rb_west => valis rb_east
      | bm = rb_east => valis rb_west
      } ;
};

turn_right(symbol?bm) -> symbol?valof{
      { bm = rb_box_north => valis rb_box_east
      | bm = rb_box_south => valis rb_box_west
      | bm = rb_box_west => valis rb_box_north
      | bm = rb_box_east => valis rb_box_south
      | bm = rb_north => valis rb_east
      | bm = rb_south => valis rb_west
      | bm = rb_west => valis rb_north
      | bm = rb_east => valis rb_south
      } ;
};

turn_left(symbol?bm) -> symbol?valof{
      { bm = rb_box_north => valis rb_box_west
      | bm = rb_box_south => valis rb_box_east
      | bm = rb_box_west => valis rb_box_south
      | bm = rb_box_east => valis rb_box_north
      | bm = rb_north => valis rb_west
      | bm = rb_south => valis rb_east
      | bm = rb_west => valis rb_south
      | bm = rb_east => valis rb_north
```

```
    } ;
};

/* ________________________________________________vision_field */
/* This function build the scene the robot can see from its new
  position. In the current version (1.0) is just the next cell/
vision_field(world_model?RW,
  integer?X,integer?Y,symbol?bm) -> world_model?valof{
  integer?Fposx = 0;
  integer?Fposy = 0;

  { bm = rb_north || bm = rb_box_north => {
      Fposy := Y - 1;
      Fposx := X ;
    }
  | bm = rb_south || bm = rb_box_south => {
      Fposy := Y + 1;
      Fposx := X ;
    }
  | bm = rb_west || bm = rb_box_west => {
      Fposy := Y ;
      Fposx := X - 1;
    }
  | bm = rb_east || bm = rb_box_east => {
      Fposy := Y ;
      Fposx := X + 1;
    }
  };
/*
trace("VISUAL FIELD"," World Model %w \n\n",[RW]);
*/
 world_model?landscape := world_model?(RW^/[Fposx,Fposy,
                                                    symbol,symbol,symbol,symbol]) ;

trace("VISUAL FIELD"," Sight %p \n",[landscape]);

 valis landscape ;

} ;


next_position( coord?Cposx, coord?Cposy, symbol?Looking_To ) -> [coord,coord]?valof{
 coord?Nposx := 0;
 coord?Nposy := 0;
 symbol?Dir := nothing;

 Dir := pname(Looking_To);

 { north = Dir => {
    Nposy := Cposy - 1;
    Nposx := Cposx ;
   }
 | south = Dir => {
    Nposy := Cposy + 1;
    Nposx := Cposx ;
   }
 | west = Dir => {
    Nposy := Cposy ;
    Nposx := Cposx - 1;
   }
 | east = Dir => {
    Nposy := Cposy ;
    Nposx := Cposx + 1;
   }
  };
  valis [Nposx,Nposy] ;
 };


 initial_world() -> any[]?valof{
  symbol?fullname = ffilename("world.db.patio");
  file?f := fopen(fullname,"r") ;

  world_model?wm := world_model?[];

  logical?keep := true ;
  while keep do {
   any?wr := read(f,1000);
   writef(stdout," Reading record %w\n",[wr]);
   if eof(f) then keep := false
   else wm := world_model?(wm \/[wr]) ;
  };

  valis wm
 } ;

 /* ______________________________________________ moving_forward */
 moving_forward(world_model?real_world,
              integer?Cposx,integer?Cposy,symbol?Looking_To,
              handle?rt,handle?interface) -> world_model?valof{
  any[]?NextPos := next_position(Cposx,Cposy,Looking_To);
  [coord?Nposx,coord?Nposy] = [coord,coord]?NextPos;

  any[]?to_ := real_world^/[Nposx,Nposy,floor_,floor_,empty,any];
  if ( to_ = [] ) then {
    [failed, m_forward, Cposx, Cposy, Looking_To ] >> rt
  } else {
    real_world := world_model?(real_world^\[Nposx,Nposy,
                                                     floor_,floor_,empty,any]);
    any[]?from_ := real_world^/[Cposx,Cposy,any,any,any,any];

    /* Catching the attributes values */
    [[any,any,symbol?Ob,symbol?Tp,symbol?St,symbol?bm]] = from_ ;
    real_world := world_model?(real_world^\[Cposx,
                                                     Cposy,any,any,any,any]);
    real_world := real_world \/ [[Nposx, Nposy, Ob, Tp, St, bm]];
    real_world := real_world \/ [[Cposx, Cposy,
                                            floor_, floor_, empty, floor_]];

    /* Trigger the animation */
    [move,Ob,Cposx,Cposy,Nposx,Nposy] >> interface ;

    /* Build the vision field of this robot */
    any[]?new_scene := vision_field(real_world,
                                                           Nposx,Nposy,symbol?bm);
    /* acknowledge execution */
    [ok, Nposx, Nposy, new_scene] >> rt ;
  };
  valis real_world
 };
```

```
/* _________________________________________ _____________ turning_right */
turning_right(world_model?real_world,integer?Cposx,
            integer?Cposy,symbol?Looking_To,
            handle?rt,handle?interface) -> world_model?valof{

 world_model?from_ := world_model?(real_world^/[Cposx,
                                                Cposy,symbol,symbol,symbol,symbol]);
 [[integer,integer,symbol?Ob,
                                    symbol?Tp,symbol?St,symbol?bm]]= from_;
 real_world := world_model?(real_world^\[Cposx,
                                                             Cposy, any, any, any, any]);
 symbol?Nbm := turn_right(bm) ;
 real_world := real_world \/ [[Cposx, Cposy, Ob, Tp, St, Nbm]];

 /* Triggering animation */
 [clear,Ob,Cposx,Cposy,floor_] >> interface ;
 [draw,Ob,Cposx,Cposy,Nbm] >> interface ;
 /* ok => relax ; DEBUGGING */
 /* Build the vision field of this robot */
 any[]?new_scene := vision_field(real_world,
                                                            Cposx,Cposy,symbol?Nbm);
 /* acknowledge execution */
 [ok, Cposx, Cposy, new_scene] >> rt ;
 valis real_world

};

/* _________________________________________________________turning_left */
turning_left(world_model?real_world, integer?Cposx,integer?Cposy,
            symbol?Looking_To,
            handle?rt,handle?interface)-> world_model?valof{
 any[]?from_ := real_world^/[Cposx,Cposy,any,any,any,any];
 [[any,any,symbol?Ob,symbol?Tp,symbol?St,symbol?bm]] = from_ ;
 real_world := world_model?(real_world^\[Cposx,
                                                Cposy, any, any, any, any]);
 symbol?Nbm := turn_left(bm) ;
 real_world := real_world \/ [[Cposx, Cposy, Ob, Tp, St, Nbm]];
 /* Triggering animation */
 [clear,Ob,Cposx,Cposy,floor_] >> interface ;
 [draw,Ob,Cposx,Cposy,Nbm] >> interface ;
 /* ok => relax ; */
 /* Build the vision field of this robot */
 any[]?new_scene := vision_field(real_world,
                                                            Cposx,Cposy,symbol?Nbm);
 /* acknowledge execution */
 [ok, Cposx, Cposy, new_scene] >> rt ;
 valis real_world
};

/* ___________________________________________________________________ turning_back */
turning_back(world_model?real_world,   integer?Cposx,integer?Cposy,symbol?Looking_To,handle?rt,handle?
interface) -> world_model?valof{
```

```
any[]?from_ := real_world^/[Cposx,Cposy,any,any,any,any];
[[any,any,symbol?Ob,symbol?Tp,symbol?St,symbol?bm]] = from_ ;
real_world := world_model?(real_world^\[Cposx, Cposy, any, any, any, any]);
```

```
 symbol?Nbm := turn_back(bm) ;
 real_world := real_world \/ [[Cposx, Cposy, Ob, Tp, St, Nbm]];

 /* Triggering animation */
 [clear,Ob,Cposx,Cposy,floor_] >> interface ;

 [draw,Ob,Cposx,Cposy,Nbm] >> interface ;
 /* ok => relax ; */

 /* Build the vision field of this robot */
 any[]?new_scene := vision_field(real_world,
                                        Cposx,Cposy,symbol?Nbm);

 /* acknowledge execution */
 [ok, Cposx, Cposy, new_scene] >> rt ;

 valis real_world

};


/* ______________________________________________ being_born */
being_born(world_model?real_world,      integer?Cposx,integer?Cposy,symbol?Looking_To,
       handle?rt,handle?interface) -> world_model?valof{

 symbol?Nbm := catenate(rb_,Looking_To) ;

 any[]?pla := real_world^/[Cposx,Cposy,floor_,floor_,empty,any];

 if ( pla == [] ) then {
   [failed, birth, Cposx, Cposy, Looking_To ] >> rt;
 } else {
   real_world := world_model?(real_world^\[Cposx,Cposy,any?Ob,
                                        any?Tp,any?St,any?bm]);
   real_world := real_world \/ [[Cposx, Cposy,
                                         rt, struct, free, Nbm]];
   /* Trigger the display */
   [birth,rt,Cposx,Cposy,Nbm] >> interface ;

   /* Build the vision field of this robot */
   world_model?new_scene := vision_field(real_world,
                                        Cposx,Cposy,symbol?Nbm) ;

   /* acknowledge execution */
   [ok, Cposx, Cposy, new_scene] >> rt ;
 } ;
 valis real_world
};


/* ________________________________________________________ world */
/* This process control the "world" or environment in the Warehouse.
  From a logical standpoint, its main function is to manage the
  "world database", which describe the world as is at each instant.
  (Snapshot database).

*/
```

```
world(handle?name,handle?interface){
 world_model?real_world := world_model?initial_world();
 repeat {
   [act, m_forward, integer?Cposx,
                                          integer?Cposy, symbol?Looking_To ] => {
      real_world := moving_forward(real_world,
                                          Cposx,Cposy,Looking_To,replyto,interface) ;
   }
 | [act, t_back, integer?Cposx,
                                          integer?Cposy, symbol?Looking_To  ] => {
      real_world := turning_back(real_world,
                                          Cposx,Cposy,Looking_To,replyto,interface);
   }
 | [act, t_right, integer?Cposx,
                                           integer?Cposy, symbol?Looking_To  ] => {
      real_world := turning_right(real_world,
                                          Cposx,Cposy,Looking_To,replyto,interface);
   }
 | [act, t_left, integer?Cposx,
                                           integer?Cposy, symbol?Looking_To  ] => {
      real_world := turning_left(real_world,Cposx,
                                                     Cposy,Looking_To,replyto,interface);
   }
/*
 | [act, take, coord?Cposx, coord?Cposy, symbol?Looking_To  ] => {
      relax
   }
 | [act, put, coord?Cposx, coord?Cposy, symbol?Looking_To  ] => {
      relax
   }
*/
 | [act, birth, integer?Cposx,
                                 integer?Cposy, symbol?Looking_To  ] => {
      real_world := being_born(real_world,Cposx,
                                                     Cposy,Looking_To,replyto,interface);
   }
 | [act, look, integer?Cposx,
                                 integer?Cposy, symbol?Looking_To ] => {
    world_model?cp := world_model?(real_world^/[Cposx,Cposy,
                                                              symbol,symbol,symbol,symbol]);
    [[any,any,any,any,any,symbol?bm]] = cp ;
    world_model?new_scene := vision_field(real_world,
                                                     Cposx,Cposy,bm) ;
    /* acknowledge execution */
    [ok, Cposx, Cposy, new_scene] >> replyto ;
   }
 | any?Msg  => trace("WORLD","Strange messagge %p\n",[Msg])
 } until end ;

};  /* of the world */


/* ************************************************** interface */
```

/* The following groups of routinesactually control the visualisation creating the graphical environment in Dialox. */

```
create_robot(symbol?name,integer?x,integer?y,symbol?bm) -> any[]?valof{
```

```
{ bm = rb_north => valis [addpic,
                                   Test,name,rob_north(name,x*sqmax,y*sqmax) ]
| bm = rb_east => valis [addpic,
                                   Test, name, rob_east(name,x*sqmax,y*sqmax) ]
| bm = rb_west => valis [addpic,
                                   Test, name, rob_west(name,x*sqmax,y*sqmax) ]
| bm = rb_south => valis [addpic,
                                   Test, name, rob_south(name,x*sqmax,y*sqmax) ]  }
```

```
};

move_object(symbol?name,integer?ox,integer?oy,integer?fx,integer?fy) -> any[]?valof{
 integer?Dx := (fx-ox)*100;
 integer?Dy := (fy-oy)*100;

trace("VIDEO","moving %s by %d %d \n",[name,Dx,Dy]);

 valis [move, Test, name, Dx, Dy ]
};


/* _______________________________________________ WorldProcess */
/* This process activates the X display and control the animation of the dock_land system.
  In order to activate this process:
      1.- start (or check active status) the April nameserver.
      2.- start (or check active status) the Dialox server.
      3.- start this process by executing:
                                   april dk_world <nameofworld> <?>

  It should be started before the dk_world process and before the robots are activated.
*/
WorldProcess(handle?wd, handle?dialox)
{
 /* Creating the world */
 [dialog,Patio,
      [column, 2,
      [row,[quit,Quit],[bitmap,xlogo32],[text,msg, ""]],
      [yellow, graph, Test, maxx, maxy],
      [row,[text,"Control Panel"],
           [button,Start,Start],
           [button,Freeze, Freeze],
           [button,Continue, Continue],
           [button,Select, Select],
           [button,Delete, Delete]
      ]]] >> dialox;

 /* Some visual help for the operator */
 [addpic,Test, axis, [group, [text, maxx/2-20,10, North],
                                        [text, maxx/2-20,maxy-10,South],
                                        [text, 5, maxy/2, West],
                    [text, maxx-30, maxy/2, East]
                  ]
 ] >> dialox ;

 {ok => relax
 | failed => writef(stdout,"Failed to create main dialog\n",[])
 };

 /* Setting the initial conditions */

 /* Controling the visualisation */
 repeat{
   [birth,symbol?name,integer?X,integer?Y,symbol?Dir] => {
      create_robot(name,X,Y,Dir) >> dialox ;
```

```
          }
 | Freeze => { Continue => relax /* Temporal stop... ack needed */
         }
 | Goal => { relax
         }
 /* Basic drawing capabilities */
 | [clear,symbol?name, integer?px,integer?py,?] => {
      handle?simul := replyto ;
      [delpic,Test,name] >> dialox ;
      /* Continue => ok >> simul ;  DEBUGGING */
   }
 | [draw, symbol?name, integer?px, integer?py, symbol?orient] => {
      handle?simul := replyto ;
      create_robot(name,px,py,orient) >> dialox ;
      /* Continue => ok >> simul ;  DEBUGGING */
   }

 /* Moving object in the display */
 | [move, symbol?name,
         integer?xi, integer?yi,
         integer?xf, integer?yf]  => {
     move_object(name,xi,yi,xf,yf) >> dialox
   }

 | ok => relax

 | any?M => writef(stdout,"Other message: %D\n",[M])

 } until ^Quit;

 end >> wd ;                              /* End of the world */
};
cat(symbol?X,symbol?Y) -> symbol? catenate(X,catenate(" ",Y));
```

```
/* Loading Dock World

  This file contains the piece of code that need to be replaced
  in the patio world, in order to display the warehouse. Observe
  that only the WorldProcess need to be replaced. The rest of
  the code is identical in both worlds.

  Version: 1.0   Robots` range of perception: the front cell

  Last modified: 14 - 6 - 94

  Jacinto Davila. FAIT 94. Imperial College.

*/


/* ************************************************** interface */
/* The following groups of routines actually control the  */
/* visualisation creating the graphical environment in Dialox. */
/*****************************************************************/

/* ___________________________________          ____ WorldProcess */
/* This process activates the X display and control
  the animation of the warehouse
  In order to activate this process:
      1.- start (or check active status) the April nameserver.
      2.- start (or check active status) the Dialox server.
      3.- start this process by executing:
                                  april dk_world <nameofworld> <?>

  It should be started before the dk_world process and before the robots are activated.
*/
WorldProcess(handle?wd, handle?dialox)
{
 /* Creating the world */
 [dialog,Warehouse,
      [column, 2,
      [row,   [quit,Quit],[bitmap,xlogo32],[text, msg, ""]],
      [yellow, graph, Test, maxx, maxy],
      [row,[text,"Control Panel"],
           [button,Start,Start],
           [button,Freeze, Freeze],
           [button,Continue, Continue],
           [button,Select, Select],
           [button,Delete, Delete]
      ]]] >> dialox;

 /* Some visual help for the operator */
 [addpic,Test, axis, [group, [text, maxx/2-20,10, North],
                                  [text, maxx/2-20,maxy-10,South],
                                  [text, 5, maxy/2, West],
                       [text, maxx-30, maxy/2, East]
                  ]
 ] >> dialox ;
```

```
/* Drawing the structures in the world */
```

```
 /* The shelves */
 [addpic,Test, shelves, [group,
                         [line, 0,100, 500,100],
                         [line, 0,200, 200,200],
                         [line, 200,200,200,300],
                         [line, 200,300,0,300],
                         [line, 300,200,500,200],
                         [line, 300,200,300,300],
                         [line, 300,300,500,300]
                  ]
 ] >> dialox ;

 /* The truck */
 [addpic,Test, truck, [group,
                         [line, 200,400, 300,400],
                         [line, 200,400, 200,700],
                         [line, 300,400, 300,700],
                         [line, 200,700, 300,700],
                         [line, 200,500, 300,500],
                         [line, 200,600, 300,600],
                         [line, 200,600, 220,660],
                         [line, 220,660, 280,660],
                         [line, 280,660, 300,600],
                         [line, 220,600, 240,640],
                         [line, 240,640, 260,640],
                         [line, 260,640, 280,600],
                         [line, 300,300,500,300]
                  ]
 ] >> dialox ;

 {ok => relax
 | failed => writef(stdout,"Failed to create main dialog\n",[])
 };

 /* Setting the initial conditions */

 /* Controling the visualisation */
 repeat{
   [birth,symbol?name,integer?X,integer?Y,symbol?Dir] => {

trace("VIDEO","Putting %s in the
              world at %d %d with face %s\n",[name,X,Y,Dir]);

      create_robot(name,X,Y,Dir) >> dialox ;

           }
 | Freeze => { Continue => relax /* Temporal stop... ack needed */
          }
 | Goal => { relax
          }
 /* Basic drawing capabilities */
 | [clear,symbol?name, integer?px,integer?py,?] => {

/* trace("VIDEO","Erasing %s",[name]); */
```

```
handle?simul := replyto ;
[delpic,Test,name] >> dialox ;
```

```
     /* Continue => ok >> simul ;  DEBUGGING */
   }
 | [draw,symbol?name,integer?px,integer?py, symbol?orient] => {
/*
trace("VIDEO","Drawing %s with symbol %s\n ",[name,orient]);
*/
     handle?simul := replyto ;
     create_robot(name,px,py,orient) >> dialox ;
     /* Continue => ok >> simul ;  DEBUGGING */
   }

 /* Moving object in the display */
 | [move, symbol?name,
         integer?xi, integer?yi,
         integer?xf, integer?yf]  => {
    move_object(name,xi,yi,xf,yf) >> dialox
   }
 | ok => relax

 | any?M => writef(stdout,"Other message: %D\n",[M])

 } until ^Quit;

 end >> wd ;             /* End of the world */
};
cat(symbol?X,symbol?Y) -> symbol? catenate(X,catenate(" ",Y));
```

common.ah

```
/* common.ah

  This file contains macros and functions used by robot.ap and
  the world simulators.
*/

/* ______________________________________________________________ macros */
/* trace( Process, OutputFile, Message, ListofObjects ); */
#macro trace( ?O, ?M, ?L) -> { relax
       writef( stdout,"%s(%s):",[ O, self]);
       writef( stdout, M, L );
       writef( stdout,"\n",[]);
};

/*                                       head */
head(any[]?T) -> any?valof{
       any[]?A := front(T,1);
       [any?Item] = A;
       valis Item
};

/____________________________________________________ extract */
extract(any[]?T) -> any[]?valof{
             valis back(T, |T|-1 )
};
```

world.db.patio

```
[ 0, 6, floor_, floor_, empty, floor_];
[ 1, 6, floor_, floor_, empty, floor_];
[ 2, 6, floor_, floor_, empty, floor_];
[ 3, 6, floor_, floor_, empty, floor_];
[ 4, 6, floor_, floor_, empty, floor_];
[ 0, 5, floor_, floor_, empty, floor_];
[ 1, 5, floor_, floor_, empty, floor_];
[ 2, 5, floor_, floor_, empty, floor_];
[ 3, 5, floor_, floor_, empty, floor_];
[ 4, 5, floor_, floor_, empty, floor_];
[ 0, 4, floor_, floor_, empty, floor_];
[ 1, 4, floor_, floor_, empty, floor_];
[ 2, 4, floor_, floor_, empty, floor_];
[ 3, 4, floor_, floor_, empty, floor_];
[ 4, 4, floor_, floor_, empty, floor_];
[ 0, 3, floor_, floor_, empty, floor_];
[ 1, 3, floor_, floor_, empty, floor_];
[ 2, 3, floor_, floor_, empty, floor_];
[ 3, 3, floor_, floor_, empty, floor_];
[ 4, 3, floor_, floor_, empty, floor_];
[ 0, 2, floor_, floor_, empty, floor_];
[ 1, 2, floor_, floor_, empty, floor_];
[ 2, 2, floor_, floor_, empty, floor_];
[ 3, 2, floor_, floor_, empty, floor_];
[ 4, 2, floor_, floor_, empty, floor_];
[ 0, 1, floor_, floor_, empty, floor_];
[ 1, 1, floor_, floor_, empty, floor_];
[ 2, 1, floor_, floor_, empty, floor_];
[ 3, 1, floor_, floor_, empty, floor_];
[ 4, 1, floor_, floor_, empty, floor_];
[ 0, 0, floor_, floor_, empty, floor_];
[ 1, 0, floor_, floor_, empty, floor_];
[ 2, 0, floor_, floor_, empty, floor_];
[ 3, 0, floor_, floor_, empty, floor_];
[ 4, 0, floor_, floor_, empty, floor_];
```

world.db.warehouse

```
[ 0, 6, floor_, floor_, empty, floor_];
[ 1, 6, floor_, floor_, empty, floor_];
[ 2, 6, truck, struct, full, truck_cabin];
[ 3, 6, floor_, floor_, empty, floor_];
[ 4, 6, floor_, floor_, empty, floor_];
[ 0, 5, floor_, floor_, empty, floor_];
[ 1, 5, floor_, floor_, empty, floor_];
[ 2, 5, box, struct, full, boxB_on_truck];
[ 3, 5, floor_, floor_, empty, floor_];
[ 4, 5, floor_, floor_, empty, floor_];
[ 0, 4, floor_, floor_, empty, floor_];
[ 1, 4, floor_, floor_, empty, floor_];
[ 2, 4, box, struct, full, boxA_on_truck];
[ 3, 4, floor_, floor_, empty, floor_];
[ 4, 4, floor_, floor_, empty, floor_];
[ 0, 3, floor_, floor_, empty, floor_];
[ 1, 3, floor_, floor_, empty, floor_];
[ 2, 3, floor_, floor_, empty, floor_];
[ 3, 3, floor_, floor_, empty, floor_];
[ 4, 3, floor_, floor_, empty, floor_];
[ 0, 2, shelf, struct, empty, shelf_cell];
[ 1, 2, shelf, struct, empty, shelf_cell];
[ 2, 2, floor_, floor_, empty, floor_];
[ 3, 2, shelf, struct, empty, shelf_cell];
[ 4, 2, shelf, struct, empty, shelf_cell];
[ 0, 1, floor_, floor_, empty, floor_];
[ 1, 1, floor_, floor_, empty, floor_];
[ 2, 1, floor_, floor_, empty, floor_];
[ 3, 1, floor_, floor_, empty, floor_];
[ 4, 1, floor_, floor_, empty, floor_];
[ 0, 0, shelf, struct, empty, shelf_cell];
[ 1, 0, shelf, struct, empty, shelf_cell];
[ 2, 0, floor_, floor_, empty, floor_];
[ 3, 0, shelf, struct, empty, shelf_cell];
[ 4, 0, shelf, struct, empty, shelf_cell];
[ 0, 7, wall, struct, full, wall];
[ 1, 7, wall, struct, full, wall];
[ 2, 7, wall, struct, full, wall];
[ 3, 7, wall, struct, full, wall];
[ 4, 7, wall, struct, full, wall];
[ 5, 7, wall, struct, full, wall];
[ 5, 0, wall, struct, full, wall];
[ 5, 1, wall, struct, full, wall];
[ 5, 2, wall, struct, full, wall];
[ 5, 3, wall, struct, full, wall];
[ 5, 4, wall, struct, full, wall];
[ 5, 5, wall, struct, full, wall];
[ 5, 6, wall, struct, full, wall];
```

exam1.ap

```
/* perpetual dialogue

  Elemental example of an APRIL multiagent implementation.
*/

/* _______________________________________________________ trace */
/* trace( Process, OutputFile, Message, ListofObjects ); */
#macro trace( ?O, ?M, ?L) -> {
       writef( stdout,"%s(%s):",[ O, self]);
       writef( stdout, M, L );
       writef( stdout,"\n",[]);
};

/* ____________________________________________________ _ agent2 */
agent2(){
 while true do {
   DoYouWantMeToTellYouTheBaldCockStory => {
     symbol?answer := genAnswer(yes_No);

trace("AGENT2","%s\n",[answer]);

     answer >> replyto
   }
 | quit => break
 }
};

/* _______________________________________________________ agent1 */
agent1(handle?agent2){

 DoYouWantMeToTellYouTheBaldCockStory >> agent2;

trace("AGENT1","Do you want me to tell you the story of the bald cock?..\n",[]);

 while true do {
   quit => break          /* the game is over */
 | any?answer => {

trace("AGENT1","It isn`t that %s, but if you want me to tell you the story of the bald cock..\n",[answer]);

     DoYouWantMeToTellYouTheBaldCockStory >> replyto
   }
 }
};

genAnswer(symbol?basicAnswer) -> symbol?valof{
 symbol?basename := gensym();
 [symbol,symbol]?splitname := split(basename, 7);
 [symbol,symbol?counter] = splitname;
 symbol?answer := catenate( basicAnswer, counter );
 valis answer            /* Return the just built answer */
} ;
```

exam1.ap

```
/* ___________________________________              main process  */
main(any[]?ar){

 if |ar| < 2 then {
   writef(stdout,"usage:
                                             april exem1 <agent2name> <agent1name>\n",[]);
   exit(1);
 };

 handle?grandson := handle?ar[1];
 handle?grandfather := handle?ar[2];

 /* the processes representing the agents are created */
 grandson names agent2() ;              /* fork call */
 grandfather names agent1(grandson) ;    /* fork call */

trace("MAIN","Starting the simulation\n",[]);

 { any?Msg => relax
 | timeout 2 secs => relax };
                                                          /* the simulation longs as most 2 seconds */

trace("MAIN","Ending the simulation \n",[]);

 /* The main process inform the agents the game is over */
 quit >> grandfather;
 quit >> grandson;

};
```

APPENDIX B. AGENT'S BRAINS IN PROLOG

This appendix shows the source-codes of the following programs written in PROLOG:

- **`reacbrain.pl`**. It contains the reactive model of brain including the locus of control algorithm, the knowledge assimilation meta-interpreter and the object-level hierarchy of goals.

- **`planbrain.pl`**. It contains the locus of control and the meta-interpreter enhanced for planning with bounded abducibility. The object level knowledge is not included (it is the same that in `reacbrain.pl`).

Within the code of `reacbrain.pl` the reader will find the description of the model of the world (`cell/5` predicate) that the agent maintains.

```
/* reacbrain.pl

  Knowledge assimilation in logic-based agents.

  This programm implements the main execution-assimilation cycle
  within the agent's  brain. It also contains the rules of
  behaviour this agent will follow.

  features:
  1.- The metapredicate stops as soon as it can abduce the
     first do term or the depth is N. As this brain only decide
     the next action to be executed is regarded as a reactive
     brain.
  2.- The object level and meta-level predicates include
     time representation but this is not actually used by the
     program.

  Last modified: 30 Aug 1994

*/

/*****************************************************************/
/*this first section implements the locus of control algorithms & */
/*activates the APRIL-PROLOG interfaces. */
/*****************************************************************/

:- dynamic cell/6, current_sit/1, stat/3.

agent( Name, InitialGoal, N ) :-
 ap_init( Name, 0 ),
 ap_receive( From, _ ),      /* Receive the wake up message */
 writef("Agent %w running with time resource = %w\n",[Name,N]),
 cycle( From, InitialGoal, N ),
 ap_end.

agent(_, _, _) :- ap_end.

cycle( Agent, Goals, N ) :- !,
 observe( Input ),
 To = Counter,                   /* Initial time for planning */
 update_stat( counter, [cycle] ),
 stat( counter, cycle, Counter ),
 /* writef("Assim-Exec Cycle number %w \n",[Counter]), */
 assimilate( Input, Goals, NGoals, To, N, _ ),
 execute( Agent, NGoals, NextGoals),
 ( pause( NextGoals ) ->
   cycle( Agent, NextGoals, N );
   true ).

observe(Input) :-
 ap_receive(_, Message),                /* Observe message */
 /* writef("Receive new scenario %w\n",[Message]), */
 Message = [observe,[X,Y,F,St,Input]],
 ( retract( current_sit( _ ) ) -> true ; true ),
 assert(current_sit( ( X, Y, F, St ) ) ).
```

```
update_world_model( [] ).
update_world_model( [[ X, Y, Obj, Typ, St, Bm ]|Rest] ) :- !,
 ( retract( cell( X, Y, _, _, _, _ ) ) ->
   /* writef("My World has changed\n"), */ true ;
   /* writef("My World has been extended\n"), */ true ),
 assert( cell( X, Y, Obj, Typ, St, Bm ) ),
 update_world_model( Rest ).


execute( Agent, OldGoals, NewGoals ) :-
 process_goal( OldGoals, Plan, SamePlan, OtherPlan ),
 /* writef("Sending new plan %w to my body\n",[Plan]), */
 ap_send( Agent, [plan,Plan] ),
 ap_receive( Agent, Ack ),
 ap_send( Agent, go ),   /* Sincronizing. Shouldn`t be needed */
 /* writef("Last action %w %w \n",[Plan, Ack]), */
 ( Ack == succeed ->
   (  update_stat( success, Plan ), NewGoals = SamePlan ) ;
   (  update_stat( failure, Plan ), NewGoals = OtherPlan ) ).

process_goal( [Goal|AltG], Plan, NewGoals, RealAltG ) :-
 ( Goal = (do(Act,T), _) ->
   ( Action = (do(Act,T)), Plan = [Act] ) ;
   ( Action = (do(look,0)), Plan = [look] ) ),
 ( Act == look -> display_stat; true ),
             /* displaying stat at the end */
 /* writef("\nExecuting \n\n",[]), printconj(Goal), */
 pop( Action, [Goal|AltG], NewGoals, RealAltG).
 /* writef("Next subgoals in this set:\n",[]), */
 /* printl( NewGoals), writef(" Alternatives\n",[]), printl(RealAltG) .*/


/* ************************************************************ */
/* This is the module of assimilation to be inserted into the    */
/* robots' brains                                        */
/* ************************************************************ */

/* ______________________________________________________ assimilate */

assimilate( Input, Gs, NGs, T, N, Tn ) :-
 update_world_model( Input ),
 /* writef("\n Assimilating %w \n",[Gs]), */
 ( resolve( N, Gs, NewGoals ) ->
   NGs = NewGoals ;
   ( writef("These goals %w don't make any sense\n",[Gs]),
     NGs = [(true)]
   )
 ),
 Tn is T + N.


resolve( 0, Goals, Goals ).                    /* Base cases */
resolve( _, [(true)], [(true)] ).

resolve( N, [true|Rest], NGoals ) :-
```

```
NNext is N - 1,
 resolve( NNext, Rest, NGoals ).

resolve( N, Goals, NGoals ) :-
 /* demostrating upon the current KB */
 Goals = [FirstAlt|RemAlt],
 demo( FirstAlt, NewSet ),          /* G <-> body1 or body2 .. */
 append( NewSet, RemAlt, NextGoals ),      /* depth first like */
 NNext is N - 1,
 resolve( NNext, NextGoals, NGoals ).

resolve( N, [_|RemAlt], NGoals ) :-
 NNext is N - 1,
 resolve( NNext, RemAlt, NGoals ).


demo( true, [(true)] ).

demo( (true, R), NewGoals ) :- !, demo( (R), NewGoals ).

demo( (not G, R), [(R)] ) :- not demo( G, _ ).

demo( (G, R), [(R)] ) :-
 predicate_property(G, built_in), !, G /*,
 writef("Solving built-in %w \n",[G])  */.

demo( (G, R), NewList ) :-
 /* Stop demo and prepare for execution */
 abducible( G ), !,
 push( G, [(R)], NewList).

demo( (G, R), NewList ) :-     /* reducing actions and subgoals */
 /* writef("\n Resolving %w \n",[G]), */
 findall( BB,
         ( clause( G, Body ), and_append( Body, R, BB ) ),
         NewList ),
 /*
 writef("\n\n New Set \n { ",[]),
 printdisj( NewList ),
 writef("\n } \n ",[]),
 */
 ( NewList = [] -> ( fail, ! ); true ).


push( _, [], [] ).
push( G, [F|R], [N|NR] ) :- !,
 and_append( G, F, N ),
 push( G, R, NR ).


pop( Act, [F|R], [N|NR], AltR ) :-
 cut( Act, F, N ),
 pop( Act, R, NR, AltR ).
pop( do(look,0), Rest, Rest, []) :- !.
   /* action do(look,0) has special function */
pop( _, Rest, [], Rest ).
```

```
cut( R_Act, (P_Act, Rest), Rest ) :- R_Act == P_Act.
```

```
and_append( First, Second, Result ) :- !,
 first_part( First, Second, Result ).
first_part( (A, R), Second, (A, Rest) ) :- !,
 first_part( R, Second, Rest ).
first_part( Last, Second, (Last, Rest2) ) :- !,
 second_part( Second, Rest2 ).
second_part( (A, R), (A, Rest2) ) :- !,
 second_part( R, Rest2 ).
second_part( A, A ).


printl( [] ).
printl( [A|B] ) :- writef(" [ ",[]),
                andprint( A ),
                writef(" ] ",[]), printl( B ), !.

andprint( (A,B) ) :-
 ( A = do(_,_) -> ( writef("-Act- %w ",[A]), andprint( B ) );
                ( writef("-+- %w ",[A]), andprint( B ) ) ).
andprint( A ) :-
 ( A = do(_,_) -> writef("-Act- %w ",[A]) ;
                writef("-+- %w ",[A]) ).

printconj( (A,B) ) :-
  writef(" %w ^ ",[A]), printconj( B ).
printconj( B ) :-
  writef("%w ",[B]).

printdisj( [] ) :- writef("]",[]).
printdisj( [A|B] ) :- writef("\n    [",[]),
                  printconj(A),
                  writef("] -+- \n    [",[]), printdisj( B ), !.



/*****************************************************************/
/* This section contains domain specific definitions            */
/* and the object level rules                                   */
/*****************************************************************/

abducible( G ) :- G=..[P|_], abd( P ), !.
             /* Add Instantiation check */
abd(do).

/* ______________________________________________________ closer */
/* Test whether (X,Y) is closer to Gx,Gy than Cx,Cy             */

closer( (_, _, _), (X, Y, D), (X, Y, D) ) .
closer( (X, Y, east), (X, Y, north), (X, Y, north) ).
closer( (X, Y, east), (X, Y, north), (X, Y, west) ).
closer( (X, Y, east), (X, Y, south), (X, Y, south) ).
closer( (X, Y, west), (X, Y, north), (X, Y, north) ).
closer( (X, Y, west), (X, Y, north), (X, Y, east) ).
closer( (X, Y, west), (X, Y, south), (X, Y, south) ).
closer( (X, Y, north), (X, Y, east), (X, Y, east) ).
closer( (X, Y, north), (X, Y, west), (X, Y, west) ).
closer( (X, Y, north), (X, Y, east), (X, Y, south) ).
closer( (X, Y, south), (X, Y, east), (X, Y, east) ).
closer( (X, Y, south), (X, Y, west), (X, Y, west) ).
closer( (X, Y, south), (X, Y, west), (X, Y, north) ).
closer( (CX, CY, _), (X, Y, _), (GX, GY, _) ) :-
 IniX is abs(GX - CX),
 IniY is abs(GY - CY),
 DIni is sqrt(IniX*IniX + IniY*IniY),
 NwX is abs(GX - X),
 NwY is abs(GY - Y),
 DNw is sqrt(NwX*NwX + NwY*NwY),
 DNw < DIni.
closer( (CX, CY, _), (X, Y, east), (GX, GY, _) ) :-
 IniX is abs(GX - CX),
 IniY is abs(GY - CY),
 NwX is abs(GX - X),
 NwY is abs(GY - Y),
 NwX =< IniX, NwY = IniY, CX < GX.
closer( (CX, CY, _), (X, Y, west), (GX, GY, _) ) :-
 IniX is abs(GX - CX),
 IniY is abs(GY - CY),
 NwX is abs(GX - X),
 NwY is abs(GY - Y),
 NwX =< IniX, NwY = IniY, CX > GX.
closer( (CX, CY, _), (X, Y, south), (GX, GY, _) ) :-
 IniX is abs(GX - CX),
 IniY is abs(GY - CY),
 NwX is abs(GX - X),
 NwY is abs(GY - Y),
 NwX = IniX, NwY =< IniY, CY < GY.
closer( (CX, CY, _), (X, Y, north), (GX, GY, _) ) :-
 IniX is abs(GX - CX),
 IniY is abs(GY - CY),
 NwX is abs(GX - X),
 NwY is abs(GY - Y),
 NwX = IniX, NwY =< IniY, CY > GY.


/* _____________________________________________ evasive actions */
/* Precompiled primitive action for avoiding obstacules */

change_dir( (X, Y, north), (X, Y, east), T ) :-
 do(t_right, T).
change_dir( (X, Y, south), (X, Y, west), T ) :-
 do(t_right, T).
change_dir( (X, Y, east), (X, Y, south), T ) :-
 do(t_right, T).
change_dir( (X, Y, west), (X, Y, north), T ) :-
 do(t_right, T).


move_forward( (X, Y, north), (X, Ny, north), T ) :-
 Ny is Y - 1, not prohibited( (X, Ny, north), T ), do(m_forward, T ).
move_forward( (X, Y, south), (X, Ny, south), T ) :-
```

```
Ny is Y + 1, not prohibited( (X, Ny, south), T ), do(m_forward, T ).
```

reacbrain.pl

```
move_forward( (X, Y, east), (Nx, Y, east), T ) :-
 Nx is X + 1, not prohibited( (Nx, Y, east), T ),
 do(m_forward, T ).
move_forward( (X, Y, west), (Nx, Y, west), T ) :-
 Nx is X - 1, not prohibited( (Nx, Y, west), T ),
 do(m_forward, T ).

/* ________________________________________________ prohibited */
prohibited( (X,Y,_), _ ) :-
 cell( X, Y, _, struct, _, _ ).
prohibited( (X,Y,_), _ ) :-
 not cell( X, Y, _, _, _, _ ) .


/* _______________________________________________ atom_action */
atom_action( m_forward, (X, Y, north), (X, Y2, north) ) :-
 Y2 is Y - 1.
atom_action( m_forward, (X, Y, south), (X, Y2, south) ) :-
 Y2 is Y + 1.
atom_action( m_forward, (X, Y, east), (X2, Y, east) ) :-
 X2 is X + 1.
atom_action( m_forward, (X, Y, west), (X2, Y, west) ) :-
 X2 is X - 1.
atom_action( t_right, (X, Y, north), (X, Y, east) ).
atom_action( t_right, (X, Y, east), (X, Y, south) ).
atom_action( t_right, (X, Y, south), (X, Y, west) ).
atom_action( t_right, (X, Y, west), (X, Y, north) ).
atom_action( t_left, (X, Y, north), (X, Y, west) ).
atom_action( t_left, (X, Y, west), (X, Y, south) ).
atom_action( t_left, (X, Y, south), (X, Y, east) ).
atom_action( t_left, (X, Y, east), (X, Y, north) ).

/* _______________________________________________ World Model */
/* Cell( CoordX, CoordY, Name, Type, Status, Bitmap */


/* __________________________________ goal <- cond, action rules */

go( Cs, Cs, To, _ ) :-
 do( look, To ).

go( Cs, Fs, To, Tn ) :-
 gradient_step( Cs, Ns, Fs, To ), T1 is To + 1,
 go( Ns, Fs, T1, Tn ).

go( Cs, Fs, To, Tn ) :-
 avoidance_steps( Cs, Ns, Fs, To, Tf ), T1 is Tf + 1,
 go( Ns, Fs, T1, Tn ).


gradient_step( CurrentSit, NewSit, FinalSit, T ) :-
 atom_action( Action, CurrentSit, NewSit ),
 closer( CurrentSit, NewSit, FinalSit ),
 not prohibited( NewSit, T ),
 do( Action, T ).
```

```
avoidance_steps( CurrentSit, NextSit, FinalSit , T, Tf ) :-
/*follow the wall*/
```

```
 change_dir( CurrentSit, NewSit, T ),
 T1 is T + 1,
 avoidance_step2( NewSit, NextSit, FinalSit, T1, Tf ).


avoidance_step2( CurrentSit, NextSit, _, T, Tf ) :-
 move_forward( CurrentSit, NextSit, T ),
 Tf is T + 1.

avoidance_step2( CurrentSit, NexSit, FinalSit, T, Tf ) :-
 T1 is T + 1,
 avoidance_steps( CurrentSit, NexSit, FinalSit, T1, Tf ).


at(S, Int) :-
 time_now( To ), current_sit((Cx,Cy,Cd,_)), T is To + Int,
 go((Cx,Cy,Cd), S, To, T).


walk_around( Sa, _ ) :-
 current_sit( (Cx, Cy, Cd, _)),

go_around( So, Sf ) :-
 go( So, Sf, 1, 1000 ), go_around( Sf, So ).


/*************************************************************/
/* ___________________________________________Auxiliary predicates */

time_now(T) :-  get_time(T).

update_stat( Result, [Act] ) :-
 retract(stat( Result, Act, Old )) ->
 ( New is Old + 1, assert(stat( Result, Act, New )) ) ;
 ( assert(stat( Result, Act, 1 )) ).

display_stat :-
 findall( ( Result, Act, Num ), stat( Result, Act, Num ), L ),
 writef("\n\n Statistics:  ",[]),
 printl( L )
 /* retractall(stat(_,_,_)) */.


/*************************************************************/
/* _______________________________________________________testers */


test1a(N) :- agent(brainof_ja,
                              [(walk_around((4,3,west), 200), true)],N).

test1b(N) :- agent(brainof_jb,
                              [(walk_around((0,3,east), 200), true)],N).

test1c(N) :- agent(brainof_jc,
                              [(walk_around((3,6,north), 200), true)],N).
```

```
test1d(N) :- agent(brainof_jd,
                                        [(walk_around((1,6,north), 200), true)],N).
```

```
/* planbrain.pl

  Knowledge assimilation in logic-based agents.

  This programm implements the main assimilation cycle within
  the agent's  brain.

  features:
  1.- The metapredicate does not stop  when it gets the first
      executable goal. This is an agent with planning capabilities.
  2.- The object level and meta-level predicates include
      time representation but this is not actually used by the
      program.
  3.- This version also limits the number of abduced terms that the
      agent can have at any time.

  Last modified: 30 Aug 1994

*/

/**************************************************************/
/* this first section implement the locus of control algorithms  */
/* and activate the APRIL-PROLOG interfaces.                  */
/**************************************************************/

:- dynamic cell/6, current_sit/1, stat/3, abd_count/1.

agent( Name, InitialGoal, N, M ) :-
 ap_init( Name, 0 ),
 ap_receive( From, _ ),         /* Receive the wake up message */
 writef("Agent %w ;
             time resource = %w ; abducing limit = %w\n",[Name, N, M]),
 init_abd_count( M ),
 cycle( From, InitialGoal, N ),
 ap_end.

agent(_, _, _) :- ap_end.


cycle( Agent, Goals, N ) :- !,
 observe( Input ),
 To = Counter,                    /* Initial time for planning */
 update_stat( counter, [cycle] ),
 stat( counter, cycle, Counter ),
 /* writef("Assim-Exec Cycle number %w \n",[Counter]), */
 assimilate( Input, Goals, NGoals, To, N, _ ),
 execute( Agent, NGoals, NextGoals),
 ( pause( NextGoals ) ->
   cycle( Agent, NextGoals, N );
   true ).


observe(Input) :-
 ap_receive(_, Message),                  /* Observe message */
```

```
/* writef("Receive new scenario %w\n",[Message]), */
Message = [observe,[X,Y,F,St,Input]],
( retract( current_sit( _ ) ) -> true ; true ),
 assert(current_sit( ( X, Y, F, St ) ) ).


update_world_model( [] ).
update_world_model( [[ X, Y, Obj, Typ, St, Bm ]|Rest] ) :- !,
 ( retract( cell( X, Y, _, _, _, _ ) ) ->
   /* writef("My World has changed\n"), */ true ;
   /* writef("My World has been extended\n"), */ true ),
 assert( cell( X, Y, Obj, Typ, St, Bm ) ),
 update_world_model( Rest ).


execute( Agent, OldGoals, NewGoals ) :-
 process_goal( OldGoals, Plan, SamePlan, OtherPlan ),
 /* writef("Sending new plan %w to my body\n",[Plan]), */
 ap_send( Agent, [plan,Plan] ),
 ap_receive( Agent, Ack ),
 ap_send( Agent, go ),      /* Sincronizing. Shouldn`t be needed */
 /* writef("Last action %w %w \n",[Plan, Ack]), */
 ( Ack == succeed ->
   (  update_stat( success, Plan ), NewGoals = SamePlan,
      up_abd ) ;
   (  update_stat( failure, Plan ), NewGoals = OtherPlan,
      reinit_abd )
 ).

process_goal( [Goal|AltG], Plan, NewGoals, RealAltG ) :-
 ( Goal = (do(Act,T), _) ->
   ( Action = (do(Act,T)), Plan = [Act] ) ;
   ( Action = (do(look,0)), Plan = [look] ) ),
 ( Act == look -> display_stat; true ),  /* displaying stat at the end */
 /* writef("\nExecuting \n\n",[]), printconj(Goal), */
 pop( Action, [Goal|AltG], NewGoals, RealAltG).
 /* writef("Next subgoals in this set:\n",[]), */
 /* printl( NewGoals), writef(" Alternatives\n",[]), printl(RealAltG) .*/


/* ************************************************************** */
/* This is the module of assimilation to be inserted into the    */
/* robots' brains                                              */
/* ************************************************************** */

/* __________________________________________________assimilate */

assimilate( Input, Gs, NGs, T, N, Tn ) :-
 update_world_model( Input ),
 /* writef("\n Assimilating %w \n",[Gs]), */
 ( resolve( N, Gs, NewGoals ) ->
   NGs = NewGoals ;
   ( writef("These goals %w don't make any sense\n",[Gs]),
     NGs = [(true)]
   )
 ),
 Tn is T + N.
```

planbrain.pl

```
resolve( 0, Goals, Goals ).                    /* Base cases */
resolve( _, [(true)], [(true)] ).

resolve( N, [true|Rest], NGoals ) :-
 NNext is N - 1,
 resolve( NNext, Rest, NGoals ).

resolve( N, Goals, NGoals ) :-
 /* demostrating upon the current KB */
 select( Goals, ReadySet, FirstAlt, RemAlt ),
 demo( FirstAlt, NewSet ),        /* G <-> body1 or body2 .. */
 combine( ReadySet, NewSet, RemAlt, NextGoals ),
 NNext is N - 1,
 resolve( NNext, NextGoals, NGoals ).

resolve( N, Goals, NGoals ) :-
 select( Goals, ReadySet, _, RemAlt ), /* Not very efficient */
 NNext is N - 1,
 append( ReadySet, RemAlt, NextGoals ),
 resolve( NNext, NextGoals, NGoals ).


select( [(true)|RG], [], true, RG ).

select( [FG|RG], [FG|RReady], FirstAlt, RemAlt ) :-
 fullplan( FG ),
 select( RG, RReady, FirstAlt, RemAlt ).

select( [FG|RG], [], FG, RG ).


combine( ReadySet, NewSet, RemAlt, NextGoals ) :-
 append( ReadySet, NewSet, TempSet ),/* Depth first search*/
 append( TempSet, RemAlt, NextGoals ).


fullplan( (G, R) ) :-
 ( abducible( G ); G = true ),
 fullplan( R ).
fullplan( G ) :- !,
 G = true.


demo( true, [(true)] ).

demo( (true, R), NewGoals ) :- !, demo( (R), NewGoals ).

demo( (not G, R), [(R)] ) :- not demo( G, _ ).

demo( (G, R), [(R)] ) :-
 predicate_property(G, built_in), !.

demo( (G, R), NewList ) :-
 abducible( G ), !,
 ( no_more_abd ->
   ( NewList = [(G, R)] ) ;
( demo( R, NewSet ), push( G, NewSet, NewList), down_abd )
```

```
 ).

demo( (G, R), NewList ) :-     /* reducing actions and subgoals */
 findall( BB,
         ( clause( G, Body ), and_append( Body, R, BB ) ),
         NewList ),
 ( NewList = [] -> ( fail, ! ); true ).


push( _, [], [] ).
push( G, [F|R], [N|NR] ) :- !,
 and_append( G, F, N ),
 push( G, R, NR ).


pop( Act, [F|R], [N|NR], AltR ) :-
 cut( Act, F, N ),
 pop( Act, R, NR, AltR ).
pop( do(look,0), Rest, Rest, []) :- !.
   /* action do(look,0) has special function */
pop( _, Rest, [], Rest ).


cut( R_Act, (P_Act, Rest), Rest ) :- R_Act == P_Act.


and_append( First, Second, Result ) :- !,
 first_part( First, Second, Result ).
first_part( (A, R), Second, (A, Rest) ) :- !,
 first_part( R, Second, Rest ).
first_part( Last, Second, (Last, Rest2) ) :- !,
 second_part( Second, Rest2 ).
second_part( (A, R), (A, Rest2) ) :- !,
 second_part( R, Rest2 ).
second_part( A, A ).


printl( [] ).
printl( [A|B] ) :- writef(" [ ",[]),
                andprint( A ),
                writef(" ] ",[]), printl( B ), !.

andprint( (A,B) ) :-
 ( A = do(_,_) -> ( writef("-Act- %w ",[A]), andprint( B ) );
                ( writef("-+- %w ",[A]), andprint( B ) ) ).
andprint( A ) :-
 ( A = do(_,_) -> writef("-Act- %w ",[A]) ;
                writef("-+- %w ",[A]) ).

printconj( (A,B) ) :-
  writef(" %w ^ ",[A]), printconj( B ).
printconj( B ) :-
  writef("%w ",[B]).

printdisj( [] ) :- writef("]",[]).
printdisj( [A|B] ) :- writef("\n    [",[]),
```

```
                    printconj(A),
                    writef("] -+- \n    [",[]), printdisj( B ), !.


init_abd_count( M ) :-
 ( retract(abd_count(_)) -> true; true ),
 ( retract(abd_init(_)) -> true; true ),
 assert(abd_count(M)),
 assert(abd_init(M)).


reinit_abd :- abd_init(M),
                              retract(abd_count(_)), assert(abd_count(M)).


no_more_abd :-  abd_count(M), M =< 0.


up_abd :- retract(abd_count(M)), NM is M + 1, assert(abd_count(NM)).


down_abd :- retract(abd_count(M)),
                            NM is M - 1, assert(abd_count(NM))

/****************************************************************/
/* This section contains domain specific definitions and the     */
/* object level rules                                           */
/****************************************************************/

/* see the details of this section inside the code of reacbrain */
/* the object level knowledge in both brains is the same   */
```

APPENDIX C. SOURCES OF THE PROLOG-APRIL INTERFACE

This appendix contains the following source-codes:

- **`stub.c`**. This file describes or declares for the SWI-PROLOG those routines and procedures (PROLOG predicates) that are being added to the original PROLOG through the C-interface.

- **`pl_ap.c`**. This file contains the source-code of those procedures that implements the PROLOG-APRIL interface and other auxiliary procedures and system routines. Observe that this code contains calls to procedures that are part of the supporting TCP/IP platform, implemented as a separated library.

- **`pl-prims.c`**. This is the only file within the SWI-PROLOG original source-codes that required alteration. We include precisely the piece of code (the `pl_halt` procedure) that was changed.

- **`talker.ap`**. It is small example of a program written in APRIL that can be used to interact with a PROLOG program with the APRIL-PROLOG extensions. Observe that this program is an ordinary APRIL program.

- **`talker.pl`**. This is a PROLOG code (with the APRIL-PROLOG predicates) that can interact with the program `talker.ap`.

stub.c

```
/*  stub.c,v 1.1 1992/07/17 12:40:36 jan Exp

   Copyright (c) 1991 Jan Wielemaker. All rights reserved.
   jan@swi.psy.uva.nl

   Purpose: Skeleton for extensions to SWI-PROLOG. (see manual).

   PROLOG-APRIL agent interface:
   Interface for communicating a prolog agent with other agents
   in an April environment. The interface implements a set of 4
   primitives communicating predicates. This predicates invoke
   the procedures for initialising the channels of communications,
   sending and receiving messages and closing the channels.
   This version of the interface is implemented on top of a TCP/IP
   plataform, using UDP as the transport layer protocol. However
   some previsions are taken for ensuring reliable communication.

   Implemented by: Jacinto Alfonso Da'vila.

   Last modified: 24 - 5 - 94.

   The TCP/IP plataform was provided by Prof. Frank McCabe.

   Imperial College. London.

*/

#include <stdio.h>
#include "SWI-Prolog.h"

extern foreign_t pl_ap_init P((term, term ));
extern foreign_t pl_ap_send P((term, term));
extern foreign_t pl_ap_receive P((term, term));
extern foreign_t pl_ap_end P((term));

PL_extension PL_extensions [] =
{
/*{ "name",         arity,  function,    PL_FA_<flags> },*/
 { "ap_init", 2,       pl_ap_init,  0 },
 { "ap_send", 2,       pl_ap_send,  0 },
 { "ap_receive", 2,       pl_ap_receive,  0 },
 { "ap_end", 0,       pl_ap_end,  0 },
 { NULL,            0,          NULL,               0 }        /* terminating line */
};
```

```
/* pl_ap.c

  This file contains the definition of the C-functions which implement the  April-PROLOG interface over the TCP libraries.
  These functions support the predicates which allow a prolog agent to communicates with other agents in an April environment.

  The actual program that controls this process should be written in PROLOG and executed by the prolog agent. (see file talker.pl )

  Limitation: The current interface only make provision for remembering the last agent from which it received messages. Therefore the prolog agent only can send messages to the last APRIl process that send a request to PROLOG. This code can be enhanced to accomodate a directory of names and addresses of APRIL process, allowing the PROLOG agent to comunicate with them without the previous restriction.

                This version allow the interchange the following types
                of data (from PROLOG to the equivalent APRIL and vice
                versa): integers, atoms (april symbols) and strings
                (also april symbols), real numbers and tuples nested
                up to 16 times (april tuples are converted in
                        prolog list).
                Prolog predicates are no allowed for conversion
                        into april format.

  bugs: The SWI-PROLOG agents have shown anomalous activities in certain ocasions (after calling the garbage collector, the program stop). More testing and debugging of the UNIX signals used by SWI-PROLOG and this interface is required.

            There are also some problems transmiting floating point numbers from APRIL to PROLOG.

  Last modified: 30 - Jul - 94.

  Jacinto Da'vila.

  Knowledge Assimilation in Multiagents Systems. FAIT Course. 1994.
*/

#include "SWI-Prolog.h"
#include <ctype.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>        /* define u_char etc. */
#include <sys/time.h>
#include <sys/socket.h>       /* Comms stuff */
#include <netinet/in.h>
#include <errno.h>
#include <signal.h>
#include <malloc.h>
```

```
#include "queue.h"
#include "tcp.h"
#include "fnames.h"
#include "logical.h"

#define MSGTRACE
#define functor_name(term) PL_atom_
                                        value(PL_functor_name(PL_functor(term)))

logical traceTCP=False;
logical traceMsg=True;

char* PLDefAgentname ="PLAgent";
char* PLAgentname ;
int CSocket = 0;
static addrpo LastQuerier = NULL ;
            /* This was the last process which sent a msg */

u_char plxbuff[MAX_TCP_MSG], *bufpo;
                                           /* pointers to the tcp buffer area */

/* ****************************************** Support functions */

/* _______________________________________________ SWI-Refresh */

void SWI_Refresh(sig, code, scp, addr)
int sig, code;
struct sigcontext *scp;
char *addr;
{
 RefreshQ();               /* refresh the message queue */
 alarm(3);                  /* schedule a new refresh in 3 seconds */

};

/* ____________________________________________________ fatal_error_handler */

fatal_error_handler(sig,type,scp,addr)
int sig, type;
struct sigcontext *scp;
char *addr;
{
 deregister() ;            /* Deregistering before quit */
 deliverSignal(sig, type, scp, addr);
                                        /* Be carefull. PROLOG may not know it */
 fprintf(stdout,"FATAL ERROR (Signal %s)\n",sig) ;
};

/* _________________________________________________________________ sig_int */
/*
* cleanup after CTRL-C and other major errors
*/
void sig_int()
{
 deregister();
 exit(SIGINT);
};
```

```
void sig_quit_agent()
{
 deregister();
 printf("Prolog Agent says bye bye.. \n");
};

char *itoa(long n)
{
 static char nm[100];

 sprintf(nm,"%d",n);

 return &nm[0];
};

/* _______________________________________________________ dispatch_event */
int dispatch_events(void){

 fd_set fdset;
 struct timeval timeout;
 int status;
 int width = 1 ;

 FD_ZERO(&fdset);
 FD_SET(0, &fdset); /*  Checking the standard input only */
 timeout.tv_sec = 2L;
 timeout.tv_usec = 0L;
 status = select(width, &fdset, NULL, NULL, &timeout);

 if (status > 0) return(PL_DISPATCH_INPUT);
 else
   if (status == 0) return(PL_DISPATCH_TIMEOUT);
   else
     if (errno==EINTR) return(PL_DISPATCH_TIMEOUT);
     else {
        perror("select() error ");
        exit(1);
     }
};


/* ___________________________________________________________ do_wait */
static int do_wait(int sock)
{
 fd_set fdset;
 struct timeval timeout;
 int status;

again:
 FD_ZERO(&fdset);
 FD_SET(sock, &fdset);
 timeout.tv_sec = 2L;
 timeout.tv_usec = 0L;
 status = select(sock+1, &fdset, NULL, NULL, &timeout);
```

```
if (status == 0) goto again;
```

pl_ap.c

```
 if (status == -1 ) {
   if (errno == EINTR)
     goto again;
   else {
     perror("select() error ");
     exit(1);
   }
 }
 return(1);
} ;


/* ************************************************************ */
/* ______________________________________________________ decode_msg */
/*Tranforms the message from the encoded form into a PROLOG term*/
u_char *decode_msg(term m,u_char *c)
{
  char s[256] ; /* Auxiliary buffers */

 switch((*c)&TAGMASK){
 case S_INT:
   PL_unify_atomic(m,PL_new_integer((long)((*c++)&VALMASK)));
   break;
 case INT:{
   int i = *(c+4);
   i |= *(c+3)<<010;
   i |= *(c+2)<<020;
   i |= *(c+1)<<030;
   c+=5;
   PL_unify_atomic(m,PL_new_integer(i));
   break;
 }
 case S_SYMB:{
   int len = (*c++)&VALMASK;
   bcopy(c,s,(len<256?len:255));
   s[(len<256?len:255)]='\0';
   PL_unify_atomic(m, PL_new_atom(s));
   c+=len;
   break;
 }
 case SYMB:{
   int len = *(c+1)<<010 | *(c+2);
   c+=3;
   bcopy(c,s,(len<256?len:255));
   s[(len<256?len:255)]='\0';
   PL_unify_atomic(m, PL_new_atom(s));
   c+=len;
   break;
 }

 case FLT:{
   char len = abs(*(signed char*)c);
   double flt = UnConvertFP(c);
   c+= len+1;
```

```
  PL_unify_atomic(m, PL_new_float(flt));
  break;
}

case S_TPL:{
  int len = (*c++)&VALMASK;
  term arg1,arg2, temp;
  int count = 1;

  if (len == 0) {
    PL_unify_atomic(m, PL_new_atom("[]"));
    break;
  };

  PL_unify_functor(m, PL_new_functor(PL_new_atom("."), 2));
  temp = m;
  while(count<=len){
    arg1 = PL_arg(temp,1);
    c = decode_msg(arg1,c);
    if (count==len) {
      PL_unify_atomic(PL_arg(temp,2), PL_new_atom("[]"));
    } else {
      arg2 = PL_arg(temp,2);
      PL_unify_functor(arg2, PL_new_functor(PL_new_atom("."), 2));
              temp = arg2 ;
    };
    count++ ;
  }
  break;
}

case TPL:{
  term arg1,arg2, temp;
  int count = 1;
  char *ch = "";
  int len = *(c+1)<<010 | *(c+2);
  c+=3;

  if (len == 0) {
    PL_unify_atomic(m, PL_new_atom("[]"));
    break;
  };

  PL_unify_functor(m, PL_new_functor(PL_new_atom("."), 2));
  temp = m;
  while(count<=len){
    arg1 = PL_arg(temp,1);
    c = decode_msg(arg1,c);
    if (count==len) {
      PL_unify_atomic(PL_arg(temp,2), PL_new_atom("[]"));
    } else {
      arg2 = PL_arg(temp,2);
      PL_unify_functor(arg2, PL_new_functor(PL_new_atom("."), 2));
      temp = arg2 ;
    };
```

```
count++ ;
```

```
   }
   break;
  }

  default:
   fprintf(stderr,"Unknown format type: %#x\n",*c++);
  }
  return c;
};


/* ____________________________________________________________ flat_list */
/* This procedure receives a list and transforms it in a printable
  string. The procedure assumed a list of at least one element
  as input...
*/
char *flat_list(term l, char *fmt ){
 char *tmp ;

  switch (PL_type(PL_arg(l,1))) {  /* first argument of the list */
    case PL_TERM: {
      char *fc = functor_name(PL_arg(l,1)) ;
      if (strcmp(fc,".")==0) { /* It's a list */
        strcat(fmt, "%[");
        flat_list(PL_arg(l,1),fmt) ;  /* This is a nested list */
        strcat(fmt, "%]") ;
      } else {
        PL_fatal_error("Composed terms are not allowed inside an April list");
      } ;
      break ;
     };
    case PL_ATOM: {
      tmp =  PL_atom_value(PL_atomic(PL_arg(l,1)));
      strncat( fmt, tmp, 256 );
      break ;
    };
  };

  switch (PL_type(PL_arg(l,2))) { /* second argument of the list */
    case PL_TERM: {
      char *fc = functor_name(PL_arg(l,2)) ;
      if (strcmp(fc,".")==0) { /* It's a list */
        strcat( fmt, "%," );
        flat_list(PL_arg(l,2),fmt);
      } else {
        PL_fatal_error("Invalid term as
                                          second argument in a Prolog list..");
      } ;
      break ;
     };
    case PL_ATOM: {
      tmp =  PL_atom_value(PL_atomic(PL_arg(l,2)));
      if (*tmp != '[') {
        strcat( fmt, "%," );
  strncat( fmt, tmp, 256 );
}
break ;
```

```
    };
    default: {
      PL_fatal_error("Invalid term as second argument
                                                          in a Prolog list..");
    };
  };
} ;


/* ___________________________________________________ encode_msg */
/* this function transforms a valid (non_variable) PROLOG term into
a APRIL value of the appropiate type (now, numbers, symbols and
tuples and send it to an april process. Only lists of atoms allowed.
*/
int encode_msg(char *to, addrpo addrto, term m ) {

 switch (PL_type(m)) {

   case PL_INTEGER: {
     long num = PL_integer_value(PL_atomic(m));
     return(send_msg( to, addrto, "%d", num )) ;
   }
   case PL_STRING: {
     return(send_msg( to, addrto,
                                        "\"%s\"", PL_string_value(PL_atomic(m))));
   }
   case PL_ATOM: {
     char *atm = PL_atom_value(PL_atomic(m));
     if (*atm != '[') return(send_msg( to, addrto, "%s",atm));
     else return(send_msg( to, addrto, "%[%]"));
   }
   case PL_TERM: { /* the lists should be treated specially */
     char fmt[10000] ;  /* format containing the tuple */
     char *fc = functor_name(m) ;
     *fmt = '\0' ;
     if (strcmp(fc,".")==0) { /* It's a list */
        strcat(fmt, "%[");
        flat_list(m,fmt) ;  /* put the list in a flat string */
        strcat(fmt, "%]") ;
        return(send_msg( to, addrto, fmt )) ; /* List sending */
     } else {
        PL_fatal_error("Term type not allowed by April encoding.");
     } ;
     break ;
   }
   case PL_FLOAT: {
     return(send_msg( to, addrto,
                                              "%f", PL_float_value(PL_atomic(m))));
   }
   default: {
     PL_fatal_error("Term type not allowed for April encoding..") ;
     return -1 ;
   }
 }
 return -1 ;
}

/* ___________________________ __________________________ UnHookPl */
/* kept for debugging purposes */
```

```
void UnHookPl(addrpo addr)
{
fprintf(stdout,"Unhooking.. ") ;
};

/* ************************************************************ */
/* The following procedures constitutes the Prolog-April interface
*/
/*_______________________________________________________ ap_init */
/* initialises the plagent ports for listening and answering queries from other April agents.
*/
foreign_t pl_ap_init(term agentname, term portN )
{
 u_short PortNum = 0;
 u_short ServerPortNum = 5071; /* Default Name server port */
 long timeout = 4L;  /* server timeout period */

 PLAgentname = (char*) malloc(80);
 *PLAgentname = '\0';

 PL_dispatch_events = dispatch_events ;

 if (PL_is_atom(agentname)) {
   strncpy(PLAgentname,(char *)PL_atom_value(PL_atomic
                                                              (agentname)),80);
   /* It can no be more than 80 characters long */
 } else {
   strcpy(PLAgentname, PLDefAgentname );
   PL_warning("AP_INIT: Invalid Prolog Agent Name");
   PL_fail;
 };

 if (PL_is_int(portN)) {
   PortNum = (u_short) PL_integer_value(PL_atomic(portN)) ;
 } else {
   PL_warning("AP_INIT: Invalid TCP Port Number");
   PL_fail;
 };

 if ((CSocket = init_server((u_short) PortNum,
                                    PLAgentname, timeout, UnHookPl)) < 0){
   CloseAddressBook();
   PL_warning("AP_INIT: I couldn't start the Prolog Server socket");
   PL_fail;
 };

 /* Installing signals handlers */
 pl_signal(SIGQUIT,  sig_quit_agent) ;
 pl_signal(SIGINT,  sig_int) ; /* In case of Ctrl-C */
 pl_signal(SIGALRM, SWI_Refresh);

 /* Initialises the alarming system */
 alarm(3);

 PL_succeed;
};
```

```
/* ______________________________________________________  __ ap_send */
/* sends a message from the prolog agent to another april agent.
*/
foreign_t pl_ap_send(term to, term msg)
{
 char *dest;
 int result;
 addrpo addrto = LastQuerier ; /* Destination address */

 if ( PL_is_var(to) ) {
   PL_warning("ap_send/2: instantiation fault. first arg");
   PL_fail;
 };

 if ( PL_is_var(msg) ) {
   PL_warning("ap_send/2: instantiation fault. second arg");
   PL_fail;
 };

 if (addrto==NULL) {
   PL_warning("ap_send/2: no address avalaible");
   PL_fail;
 };
 dest = PL_atom_value(PL_atomic(to));
 result = encode_msg(dest,addrto,msg);

 if (result>=0) PL_succeed;
 else PL_fail;

};

/* ____________________________________________________________________ ap_receive */
/* receives a message send from an april agent. Observe that the term msg should be a variable. This version
can not perform pattern-matching. */
foreign_t pl_ap_receive(term from, term msg) {
 int msglen;
 char *fromhandle;
 addrpo replytoaddr = NULL ;
 term tmsg;
 char *msg_text;
 bktrk_buf buf;

 if (!(PL_is_var(msg))) {
   PL_warning("ap_receive/2: Variable
                                        required in second argument");
   PL_fail ;
 };

 /* PL_lock(from); PL_lock(msg); */

 fromhandle = (char*) malloc(80); /* Store the sender handle */
 *fromhandle = '\0';

 /* Get the next message from the TCP Channel */
```

```
again:
 if (do_wait(CSocket) && (replytoaddr =
     get_msg(PLAgentname,fromhandle,CSocket,"%#",&msglen,plxbuff))) {

    if (replytoaddr != NULL) {
      msg_text = plxbuff;
      decode_msg(msg,msg_text) ;
      LastQuerier = replytoaddr;

      /* Creating the new atoms and returning their
                values in the arguments */
      if (!(PL_unify_atomic(from, PL_new_atom(fromhandle)))) {
         free(fromhandle);
         /* PL_unlock(msg); PL_unlock(from); */
         PL_fail ;
      };

      /* Acknowledging the message */
      /* send_msg(fromhandle,replytoaddr,"%s","ok"); */

    }
 } else goto again;
 free(fromhandle);
 /* PL_unlock(msg); PL_unlock(from); */
 PL_succeed ;
};

/__________________________________________________________ ap_end */
/* cancels the communications capabilities of the prolog agent.
*/
foreign_t pl_ap_end()
{
 deregister();  /* Deregister with the name server */
 free(PLAgentname) ;  /* free the space for the agent name */
 CloseAddressBook();
 close(CSocket);
 PL_succeed ;
};
```

pl-prims.c

```
/*  pl-prims.c,v 1.11 1993/11/12 10:22:26 jan Exp

   Copyright (c) 1990 Jan Wielemaker. All rights reserved.
   See ../LICENCE to find out about your rights.
   jan@swi.psy.uva.nl

   Purpose: primitive built in

   NOTE: This is the code that was changed to integrate
         the APRIL-PROLOG interface. the change involved
         only one routine in this file and thus, we decide
         to include only the code of such routine (pl_halt).

                         Jacinto Davila. Imperial College. 1994.
*/

word
pl_halt()
{
 kill(getpid(), SIGQUIT);
 Halt(0);
 /*NOTREACHED*/
 fail;
}
```

talker.ap

```
/* talker.ap

  Program example for interchanging messages between an
  APRIL process and a PROLOG program. The apnameserver
  should be active.

  APRIL side.

  To start the program:
  1) run "april talker prologag"

  To stop the program:
  1) press Ctrl+C


  Last modified: 29 Jun 1994
*/

main(any[]?args)
{
 handle?plagent:= handle?args[1];
 number?i = 1;
 number?f = 100 ;
 while i<2 do {
   [hello,i] >> plagent; /* Send an integer */
   i+:=1;
 } ;
 i := 1;
 while true do {
   [symbol?F,symbol?S,any?L] => writef(stdout,"received [%s,%s,%p]\n",[F,S,L]);
   [gensym(),[gensym(),[gensym(),[i]]]] >> plagent;
   i+:= 1;
 }
}
```

```
/* talker.pl

  Program example for interchanging messages between an
  APRIL and PROLOG. The apnameserver should be active.

  PROLOG side.

  To start the program:
  1) run plagent.
  2) load this program: "[talker]."
  3) type the query-call "agent."
  4) run the april side

  To stop the program:
  1) press Ctrl+C

  Last modified: 29 Jun 1994
*/

agent :-
 ap_init(prologag,0),
 initial_chat(F,1),
 sending(F),
 ap_end.

initial_chat(_,0).
initial_chat(F,N) :- /* Needed for establishing the conection */
 ap_receive(F,M),
 write(M),nl,
 NN is N - 1,
 initial_chat(F,NN).

sending(F) :-
 ap_send(F,[hello,myfriend,[]]),
 ap_receive(F,M),
 write(M),nl,
 sending(F).
 write('send more messages?'),
 read(Ans),
 ((Ans = 'y') -> sending(F) ; true).
```

Logic Programming Agents

APPENDIX D. DIALOX DISPLAYS.

In this appendix we include some pictures take from the DIALOX window displaying the *world* during the simulations. The first picture shows one robot (`ja`) travelling its experimental route in the warehouse. The remaining pictures were taken from the animation of four robots interacting in the `patio` world.