ADVIS 2002

The Implementation of a Robotic Replanning Framework

Şule Yıldırım and Turhan Tunalı
Ege Univ. International Computer Institute, 35100, Bornova İzmir
{yildirim,tunali}@ube.ege.edu.tr

Abstract. In this study, the implementation of a previously proposed robotic replanning framework is presented. The proposed framework integrates a high level replanning paradigm into a three layer robotic architecture. There has been a great deal of studies on managing unexpected events at lower two levels of three layer architectures but doing replanning at highest level still needs investigation. Supporting replanning with real-time vision feedback from working environment and integrating a learning mechanism as a basis increases the success ratio.

1 Introduction

The modern robotic control architectures have shown an evolution from SPA (Sense Plan Act) to Subsumption to Three Tier architectures. In Subsumption high level layers interface with lower level ones by suppressing the results of the lower-level computations and superseding their results [1]. In three tier architectures, higher-level layers interface with lower level ones by providing input or advice to the lower-level layers or to answer the queries of the lower layers [2]. Recent robotic control architectures compose of three layers: deliberation, sequencing and reactive feedback control [2]. Researchers use one or more of these layers in their architectures depending on the way they look at how robotic tasks should be handled. If the robotic environment necessitates planning, that is, when a timely order of actions is needed before execution, a deliberation dependent architecture is used. However, if the real world is changing frequently as well as there's uncertainty in the world, the reactivity part of the architecture is stressed. The researchers in the latter group argue the usefulness of planning in dynamic environments in which it will be impossible to execute a whole plan to the end without any changes in the environment. They conclude that generating a plan that will become obsolete is unnecessary.

In our study, intend is to generalize the architecture in a way that it can be used to handle many different tasks in the real world. This brings with it the result that there will be tasks that need more deliberation than reactivity and tasks that will need more reactivity than deliberation. For that reason, including only one layer will prevent generality and hence a three layer approach helps to obtain generality in this study. The second thing is that when there's not much need for reactivity and when a planning execution error occurs because of an unexpected happening in the environment, the error handling can be more efficient at the deliberation layer where planning is also done. As a result, if there's planning in a system, handling execution errors at planning level in addition to the handling of sequencing and reactive levels can increase time and energy efficiency.

The paper is organized as follows: In Section 2, after giving a brief survey of literature, we present our replanning approach at the deliberation level. In Section 3, our domain dependent planning framework is given. Section 4 explains the vision support of the whole system. Finally, in Section 5, concluding remarks are made.

2 Replanning at the Deliberation Level

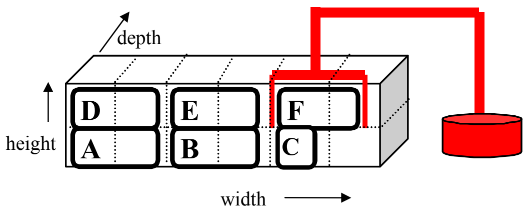
Some previous work does replanning at the highest level [3, 4]. However, these studies make some modifications in a current plan to make the plan useful for the situation after the effects of unexpected happenings appear on the workspace of the robot. The way to handle an obsolete plan in case of unexpected happenings is dependent on the nature of the tasks that the plans are generated for. For example, in case that a mobile robot misses a corridor on its way, the only way to travel along the corridor is to realize that it missed it and go back to the entrance of the corridor and direct its way through the corridor [5]. Although, there doesn't seem to be any other alternative for the robot to correct its action in the above case, there are problems that provide the opportunity of selecting alternative action paths to achieve a given goal in case of unexpected happenings. In this study, we extend our problem space to this class of problems and investigate two of them for replanning at deliberation layer.

The first one is the "mixed pieces problem". A detailed definition of mixed pieces problem can be found in [6]. A brief description is as follows:

Given are an initial and a goal state for a robot workspace. The objects in the workspace are blocks with sizes that a robot arm can grip. The blocks are positioned in cells of a chessboard like grid structure. They are labeled with letters on the top to differentiate them from each other. The problem is to find a sequence of actions with minimal cost that will place the blocks from their initial state positions to their goal state positions. In our solution, the cost of a sequence of actions is *estimated* by using *a learning mechanism* that makes use of traveled distance and planning time of previous executions [6]. The key point in this problem is that more than one block can have the same label letter and that a block can be placed in any one of the goal positions that its label letter fits. The other important points are:

1) The destination position of a block can be occupied.
2) An unexpected happening can occur during the transfer of the blocks from their initial state to their goal state such as the mixing of blocks and hence replanning might be necessary.

The other one is the "container positioning problem" given in Fig. 1. In this problem, a bunch of containers are to be placed from a ship to a specific place at a port. Also a container can be removed from its place in the port back to the ship. These operations are achieved under some previously defined constraints. The constraints can be the size of the blocks, the maximum number of blocks that can be placed on top of each other, the rule that a large size container cannot be placed on top of a small size container but on two small size containers, etc. The optimality of distance traveled by the robot arm is also considered in this problem. In addition, the system might need to replan in case that some constraints are avoided or a container is placed in a wrong position depending on the timely accumulating mechanical errors in the arm.



In the left port, container "F" shouldn't be inserted on top of container "C". Six small size containers or three small size containers can be placed next to each other.

Fig. 1. An example container domain.

When these two problems are examined carefully, it will be seen that the order of the actions in a sequence to achieve a task is important for obtaining time and energy optimality. In both of these problems, an initial and a goal state is defined and a plan sequence is generated such that the robot arm travels an optimal distance when a task is accomplished and spends the minimal planning effort before the plan is put into execution. Optimality is also considered in the case of unexpected events changing the workspace of a robot. Thus, the idea of taking prospective actions that minimize both the overall distance to be traveled and the planning time is also present in replanning. The new sequence of actions generated as a result of replanning is either completely or partially different from the original plan.

The pick and place sequencing problems that are explored to implement the proposed replanning framework are optimality problems. However, current software is not only designed to solve optimality problems but also other types of problems that can be solved within this scope if necessary domain definitions and rules are given to the system.

Some planning domains in literature replan by either assigning alternative values to the variable(s) of a selected operator for a situation if currently assigned values don't produce the expected effects or by selecting an alternative operator for a situation if the selected one doesn't achieve the intended action [7, 8]. That is, replanning is finding the right "value(s), operator" combination that produces the expected effects when executed in a certain situation. On the other hand, some planners send the robot command sequences for multiple actions or even complete plans assuming that action failures, beneficial side effects and exogenous events will be rare without making a monitor at the end of execution of each action. Some planners use internal recovery procedures that handle common, known failures with simple and directly applicable behaviour.

The first of the above approaches corresponds to generating a new sequence of actions from the state affected by the unexpected happening to the goal state. In this study, this approach is only one of the options for handling unexpected events. The other option is to clear the effects of an unexpected event and continue with the rest of the original plan. The final but an important option is to go to a previously stored intermediate state that is known to be on the path of an optimal plan. The optimal plan is generated previously for a similar situation. The idea behind considering alternative approaches in a replanning situation is to integrate optimality consideration in replanning if optimality itself is not too costly to obtain. Although generating optimal or suboptimal plans during planning is intended, this is not always possible due to the nature of some problems, i.e. NP-complete problems. Hence, integrating optimality consideration into replanning helps in overall efficiency of a robotic system. The algorithms for planning and replanning approaches will be given in the following paragraphs. It should be noted that there's a scheduler for scheduling subtasks in a robotic system. Neural Network training, receiving of initial and goal states, planning and execution of plans are the subtasks defined.

Following is the algorithm for the task scheduler.

```
main(){
NN_training (); //Neural Network is trained for
//segmenting and recognizing objects in workspace later.
user_input () ; //User inputs initial and goal states
//for real world execution or random matrices are
//generated for initial, goal states for simulation.
Expected_World_Model = Initial_State;
plan = planner(Initial_State, Goal_State);
execute(plan);
}
```

The execute function whose algorithm given below executes each step in a plan by sending the plan step to the robot controller. When the execution of a plan step is completed, Expected_World_Model is updated with the belief of execute function. Later, an image of the world is grabbed, processed and compared with the belief to figure out if the system is in synchronization with the real world. If there's a difference between the Expected_World_Model and the real world then replanning is done.

```
execute(plan){//plan is the name of the file
              //containing a plan.
for each plan step {
  plan_step = get_a_plan_step_from_plan();
  RobotController(plan_step);//executive layer
  Update Expected_World_Model;
  Grab an image of the world;
  Process grabbed image to form Observed_World_Model;
  dm=compare(Expected_World_Model,Observed_World_Model);
  if (dm > 0) do replanning; }}
```

The replanner has three alternatives to choose from. An intermediate state which is known to be on the optimal or suboptimal path has always the priority over the other two alternatives. In the following algorithm, if the function get_intermediate_state() returns true, that is , there's a previously obtained intermediate state for a similar situation, then the plan stored in the plan base is retrieved and is put into execution. Otherwise, the other two alternatives are considered for replanning. If the value of variable "dm" which corresponds to a distance metric to measure the difference between two states and whose value was obtained in the latest call of the function "execute()" is less than a previously settled threshold value "thr", then the alternative of backtracking to the state before the unexpected event and executing rest of the original plan is chosen. In this case, new_plan corresponds to a sequence of actions that brings the state after the unexpected happening to the state right before the effects of the unexpected happening. Meanwhile, alter1_cost() function calculates the cost of using alternative 1 for replanning by taking into account planning time for backtracking and total distance traveled to reach the goal state. There's a learning mechanism behind choosing a proper alternative. The system *learns* the value of the threshold whose value was assigned randomly at startup as it progresses. The details of the learning mechanism are in [6]. On the other hand, if the value of dm is not less than the threshold value, the alternative of generating a completely new plan from the state with the effects of the unexpected happening to the goal state is selected. Cost calculation (alter2_cost()) is done and used accordingly. Plans are put into execution when they are generated.

```
replanner() {
  (bool) yes = get_intermediate_state();
  if(yes){
     new_plan = retrieve_plan_for_intermediate_state();
     execute(new_plan);}
  else if(dm != 0 && dm < thr) {
          alter1_cost();
          new_plan=planner(Observed_World_Model,
          Expected_World_Model);
          execute(new_plan);
          execute(plan); }//rest of plan will be executed.
        else { alter2_cost();
          new_plan = planner(Observed_World_Model,
                             Goal_State);
          execute(new_plan); }}
```

3 Domain Dependent Planning

In order to achieve a general planner, planning for different domains should be allowed. In this study, general planning is achieved by using a means-ends analysis planning mechanism [9] that is represented by predicate calculus and allows representation of different planning domains, domain operators and rules. Means-ends analysis is a paradigm that applies domain specific search control rules on a search tree during planning and hence chooses a sequence of operators. Operators are nodes of a search tree. The functionality of control rules can be extended to incorporate finding an optimal sequence of actions if there's one.

In Vision Guided Planner, VGP, there are two blocks world domains. The first one is the mixed pieces domain where the other is the container load/unload domain in a port. Each domain has its own search control rules and the planner activates the domain specific control rules when planning for a domain. The objects in a domain are represented by a structure called object and it has attributes such as its type, name, size, present x and y coordinates, goal x and y coordinates, etc.

The planning algorithm for mixed pieces domain solves the mixed pieces problem and is later expressed in the form of predicate calculus. The algorithm is as follows:

```
Mixed_Pieces(){
Object obj;
Read_State(Initial); Read_Satete(Goal);
int unprocessed=find_number_of_unprocessed_blocks();
  strcpy(obj.type,"ARM");
  While(unprocessed > 0) {
  if(obj.type == "BLOCK"){
       find_closest_destination_from_block(obj);
       if(empty_destination(obj)){
          move_block(obj);
          strcpy(obj.type, "ARM");}
        else {
          find_closest_empty_cell_from_destination(obj);
          //Above function incorporates the movement of
          //the block in destination cell of object obj
          //to the empty cell.
          move_block(obj);//move block from initial to
          //goal position.
            strcpy(obj.type, "ARM");}
  else if(obj.type == "ARM") {
          find_closest_destination_from_arm(obj);
          move_arm(obj); //move to the coordinates of
          //block to be moved.
          strcpy(obj.type, "BLOCK");} //assign next
          //moving object as "BLOCK".
  unprocessed = unprocessed – 1;
  }
```

The planning algorithm for container insertion and removal are same as their corresponding control rules. Two functions in the algorithms will be presented below:

```
int find_closest_empty_destination_from_arm(object *obj,
int *x, int *y, int *z){
int i,j,k;
for(i = 1; i <= depth; i++)   // three-dimensional
  for(j = 1; j <= height; j++)      //domain
    for(k = 1; k <= width; k++){
      if(obj->size == 0 && port[i][j][k].name==' '
         && port[i][j-1][k].name != ' ')//checks that
         //underneath is full. Small is 0, large is 1.
         {*x=i;*y=j;*z=k; return 0;}
      if(obj->size == 1 && (k+1)<=width &&
         port[i][j][k].name==' ' &&
         port[i][j][k+1].name==' ' &&
         port[i][j-1][k].name != ' ' &&
         port[i][j-1][k+1].name != ' ')
         {*x=i;*y=j;*z=k; return 0;}}
return 1;}
```

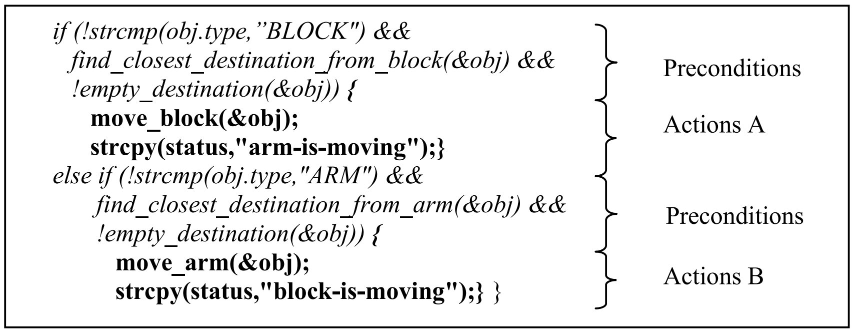
Assume that a port consists of a three-dimensional grid structure as in Fig. 1 and each three-dimensional grid is called a cell. A small container occupies one cell where as a large container occupies two cells. The first if in the above function checks if a cell (i, j, k) at port is empty, if container size is small (size 0), and if cell (i, j-1, k), that is, the cell below (i, j, k) is full or floor. If it is full or floor, the function returns (i, j, k) for insertion and if not, continues to search for another empty cell.

In the second if, a large container is being inserted. A large container occupies two cells next to each other, i.e. (i, j, k) and (i, j+1,k). Second if checks whether container size is large, whether (i, j, k) and (i, j, k+1) are empty, whether k+1 is within grid structure and whether (i, j-1, k) and (i, j-1, k+1) are full or floor. If they are, the function returns (i, j, k) for insertion and if not, continues to search for another empty cell.

```
void make_container_approachable(object obj){
int x,y,z,j,toplevel = 0;
toplevel = get_topmost_object(obj);
for(j=toplevel; j > 0; j--)
  get_temp_destination(obj, &x, &y, &z); // return
  //temporary destination coordinates in x,y,z.
  move_to_temp_destination(obj,x,y,z,toplevel);
  port[obj.curlocz][j][obj.curlocx].name=' '; //make
  //topmost cell empty.
}}
```

In the above function, get_topmost_object(obj) function gets the y axis coordinate of the topmost container on container to be removed. get_temp_destination(obj, &x, &y, &z) function returns the coordinates of a temporary cell for the topmost container. move_to_temp_destination(obj, x, y, z, toplevel) function moves the topmost container to the temporary cell with (x, y, z) coordinates. Since, the topmost container is moved now, its place is assigned empty with the next statement.

The control rules in predicate calculus and the planner algorithm are given in the following figures.



```
if (!strcmp(obj.type,"BLOCK") &&
  find_closest_destination_from_block(&obj) &&
  !empty_destination(&obj)) {
    move_block(&obj);
    strcpy(status,"arm-is-moving");}
else if (!strcmp(obj.type,"ARM") &&
    find_closest_destination_from_arm(&obj) &&
    !empty_destination(&obj)) {
      move_arm(&obj);
      strcpy(status,"block-is-moving");} }
```

Fig. 2. Control Rule for Mixed Pieces Domain.

```
void controlRule_container_insertion(){
int x,y,z;
if(!strcmp(obj.type,"container") &&                                   } Prec.
  !find_closest_empty_destination_from_arm(&obj,&x,&y,&z)) {          }
        insert_container(&obj,x,y,z);                       } Actions
        strcpy(status,"insertion-is-completed");}            }
else {strcpy(status,"no-place-for-insertion");              } Actions
        unprocessed --; } }                                  }
```

Fig. 3. Control Rule for Container Domain (Insertion).

```
void controlRule_container_removal(){
find_pose_of_container(&obj);
if(!strcmp(obj.type,"container") &&                 } Preconditions
  !container_top_full(&obj)) {                      }
        remove_container(&obj);                     }
        strcpy(status,"removal-is-completed");}     }   Actions
else { make_container_approachable(&obj);           }
        remove_container(&obj);                     }   Actions
        strcpy(status,"removal-is-completed");}}    }
```

Fig. 4. Control Rule for Container Domain (Removal).

```
planner(initial, goal)
{
While(!all goals are executed)
            Apply domain specific search control rule ;
}
```

Fig. 5. Planner that takes two states as input and produces a plan.

4 Vision Support for Replanning

For each of the domains, vision information is supplied to the replanning level. The objects in mixed pieces domain are labeled with letters, A, V, S, K, P, F for black pieces and X, Y, Z, T, U, W for white pieces (Fig. 6). The blocks in port domain can be labeled with any of these letters. In order to recognize the objects, the top view image of a domain is grabbed after each execution step of a plan and is segmented. A backpropogation neural network is used for the recognition of the letters in the labels. Currently, the number of hidden layers in neural net is one and a feature vector with

16 features is supplied for a letter to be trained and recognized. Each segmented image is fifty pixels in length and width.

In port domain, blocks are allowed to be on top of each other but a two dimensional top view of the domain helps in obtaining the Observed_World_Model. However, Expected_World_Model for the port domain should have three dimensions to keep a correct belief of the workspace during execution. During detection of exogenous events in port domain, three dimensional Expected_World_Model is map into two dimensions with only storing the top level blocks (in one container cell, a top element can be at level 3 where as in another one it can be at level 1) and making comparison possible with two dimensional Observed_World_Model. In mixed pieces domain, there's no need for such a mapping since both of these models have two dimensions.

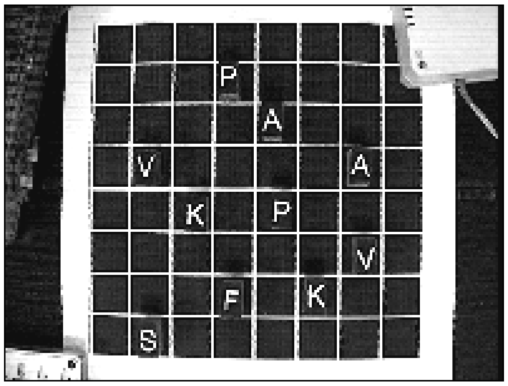
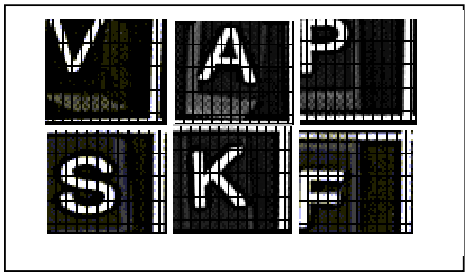


Fig. 6. A top view of mixed pieces domain.

The segmented cells and Observed World Model obtained from Fig. 6 is in Fig. 7.



				A			
	V					A	
		K		P			
						V	
				F	K		
	S						

Fig. 7. Segmented cells and the Observed_World_Model obtained from Fig. 6.

5 Conclusion

In this study, we have explained the details of a replanning framework we have proposed for achieving robotic tasks. Although the implementation is done in blocks world due to the technical constraints, the implemented architecture is kept general for other domain definitions such as constraints, search control rules, etc. The originality of the replanning mechanism comes from the fact it incorporates a learning based decision making paradigm and it is at the highest level of abstraction instead of in sequencing and reactive control layers of three tier architectures. The implementation shows us that this approach can be used in planning domains also increasing efficiency in planning and executing plans.

6 References

1. Arkin, R. C., "Behaviour-Based Robotics", The MIT Press, Cambridge, Massachusetts, 1999.

2. Gat, E., 1998, "Three Layer Architectures", in Artificial Intelligence and Mobile Robots, MIT Press.

3. Laird, E. L., Congdon, C. B., and Coulter, K. J., 1998, "The Soar User's Manual Version 8.2", University of Michigan.

4. Veloso, M. M., Carbonell, J., P'erez, M. A, Borrajo, D., Fink, E., and Blythe, J., 1995, "Integrating planning and learning: The Prodigy architecture", Journal of Experimental and Theoretical Artificial Intelligence, 7(1), pp. 81-120.

5. Koening, S. and Simmons, R. G., "Xavier: A Robot Navigation Architecture Based on Partially Observable Markov Decision Process Models". , in: *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R.P. Bonasso, R. Murphy (eds.), MIT Press, 1998.

6. Yıldırım, Ş. and Tunalı, T., 1999, "A new methodology for dealing with uncertainty in robotic tasks", XIV. Int. Symp. on Comp.& Inf.Sci., Kuşadası, TURKİYE.

7. Haigh, K. Z., 1998, "Situation-Dependent Learning for Interleaved Planning and Robot Execution", Ph.D thesis, CMU.

8. Pryor, L., & Collins, G., 1996, "Planning for Contingencies: A Decision_based Approach", Journal of Artificial Intelligence Research, 4, pp. 287-339.

9. Newell, A. and Simon, H. (1963). GPS: A program that simulates human thought. In *Computers and Thought*, ed. Feigenbaum and Feldman. McGraw-Hill, New York.