Metalevel Programming in Robotics: Some Issues

A. Kumar N. and N. Parameswaran
Indian Institute of Technology
Madras, India

ABSTRACT. Computing in robotics has two important requirements: efficiency and flexibility. Algorithms for robot actions are implemented usually in procedural languages such as VAL and AL. But, since their excessive bindings create inflexible structures of computation, we propose in this paper, Logic Programming as a more suitable language for robot programming due to its non-determinism, declarative nature, and provision for metalevel programming. Logic Programming, however, results in inefficient computations. As a solution to this problem, we discuss a framework in which controls can be described to improve efficiency. We have divided controls into: (i) in-code and (ii) metalevel and discussed them with reference to selection of rules and dataflow. We have illustrated the merit of Logic Programming by modelling the motion of a robot from one point to another avoiding obstacles.

1. Introduction

Computing in robotics requires both efficiency and flexibility. Large scale real time computation, both symbolic and numeric, is necessary for even apparently simple robot movements. Robot's actions must be as time-efficient as possible at the end-effector and actuator levels[1]. At the same time, planning, backtracking, consulting knowledge about the world etc., requires flexibility.

The world of the robot is dynamically changing, as in the case of telerobots, and the change in the world must be continually noted. Accordingly, the planning of the robot at the object and the objective levels has to be altered / readjusted. This is a strong case in favour of flexible computation for robotics at the object and the objective levels.

In robotic computation, a third requirement is effective man machine communication [2]. This requires that the programming environment be as conducive to natural input and the language be as readable as possible.

Traditionally, languages for robots are procedural, such as VAL[3], AL [4] and RAPT [5]. They provide for efficient computation, but are not flexible or easily modifiable.

We propose that Logic Programming is more suitable for robot programming due to its non-determinism, declarative nature, and provision for metalevel programming. Description of the world is most conveniently done in the declarative semantics of Logic Programming. Dynamic change in the world can be easily incorporated by the addition / deletion of logical assertions in the program. The fact that visual world can be best described declaratively has been exploited in [6] for declarative graphics.

However, a major problem in using Logic for computation in robotics is its inefficiency. Typically, a Logic Programming language such as Prolog follows the depth-first strategy with chronological backtracking and a few control features such as cut to improve efficiency [7]. However, recently, many features have been introduced in Prolog, to improve efficiency, such as intelligent backtracking[8,9], annotated variables[10], and metalevel programming[11,12].

Of greatest relevance to robotic computation is the last feature viz., metalevel programming. Here, the specification of control strategy for the object level program (OLP) is expressed seperately, at a different level called metalevel. It permits one to intervene in the interpretation of the object level programs to define new strategies of control. It also allows one to specify one's own interpreter.

Since metalevel description is kept entirely seperated from logic level, the basic procedures and world description at the object level are left entirely untouched by efficiency considerations. Object level programs are still as declarative and flexible as ever. An

additional element of flexibility is that the control strategy for any computation can be changed to suit the needs of the current goal and world.

2. In-code and Metalevel Controls

For robot programming, we present control at two levels: (a) in-code, (b) metalevel; and discuss them with respect to two types of controls : (i) selection control (ii) data flow control.

In-code control refers to control expressed textually along with the OLP clauses and the syntax and the semantics are defined as part of the OLP language. At this level we provide:

* efficient constructs which are procedural in nature.
* liberal bindings of as much dataflow as possible, using the idea of locations.

However, at the global level, we propose largely a declarative language.

Metalevel control refers to control over selection of rules and is textually separated from OLP. At this level, control is expressed in the form of rules (different representations are possible) so that it is easy to modify when required.

This strategy of splitting the controls has the following consequences on computation :

1. Locally, the computation is expressed as efficiently as possible at the cost of flexibility. But, since it is only local, the resulting inflexibility may be acceptable as, in the worst case, it can always be replaced when modifications are required.

2. Globally, the structure of computation is kept as flexible as possible, at the cost of efficiency i.e.,the programs need not be radically re-altered when the world or the input changes for a robot. Efficiency is improved by carefully choosing metalevel rules. Hence, readability and modifiability of programs are preserved, which is very important for robotic computation.

The following are the control features we have introduced to improve the efficiency:

CONTROL FLOW:

Incode: The scope of in-code controls is restricted to individual clauses and they are directed to specific predicates.

(i) Forward jump F(X): Let F(X) be a predicate in the body of a clause C. The execution of this predicate results in the transfer of control to the predicate in the clause C, to which X is instantiated, skipping the predicates between F(X) and X in C. F(X) fails when X is uninstantiated or when backtracking.

(ii) Backward Cut !(X): Let !(X) be a predicate in the body of a clause C. !(X) succeds in the forward direction trivially. While backtracking, if !(X) is encountered, control is transferred to the predicate X occuring within the clause C.

Metalevel: Several control features defined in Metalog[11] actually fall under this: CHOOSECLAUSE, INHIBCLAUSE, INHIBACK, FACTOR and BACKFAIL. In addition, we propose OR-parallelism as a control structure on the set of modules. This makes it possible to control parallelism by controlling the number of processes. An important restriction we have placed here is that these controls are expressed with respect to clauses and not individual predicates. Thus the control here will tell which clause to use but not which predicate in a clause. Controls of this type affect the top-to-bottom selection of clauses for interpretation within a program. The scope of all the metalevel declarations is the entire program.

(i) OR Parallelism

(PARCHOOSECLAUSE literal i[1],i[2],....,i[n] clausebody) :- condition.

The execution of this predicate results in the simultaneous selection of n clauses for interpretation in OR parallellism where the clause heads are unifiable with literal i[j] denotes the j-th clause among the m clauses whose heads unify with literal. When n = 1, and i[1] = 1, PARCHOOSECLAUSE reduces to CHOOSECLAUSE of Metalog[11].

DATAFLOW:

Incode: In Prolog, data flow within a clause occurs through side-effects in the sense of [13]. We introduce new predicates create and assign to let data flow both in the forward and backward (on backtracking) directions. This unlimited data flow within a clause enhances efficiency considerably.

(i) create(L1,L2,....,Ln): Let C be the clause in which the predicate create occurs.

Execution of this predicate in the forward direction results in the creation of locations Li, i = 1,...,n. These locations can be used to store values using the assign predicate described below. Create fails during backtrack.

(ii) assign(L,expression): Execution of this predicate results in storing the value of the expression in L, where L is a location. It fails when L is not created, and during backtrack.

Metalevel: At this level the rules define global data locations for the program. Data flow can occur across the clauses through parameters or global locations when it is convenient. Distributed computing for coordinated activities in a colony of robots can be achieved by appropriate metalevel descriptions of control and data flow.

Global locations

GLOBAL L1,L2,...,Ln.

This declaration at metalevel defines n locations L1,...,Ln that can be used as locations to store data and use them later in the program; they are visible over the entire program.

3. ILLUSTRATION

In this section, we address the problem of a robot moving in a world, avoiding obstacles. For simplicity, the world is divided into a number of squares. The obstacles are distributed in the world as shown in Figure 1. The start position of the robot is assumed to be the square (1,1), denoted by p(1,1). The target is the square p(6,4). We restrict the motions of the robot to horizontal and vertical directions only.

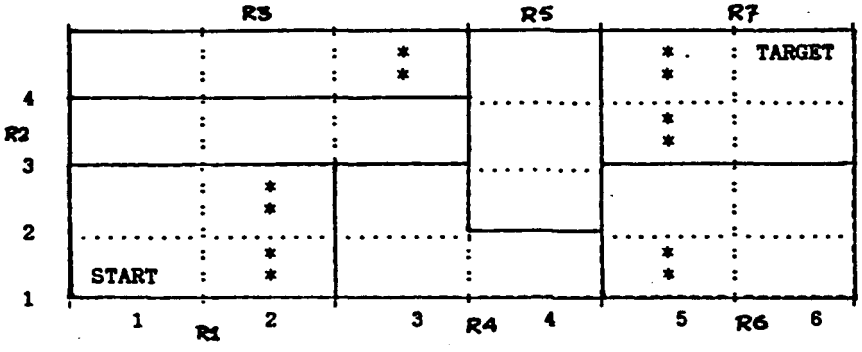

Figure 1: The robot world.

The solution is provided at two levels: object level, and metalevel. At the object level, the mechanical actions and the simple heuristics the robot employs on encountering obstacles are described. At the metalevel, knowledge about the distribution of the obstacles in the world and the mechanisms for supplying information about local regions in the world are described in order to guide the robot towards its target. The knowledge encoded in the object level description is complete in the sense that the knowledge is sufficient, in principle, for the robot to reach any given target, by exhaustive search. Metalevel knowledge is incomplete in this sense, but it can greatly aid in reducing the exhaustive search, and thus improving the efficiency.

At the object level, we have the following heuristic to guide the robot: Let the current position of the robot be (Cx,Cy) and the target location (Tx,Ty), where Tx >= Cx and Ty >= Cy. If Tx = Cx and Ty = Cy, then the robot has reached its target, and therefore it halts; else, the robot moves to the square (Cx + 1, Cy) if Tx > Cx and (Cx + 1, Cy) is obstacle-free, or to the square (Cx, Cy + 1) if Ty > Cy and (Cx, Cy + 1) is obstacle-free. Otherwise, it consults the expert in the current region.

At the metalevel, the squares are grouped into regions. For example, p(1,1), p(1,2), p(2,1) and p(2,2) form R1.(see Figure 1.) Each region has an expert who has the knowledge of the obstacles present in that domain. The expert has the ability to suggest a way out to each of its neighbouring regions, when consulted. For example, when consulted, R1 shows the way out to R2 via the square p(1,3). At the metalevel, we also have the following control heuristic to avoid infinite paths: In any region Ri,the robot ignores the advice of the expert of this region, if the advice leads the robot back to any of the previously travelled region Rj. In case of failure of a solution suggested by an expert, the robot backtracks to any alternative solutions that the expert may have suggested. If all the suggestions of the

current expert fail to provide an exit, the robot will take up any untried alternative suggestions of the earlier expert; and so on.

The robot initially traces the following squares: p(1,1), p(1,2), p(1,3), p(2,3), p(3,3), p(4,3), p(4,4). At this point, the robot is not able to move further on its own, and so it consults R5. R5 suggests the way out to R2, R4 and R6. The way out to R2 is ignored. If the robot accepts the way out to R4, via p(4,1) or p(3,2), the robot finds itself once again at a dead end. However, on backtracking, it tries the third solution suggested by R5, thus entering R6, and the target eventually.

```
/* Robot: Object level program */
/* Initialise stack for local expert and start */

toptrav( p(*X,*Y), p(*X1,*Y1) ) :-  create ( *list ),
                                    assign (*list, [] ),
                                    trav ( p(*X,*Y), p(*X1,*Y1) ).

trav( p(*X,*Y), p(*X,*Y) ) :- !
trav( p(*X,*Y), p(*X1,*Y1) ) :-
                        advance(p(*X,*Y),p(*X1,*Y1),p(*X2,*Y2)),
                        trav( p(*X2,*Y2), p(*X1,*Y1) ).

advance( p(*X,*Y), p(*X1,*Y1), p(*X2,*Y2) ) :-  *X1 > *X,
         *X2 is *X + 1,
         checkobst( *X2,*Y ), /* A built-in procedure that checks
                                              whether the square (X2,Y) has an obstacle */
         in_region( p(*X2,*Y), *R ), /* Finds out the region to which (X2,Y) belongs */
         test_append( *R, *list ), /* Checks whether *R has already been traversed */
         move( p(*X2,*Y) ), /* A built-in procedure that enables the robot move by a square */
         F( ! ),  move( p(*X,*Y) ),  !( move ).  /* The robot retraces its path
                                                    in  case  of failure */

advance( p(*X,*Y), p(*X1,*Y1), p(*X2,*Y2) ) :-  *Y1 > *Y,
                        *Y2 is *Y + 1,
                        checkobst( *X,*Y2 ),
                        in_region( p(*X,*Y2), *R ),
                        test_append( *R, *list ),
                        move( p(*X,*Y2) ),
                        F( ! ), move( p(*X,*Y) ), !( move ).

advance( p(*X,*Y), p(*X1,*Y1), p(*X2,*Y2) ) :-
                        in_region( p(*X,*Y), *R1),
                        consult(  *R1,  p( *X,*Y ),  p( *X2,*Y2 )).  /* If met with a dead end,
                                                                        consult local expert */

/* To check whether R has been already consulted */

test_append( *R, *list ) :- member( *R, *list ), !.
test_append( *R, *list ) :- append( *R, *list, *list1 ),
                            assign( *list, *list1 ).
```

The metalevel description defines the local experts, and the position of obstacles. Associated with each expert is its knowledge regarding the position of squares which the robot should try to reach in order to avoid infinite paths.

```
/* Local Experts: Metalevel program */

consult( *R1, p(*X,*Y), p(*X2,*Y2) ) :- out( *R1, *Z ),
                        member( p(*X2,*Y2), *Z ),
                        in_region( p( *X2, *Y2 ), *R2 ),
                        not( member( *R2, *list ) ),
                        advance(p(*X,*Y),p(*X2,*Y2),p(*X2,*Y2)).

region( R1, [ p(1,1), p(1,2), p(2,1), p(2,2) ] ). /* Definition of region R1 */
region( R2, [ p(1,3), p(2,3), p(3,3) ] ).
region( R4, [ p(3,1), p(3,2), p(4,1) ] ).
                  ..
                  ..
out( R5, R2, [ p(3,3) ] ).
out( R5, R4, [ p(3,2), p(4,1) ] ).
out( R5, R6, [ p(5,2) ] ).
                  ..
                  ..
```

```
/* Description of location of obstacles */
obst( 2,1 ).        obst( 2,2 ).        obst( 3,4 ).
obst( 5,1 ).        obst( 5,3 ).        obst( 5,4 ).
```

4. Conclusion

In this paper, we have discussed the merits of Logic programming for robotic computations. Towards improving the efficiency of computation, while retaining the declarative nature of programming, we have split the control into two levels: in-code and metalevel. We have proposed several constructs at both these levels, that improve efficiency. We have illustrated the elegance of metalevel logic programming and the usage of the predicates we have introduced, on a simple robot problem.

REFERENCES:

1. Kogan Page,Logic and Programming., Robot Technology series Vol 5., London 1983.

2. Ambler A.P., 'Languages for programming robots', in Robotics and Artificial Intelligence, NATO ASI series 1984.

3. 'Users guide to VAL', Unimation Inc.

4. Goldman R., Mujtaba S.M.,'The AI users manual', Stan-CS-79-718, Stanford Univ.,Jan 1979.

5. Popplestone R.J.,Ambler A.P.,Bellos T.M., 'An interpreter for a Language for describing assemblies', AI, vol.14, 1980, pp79-107.

6. Richard Helm and Kim Marriott, 'Declarative Graphics', Proc. of Third Intl. Conf. on Logic Programming, July 1986 (Lecture Notes in Comp. Science 225).

7. W. F. Clocksin and C. S. Mellish, 'Programming in Prolog' Springer Verlag, 2nd. edition 1984.

8. M. Bruynooghe and L. M. Pereira, 'Deduction revision by intelligent backtracking', in 'Implementations of Prolog' ed. J. A. Campbell, Ellis Horwood Ltd., 1984.

9. P. T. Cox, 'Finding backtrack points for intelligent backtracking' in 'Implementations of Prolog' ed. J. A. Campbell, Ellis Horwood Ltd., 1984.

10. Clark K. L., and McGabe F. G., 'IC-PROLOG : Language features' Proc. Logic Programming Workshop, Debrecen, July 1980.

11. Mehmet Dincbas et al., 'Metacontrol of Logic programs in Metalog' in Proc. of the Intl. Conf. on Fifth Generation Computer Systems, ICOT, 1984.

12. Colmerauer A., 'PROLOG-II:Manuel de reference et modele theorique', GIA, Faculte des Sciences de Luminy, Mars 1982 .

13. Terrence W. Pratt, 'Programming Languages: Design and Implementation', Prentice Hall Inc., 2nd edition 1984.