A New Logic Architecture for Intelligent Agents

Gabriela Şerban

Department of Computer Science
"Babeş-Bolyai" University
1, M. Kogalniceanu Street, Cluj-Napoca, Romania
tel: +40.264.405325, fax: +40.264.191.960
gabis@cs.ubbcluj.ro
http://www.cs.ubbcluj.ro/~gabis

Abstract. It is well-known that one of the concrete architectures for intelligent agents is the logic one. In our opinion, the symbolic representations for the intelligent behavior are very important, and the logic approach is elegant and has a pure semantic. The aim of this paper is to present a new logic architecture for intelligent agents (LASG - a Logic Architecture based on Stacks of Goals) [1]. This architecture combines the traditional logic architecture with a planning architecture [8]. The advantages of the proposed architecture are shown in the paper. As a case study we illustrate the use of the LASG algorithm on a maze searching problem.

Keywords: intelligent agents, logic, declarative programming.

1 Introduction

The logic approach is a topic of Symbolic Artificial Intelligence and has its own importance in the field of intelligent agents, even if it is well-known the controversy between the traditional approach and the intelligent calculus in the field of Artificial Intelligence.

On the other hand, knowledge representation is one of the most important subareas of Artificial Intelligence and logic programming has a major application in this field [7].

Moreover, the only intelligence requirement we generally make for the agents is that [6] they can make an acceptable decision about what action to perform next in their environment, in time for this decision to be useful. Other requirements for intelligence will be determined by the domain in which the agent is applied: not all agents will need to be capable of learning, for example.

In such situations, a logic architecture is very appropriate, and offers, in our opinion, a simple and elegant representation for the agent's environment and desired behavior.

According to the traditional approach [6], the symbolic representations are *logical formulae*, and the syntactic manipulation corresponds to *logical deduction*, or theorem proving.

In such a logic approach, the agent could be considered as a *theorem prover* (if ϕ is a theory that explains how an intelligent agent should behave, the system might generate a sequence of steps - actions - that leads to ϕ - in fact a proof for ϕ).

However, some disadvantages of the logic approach are:

- the computational complexity of a theorem proving process raises the problem if the agents represented this way can really operate in time-restricted environments;
- the process of decision making in such logic architectures is based on the assumption that the environment does not change its structure, essentially, during the decision process (a decision that is correct at the beginning of the process, will be correct at the end if it, too);
- the problem of representation and reasoning in complex and dynamic environments is, an open problem, as well.

It is known that logic programming theories of reasoning about actions and planning implementations are interconnected [5].

On the other hand Brogi shows in [4] that the complex planning strategies find natural logic-based formulations and efficient implementations in the framework of deductive database languages.

2 A Logic Architecture Based on Stacks of Goals (LASG)

In this section we propose an approach that shows how to use a STRIPS-like planning strategy to describe a logic architecture for intelligent agents.

We will consider, in the following, the case of an agent which goal is to solve a given problem (pass from an initial to a final state), based on a set of operators (rules) that could be applied on a given moment [9].

In a LASG architecture, we will use the *declarative representation* of the knowledge.

Let L be a set of sentences from the first-order logic, and $D = P(L)$ the set of *L-databases* (the set of sets of *L-formulae*). In the model that we propose, the *internal state* of the agent will be given by an element from D (for simplicity, we will consider it as a formula in a conjunctive normal form).

2.1 Case Study: Searching a maze

In this section we will consider the following problem: we have a maze that has a rectangular form; in some positions there are obstacles; a robotic agent starts in a given state (the initial state) and it tries to reach a final (goal) state, avoiding the obstacles; in a certain position on the maze the agent could move in four directions: north, south, east, west (there are four possible actions). We will assume that the dimensions of the maze are known: **M** is the number of rows, **N** is the number of columns.

In the example that we choose, the environment (the maze) is not dynamic (it suffers no modifications after the agent's actions). However, this assumption is not essential, it has no significant influence on the agent's behavior.

We consider that:

- a position on the maze is identified by a pair (X, Y) (the line, respectively the column);
- the left up corner of the maze is marked as the position (1, 1).

The four actions that the robotic agent could execute are the following:

NORTH(X, Y) - from the position (X, Y) the robot moves in the north direction. The positions (X, Y) and (X-1, Y) must be into the maze and must not contain obstacles.

EAST(X, Y) - from the position (X, Y) the robot moves in the east direction. The positions (X, Y) and (X, Y+1) must be into the maze and must not contain obstacles.

SOUTH(X, Y) - from the position (X, Y) the robot moves in the south direction. The positions (X, Y) and (X+1, Y) must be into the maze and must not contain obstacles.

WEST(X, Y) - from the position (X, Y) the robot moves in the west direction. The positions (X, Y) and (X, Y-1) must be into the maze and must not contain obstacles.

In order to specify both the conditions in which the operations hold and the results of executing the operations, we will use the following predicates:

FREE(X, Y) - the position (X, Y) is *free* (does not contain an obstacle).

IN(X, Y) - the robotic agent is in the position (X, Y).

VALID(X, Y, M, N) - the position (X, Y) is *valid* in the given maze (is into the maze).

POSSIBLE(X, Y, M, N) - the position (X, Y) is *free* and *valid*.

We notice that:

$$POSSIBLE(X,Y,M,N) <=> FREE(X,Y)\, and\, VALID(X,Y,M,N) \quad (1)$$

In such a logic representation, there are valid some logic declarations. For example,

$$not\, FREE(X,Y)\, and\, VALID(X,Y,M,N) \rightarrow not\, IN(X,Y) \quad (2)$$

$$not\, FREE(X,Y)\, or\, not\, VALID(X,Y,M,N) \rightarrow not\, IN(X,Y) \quad (3)$$

As in a planning system, in a LASG architecture must be realized the following functions:

1. how to detect the best rule to apply, based on the best (possible heuristic) information available;

2. how to apply the chosen rule in order to compute the new problem's state;
3. how to detect if a solution was found;
4. how to detect if the system was blocked, in order to abandon the blocked paths and the system's effort to be directed in most interesting directions.

2.2 How to select the rules

The most used technique for choosing the appropriate rules is to determine a set of differences between the desired final state and the current state, and then to identify the relevant rules for reducing the differences. If more rules are identified, a variety of heuristic information could be exploited in order to chose the rule to be applied. This technique is based on the means-end analysis.

2.3 How to apply the rules

A possibility to apply the rules is to describe for each possible action the changes that it brings to the state's description. Moreover, some declarations are needed, in order to state that the rest of the description remains the same. A solution for this problem could be to describe a state as a set of predicates representing the facts that are valid in the given state. Each state is explicitly represented as an argument of the predicates. For example, assume that the current state S is described by

$$POSSIBLE(X, Y, M, N, S)\ and\ IN(X, Y, S)\ and \quad (4)$$
$$POSSIBLE(X-1, Y, M, N, S)$$

and the rule that describes the operator **NORTH(X, Y)** will be

$$POSSIBLE(X, Y, M, N, S)\ and\ IN(X, Y, S)\ and \quad (5)$$
$$POSSIBLE(X-1, Y, M, N, S) \rightarrow IN(X-1, Y, DO(NORTH(X, Y), S))$$

In the above equation DO is a function which specifies the state that results after applying a given action in a given state.

For assuring the correctness of the deduction mechanisms, it will be necessary a set of rules to describe those components of the states that are not affected by the operators (the so named *frame axioms*). The advantage of this approach is that a unique mechanism, the resolution, could realize all the operations needed to describe the states. However, the disadvantage is the big number of axioms, if the states' descriptions are complex.

In the architecture that we propose in this section, the number of explicit frame axioms that should be used is not so big.

Each operator will be described by a list of new predicates that are made true by the operator and a list of old predicates that are made false by the operator. The two lists are named ADD, respectively DELETE. Moreover, for each operator a third list is specific, PRECONDITION, which contains all the predicates that must be true in order to apply the operator. The frame axioms

are implicitly specified in LASG. Each predicate not included in the ADD or DELETE lists of an operator is not affected by that operator.

The LASG operators that correspond to the operations presented above are shown in Figure 1. For simplicity, we numbered the four moving possibilities of the robotic agent from a given position (X, Y) as follows: 1- North, 2 - East, 3 - South, 4 - West. We also consider two vectors **dx = (-1, 0, 1, 0)** and **dy = (0, 1, 0, -1)** which gives the moves relative on line and column corresponding to the four actions. Thus, the operator corresponding to the **k**-th move from the position (X, Y) could be described as below:

O(X, Y, M, N, K)
P: POSSIBLE(X, Y, M, N) and IN(X, Y) and POSSIBLE(X+dx[k], Y+ dy[k], M, N)
A: IN(X+dx[k], Y+ dy[k])
D: IN(X, Y)

Fig. 1. The operators' description

The application of an operator **O** on a state **S** (given as a logic formula ϕ) means that the predicates from the ADD list of the operator should be added in ϕ. On the other hand, the return to the state before applying the operator **O** (the backtracking) means that the predicates from the DELETE list of the operator should be deleted from ϕ.

3 The LASG algorithm

The idea of the algorithm is to use a stack of goals (a unique stack that contains both goals and operators proposed for solving those goals). The problem solver is also based on a database that describes the current situation (state) and o set of operators described by the PRECONDITION, ADD and DELETE lists. For illustration, we will apply this method on the example shown in Figure 3 (the number of rows M is 4 and the number of columns N is 3).

At the beginning of the problem solving process, the stack of goals contains

IN(1, 3)

We have to find an operator which makes true the predicate from the top of the stack (in other words, the predicate **IN(1, 3)** must appear in the ADD list of the operator). We find (by variables' bounding) two possibilities: the operator **O(1, 2, 4, 3, 3)** and **O(2, 3, 4, 3, 1)**. We separate the initial stack into two stacks, we place in the top of the corresponding stack (instead of **IN(1, 3)**) the operator that was found and the predicates from it's PRECONDITION list.

IN(2, 3)	IN(1, 2)
FREE(2, 3)	FREE(1, 2)
VALID(2, 3, 4, 3)	VALID(1, 2, 4, 3)

are implicitly specified in LASG. Each predicate not included in the ADD or DELETE lists of an operator is not affected by that operator.

The LASG operators that correspond to the operations presented above are shown in Figure 1. For simplicity, we numbered the four moving possibilities of the robotic agent from a given position (X, Y) as follows: 1- North, 2 - East, 3 - South, 4 - West. We also consider two vectors **dx = (-1, 0, 1, 0)** and **dy = (0, 1, 0, -1)** which gives the moves relative on line and column corresponding to the four actions. Thus, the operator corresponding to the **k**-th move from the position (X, Y) could be described as below:

O(X, Y, M, N, K)
P: POSSIBLE(X, Y, M, N) and IN(X, Y) and POSSIBLE(X+dx[k], Y+ dy[k], M, N)
A: IN(X+dx[k], Y+ dy[k])
D: IN(X, Y)

Fig. 1. The operators' description

The application of an operator **O** on a state **S** (given as a logic formula ϕ) means that the predicates from the ADD list of the operator should be added in ϕ. On the other hand, the return to the state before applying the operator **O** (the backtracking) means that the predicates from the DELETE list of the operator should be deleted from ϕ.

3 The LASG algorithm

The idea of the algorithm is to use a stack of goals (a unique stack that contains both goals and operators proposed for solving those goals). The problem solver is also based on a database that describes the current situation (state) and o set of operators described by the PRECONDITION, ADD and DELETE lists. For illustration, we will apply this method on the example shown in Figure 3 (the number of rows M is 4 and the number of columns N is 3).

At the beginning of the problem solving process, the stack of goals contains

IN(1, 3)

We have to find an operator which makes true the predicate from the top of the stack (in other words, the predicate **IN(1, 3)** must appear in the ADD list of the operator). We find (by variables' bounding) two possibilities: the operator **O(1, 2, 4, 3, 3)** and **O(2, 3, 4, 3, 1)**. We separate the initial stack into two stacks, we place in the top of the corresponding stack (instead of **IN(1, 3)**) the operator that was found and the predicates from it's PRECONDITION list.

IN(2, 3)	IN(1, 2)
FREE(2, 3)	FREE(1, 2)
VALID(2, 3, 4, 3)	VALID(1, 2, 4, 3)

O(2, 3, 1, 4, 3) (1) **O(1, 2, 3, 4, 3)** (2)

For each stack, we repeat the operations described above with the predicate from the top of the stack. At a given moment, there are four possibilities:

- in the top of the stack is an operator; in this case we remove it from the top, and we retain the operator as part of the problem's solution;
- the predicate from the top of the stack is true; in this case we remove it from the top;
- the predicate from the top of the stack is false; in this case we have to find operators that make the predicate true; we ramify the stack; we add the operators (with their preconditions) in the stack;
- the predicate from the top of the stack can not be satisfied, which means that the system was blocked; in this case we have to abandon the current path, because it will not lead to a solution.

The operation is repeated until the stack became empty (a solution of the problem was found), or until all the possibilities were blocked (in this case the problem solving fails).

If we continue to apply the algorithm on our example, two solutions will be reported:

1. O(4, 1, 4, 3, 1)
2. O(3, 1, 4, 3, 1)
3. O(2, 1, 4, 3, 2)
4. O(2, 2, 4, 3, 2)
5. O(2, 3, 4, 3, 1)

1. O(4, 1, 4, 3, 1)
2. O(3, 1, 4, 3, 1)
3. O(2, 1, 4, 3, 2)
4. O(2, 2, 4, 3, 1)
5. O(1, 2, 4, 3, 2)

In fact, the algorithm consists in a process of **backward reasoning** (we starts from the final state), method known in the literature as a **goal directed reasoning**.

We assume that are given:

- *SI* (the initial state for the agent);
- *SF*(the final state that the agent tries to reach) - there could be a set of final states;
- a set of *operators* $O = \{O_1, O_2, \cdots O_k\}$ that are available to the problem solving agent. For each operator O_i the agent knows the three lists: PRE-CONDITION, ADD and DELETE.

The agent's goal is to reach the final state *SF*, starting from the initial state *SI*, keeping a history H of the visited states ($H = \{S_1, S_2, \cdots S_m\}$, where $S_1 = SI$ and $S_1 = SF$), or of the applied operators ($H = \{O_{i,1}, O_{i,2}, \cdots O_{i,m-1}\}$). In the case that the problem has no solution, H will be empty.

The algorithm which determines a solution of the problem (if exists a solution) is described in Figure 2.

The non-determinism of the step 4 from the above described algorithm has to be implemented as a kind of search procedure (a limited depth-first search).

1. – we create a stack of goals S (the solution stack) that initially contains the predicates that should be satisfied in the final state SF. In other words, if the final state could be written as conjunction of logic sentences $SF = \{\phi_1 \, and \, \phi_2 \, and \, \cdots \, and \, \phi_n\}$, then $S = \{\phi_1, \phi_2, \cdots \phi_n\}$;
   – SC (the current state):= SI (the initial state);
   – H:=empty;
2. **If** S is empty and the final state SF was reached, **then** the algorithm stops and the final solution is reported; **else go to** step **3**;
3. **3.1 If** the top of the stack contains an operator O_i, on add the operator in H, on remove the top of stack, on recalculate the current state SC at which on add the predicates from the A list of the operator O_i; **go to** step **2**; **else go to** step **3.2**;

   3.2 On choose the predicate from the top of the stack (ϕ_1). **If** ϕ_1 is satisfied in SC, we remove it from the top of the stack; **go to** step **2**; **else go to** step **4**;
4. We look for the operator O_i (the operators, if are several) that makes ϕ_1 true. If there are several operators $O_{i,1}, O_{i,2}, \cdots O_{i,s}$, on ramify the solution (on obtain s stacks)

   for j=1,s (for each of the s stacks)

   - we add on the top of the stack S_j the predicates from the PRECONDITION list of the operator $O_{i,j}$; **go to** step **3**.

Fig. 2. The algorithm to determine a solution in a LASG architecture

4 Comparison between the LASG architecture and the traditional logic architecture

The Logic Architecture based on a stack of goals improves the traditional logic architecture for intelligent agents in the following directions:

- comparatively to the traditional logic approach, where the number of the frame axioms that should be saved in order to make a correct inference is large, the LASG architecture reduces this number. In other words, the space complexity is reduced;
- because of a limited Depth First search, the time complexity is reduced, too;
- the representation is very simple and elegant.

5 Experiment

Because the above described architecture is based on logic and because the algorithm described in Figure 2 needs backtracking for finding all solutions, the implementation was made in Visual Prolog. It is well-known that the declarative programming languages (as Prolog) have a built-in control mechanism which allows finding all the solutions of a problem.

We have to say that the stack (stacks) of goals that we have to create for applying the algorithm (Figure 2) are retained implicitly by the control strategy of Prolog (a mechanism which allows backtracking).

For illustrating the algorithm we consider the simple environment shown in Figure 3.

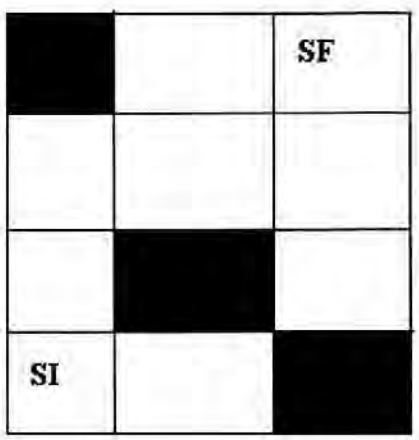

Fig. 3. The agent's environment

The positions filled with black on the maze contains obstacles.

We implemented a Prolog program, which basic non-deterministic predicate is **path(Xi, Yi, Xf, Yf, M, N, L)**, having the flux model **(i, i, i, i, i, i, o)**, and the following signification for the arguments:

- **Xi, Yi** - the coordinates (line and column) of the initial position (the starting position for the agent);
- **Xf, Yf** - the coordinates (line and column) of the final position (the position that the agent tries to reach);
- **M, N** - number of lines and respectively columns of the maze;
- **L** - the list of positions visited by the agent for reaching **SF** starting from **SI** (if the problem has no solution, the list will be empty).

For solving the problem, we considered an LASG architecture and we applied the algorithm described in Figure 2.

The goal has the form

goal: **path(4, 1, 1, 3, 3, 3, L)**

and the solutions are two:

L=[[4, 1], [3, 1], [2, 1], [2, 2], [2, 3], [1, 3]]
L=[[4, 1], [3, 1], [2, 1], [2, 2], [1, 2], [1, 3]]

We mention that we consider different rectangular environments and the results are good.

6 Conclusions and Future work

In a logic based architecture, the intelligent behavior is generated in the system by a symbolic representation of the environment and of the behavior and by a symbolic manipulation of this representation.

We intend to make a better evaluation for the LASG architecture and to compare our approach with other works, such as, the traditional planning strategies (STRIPS based), event and situation calculus, and the dynamic logic programming approach [3].

Further work could be done in the following directions:

- in which way we could use some heuristic information in order to reduce the computational complexity of the deduction process;
- in which way we can combine the traditional logic architecture with other planning architectures (TWEAK, hierarchical planning architectures [8]);
- in wich way the system is able to deal with dynamic environments.

References

1. Serban, G.: Development methods for Intelligent Systems. Ph.D. Thesis. "Babes-Bolyai" University of Cluj-Napoca, Romania (2003)
2. Weiss, G.: Multiagent systems - A Modern Approach to Distributed Artificial Intelligence. The MIT Press, Cambridge, Massachusetts, London (1999)
3. Alferes, J.J., Leite, J.A, Pereira, L.M., Przymusinska, H., Przymusinski, T.C: Dynamic Logic Programming (1998)
4. Brogi, A., Subrahmanian, V.S and Zaniolo, C.: The Logic of Totally and Partially Ordered Plans: A Deductive Database Approach. Annals of Mathematics and Artificial Intelligence, 19(1,2) (1997) 27–58
5. Baral, C.: Relating Logic Programming Theories of Actions and Partial Order Planning. Annals of Math and AI, vol. 21 (1997) Nos 2-4
6. Wooldridge, M.: Agent-Based Software Engineering. Mitsubishi Electric Digital Library Group, London (1997)
7. Baral, C. and Gelfond, M.: Logic programming and knowledge representation. Journal of Logic Programming 19,20 (1994) 73–148
8. Rich, E., Knight, K.: Artificial Intelligence. Mc Graw Hill, New York (1991)
9. Winston, P.: Artificial Intelligence. Addison Wesley, Reading, MA (1984) 2nd ed
10. Reiter, R.:On closed world databases. In Gallaire, H. and Minker, J. (eds.): Logic and DataBases, Plenum Press, New York, (1978) 119–140