Robots That Create Alternative Plans against Failures

C. Ugur Usug. Dogan Altan. Sanem Sariel-Talay

Artificial Intelligence and Robotics Laboratory
Computer Engineering Department
Istanbul Technical University, Istanbul, Turkey
(e-mail: {usugc,daltan,sariel}@itu.edu.tr)

Abstract: Automated action planning is crucial for efficient execution of mobile robot missions. Automated planners use complete domain descriptions to construct plans. Nevertheless, there is usually a gap between the real world and its representation. Therefore, there is another source of uncertainty for mobile robot systems due to the impossibility of perfectly representing action descriptions (e.g., preconditions and effects) in all circumstances. Incomplete domain representations may lead a planner to fail constructing a valid plan when unforeseen events are encountered. We investigate these types of situations, especially the failure cases and how robots can recover from real-time execution failures. The main focus of our research is to design a dynamic planning framework which can generate alternative plans by applying generic updates in the domain representation when the execution of a plan fails. Our proposed method constructs new feasible plans by using the updated domain representations even if the outcomes of the operators are partially known in advance or feasible plans are not possible with the original representation of the domain. Besides updating the domain representation, our method manipulates the planner by using a reasoning mechanism so that it chooses more relevant actions to recover from failures. This is achieved by considering the effects of the failed action and trying to accomplish these effects with alternative actions.

Keywords: Multirobot planning, dynamic planning, failure recovery.

1. INTRODUCTION

Cognitive robots need automated onboard high-level action planning for online generation of action sequences against exogenous events. Temporal planners, capable of generating efficient sequences of durative actions, are convenient to be used with real robot missions. There have been significant advances in devising efficient temporal planners for continually enhanced benchmark planning domains. However, these algorithms operate at a high level and are not completely able to deal with hardware or physical environment limitations. In particular, real world is partially observable and involves different sources of uncertainty due to noisy sensor information, unexpected outcomes of actions and failures. Our research focuses on continual planning, execution and monitoring issues for cognitive robots. In particular, we investigate failure recovery methods for enabling efficient completion of tasks on unexpected outcomes.

Failure detection and diagnosis is investigated in earlier planning frameworks (de Jonge, Roos and Witteveen 2009) and plan repairing methods are proposed for recovering from failures (van der Krogt and de Weerdt 2005, Micalizio 2009). However, in some real-world cases, a planner may not come up with a valid (re)plan with the available operators at hand (Brenner and Nebel 2009, Göbelbecker et al. 2010) but with unforeseen opportunistic features or outcomes of actions. This is due to lack of detailed and realistic representations (e.g., preconditions and effects) of actions (i.e., real-world instantiations of planning operators) in a planning domain or the abstraction of the problem for reducing complexity. When action representations are incorrect or incomplete, learning methods are required. However, even for simple learning methods, a certain amount of background knowledge is needed.

Our prior work (Usug and Sariel-Talay 2011) presents a dynamic temporal planning framework for mobile robots to handle action failures which occur due to misbeliefs about environmental issues. In this work, we extend that framework by applying more intuitive reasoning methods in selecting alternative actions to overcome failures. These types of failures are analysed in a taxonomy of failures presented in our earlier work (Karapinar, Altan and Sariel-Talay 2012). Different from earlier studies, the presented solution constructs alternative plans even when guidance by experts, relevant new information or repairing operators for replanning is not available. Real outcomes of existing operators may not be completely known in advance (e.g., due to abstraction) or valid plans are not possible with the existing representation of the domain. The proposed framework includes a domain representation update procedure that provides reasoning tools to replan accordingly for the resolution of a failure by reasoning about the cause of a failure in planning level using low-level scene understanding. Background knowledge needed for updating the existing domain operators is almost negligible and a generic approach is applied by searching for the effects of existing operators that may resolve the failure. After the

required domain representation updates are performed, a replanning approach is employed as opposed to repairing since makespan (near-) optimal solutions are targeted (Cushing, Benton and Kambhampati 2008). TLPlan (Bacchus and Ady 2001) is used as a forward chaining temporal planner in the system to construct makespan-optimized plans. Low and high-level planning procedures of the system are made compatible by using efficient domain representations (e.g., map).

The rest of the paper is structured as follows. The next section presents the formulation of the investigated problem. Section 3 describes the dynamic temporal planning framework as a proposed solution to the presented problem and the developed algorithms for domain representation updates and replanning. The experimental results are presented in the following section and then, the paper is concluded with suggestions for future work.

2. PROBLEM STATEMENT

A planning domain is as a tuple $\Delta = (\mathcal{T}, \mathcal{C}, \mathcal{P}, \mathcal{O})$ where $\mathcal{C}$ is a set of constants, $\mathcal{T}$ is a set of types, $\mathcal{P}$ is a set of predicates and $\mathcal{O}$ is a set of planning operators representing real-world actions that can be executed by robots. A planning problem is modeled as $P_r = (\Delta, I, G)$ where I and G are initial and goal states, respectively. Each world state includes a set of facts including the representations of resources and robots in the system ($r_i \in R$). A planning operator $o \in \mathcal{O}$ is represented with a set of parameters $param(o)$, a set of preconditions $pre(o)$ and add/delete effects $add(o)/del(o)$. An operator o is only applicable in the current state S if $pre(o) \subseteq S$. Whenever o is applied, the world state S is transformed to a successor S' which is represented as $add(o) \cup (S \setminus del(o))$.

A temporal plan P, a solution to a planning problem P_r, satisfying $makespan(P)$ optimization with a sequence of instantiated and scheduled operator instances, which correspond to real-world actions ($operator(a) = o$). Each action a is represented with a set of arguments $arg(a)$, a set of allocated robots for execution and a start time $t_s(a)$ in the constructed temporal plan. Action a has $dur(a)$ representing the total duration (*i.e.*, the amount of duration between $t_s(a)$ and the time step that $at-end$ effects are available) of the corresponding real-world action. $valid(P, S, G)$ is a simulation control function that checks the validity of the plan P prior to its execution. A plan is valid if its operators satisfy the constraints and achieves the goal state G from the current state S by applying them. An *executable* plan is a *valid* plan of which real-world execution is possible with the existing resources. In some situations, even when a plan P is *valid*, its runtime execution may fail. In this case, P becomes *non-executable* during the execution of an action, a_{failed}, in state S_{failed}. The *cause* of failure is represented as $c \in S_{failed}$ and it has an attribute list of $attr(c)$ defined in Δ and each element $attr_i$ of this list has a value $value(attr_i, S)$ in S. It is assumed that c is detected as just the subject of the action or encountered objects by the low level components (*e.g.*, perception and path planning modules) of the robot. When such a failure occurs, the existing plan cannot be executed and replanning is needed. In this case, the corresponding domain representation update procedure transforms the planning problem from P_r to P_r' for replanning. P_r' includes the encoded *cause, c* and the failure situation. The planner may still fail to find an *executable* plan for P_r' due to missing information although an *executable* plan exists in the real world. In that case, the updated planning problem P_r' should be transformed to P_r'' that includes necessary updates to generate an *executable* plan.

The problem that we investigate in this paper is updating the domain representation appropriately whenever an execution failure occurs. Relevant updates should be made in such a way to autonomously find an *executable* plan involving alternative actions to the failed ones. Note that before the domain representation is updated, these alternative actions may not be applicable for generating a valid plan due to incomplete representations. Therefore, an *executable* (also *valid*) plan is to be found for P_r'' even when no *valid* plan is found for P_r' or there are *valid* but *non-executable* plans (*i.e.*, with non-executable actions a for which $pre(a) \subseteq S_{failed}$) generated by replanning for P_r'.

The presented problem involves two types of action execution failures, namely, *temporary* and *permanent* failures. *Temporary failures* occur at runtime and can be resolved by replanning. Temporarily failed actions may become available whenever alternative actions are executed to achieve the required preconditions of these failed actions. When an action *permanently* fails, there is no way to execute that action to achieve the goals.

As a motivating example to illustrate the above mentioned problem, a single-robot scenario can be given with an object, located on one end of a corridor, to be moved to the other end. The robot is able to execute several actions such as pick/drop objects by its gripper, move from one location to another and push movable objects. However, since the environment is unstructured, the move action fails *temporarily* while transferring the object due to an unknown obstacle blocking the passage to the destination. This failure is detected by the path planner module of the robot after updating its map and path plan accordingly. If the path planner cannot find a solution, replanning is needed in the higher level. Although a *valid* plan is impossible from the perspective of the planner, there indeed exists a *valid* and *executable* plan according to which the robot pushes this obstacle until the target is reachable. Although action *push* has a single effect of changing the location of an object, it is not designed to have an effect of clearing the pathway to reach at the target which prevents it to be included in the replan.

3. DYNAMIC TEMPORAL PLANNING

One way to handle a real-time action execution failure is changing the initial and/or goal state representations (Fox et al. 2006) to reconstruct a *valid* plan. However, it is not guaranteed that this new plan is *executable* in the real world. Additional precautions should be taken to prevent generating such plans (Cushing and Kambhampati 2005). There may be different intuitive solutions to the presented problem: (1)

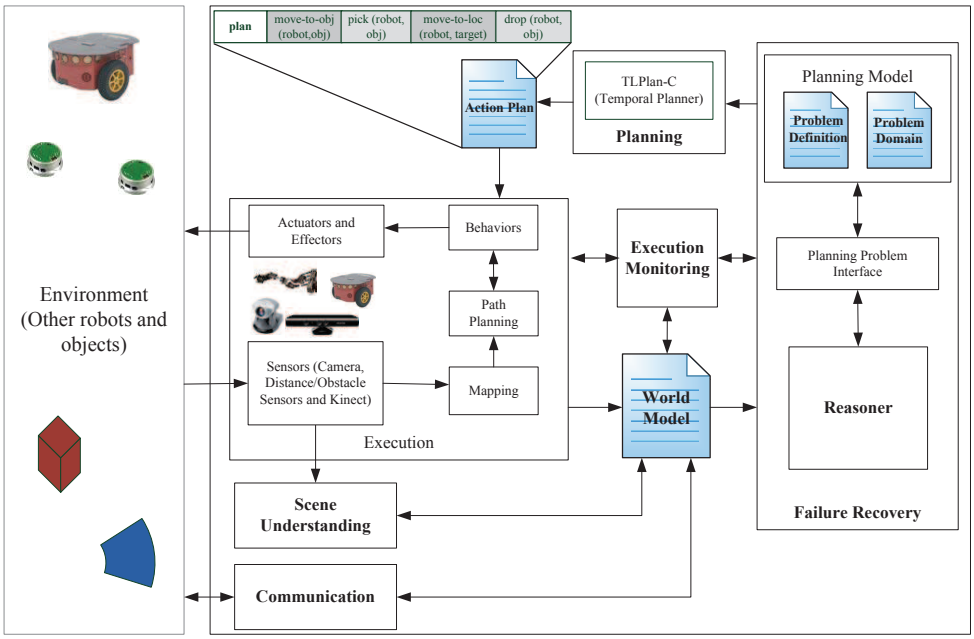

Fig. 1. Dynamic Temporal Planning Framework

disabling the failed actions, (2) *suspending* the failed actions, (3) incremental or subgoal planning. Disabling the failed actions may prevent their use in the constructed plan permanently. The second method overcomes this problem by just suspending the selection of the failed action for a certain time period. In this case, since the duration of suspension is domain-dependent and unpredictable, a complicated reasoning process is needed to estimate the realistic duration for the suspension. Incremental or subgoal planning is another method, however, in this case, suboptimal solutions may be observed or the planner may fail to find a subplan to achieve subgoals if there is missing domain information.

Our solution to the presented problem involves a dynamic temporal planning framework (Fig. 1) to handle *temporary* action failures. The overall framework employs four interleaved processes, namely, planning, execution, monitoring and updating the knowledge base and recovering from failures if any. The framework constructs plans to achieve given goals, executes the actions in the constructed plan, updates the domain representation appropriately on a failure, reasons about the failure and, then replans continually. After appropriate domain representation updates are performed, a new *executable* plan can be constructed if any, even when there is not a *valid* plan with the given operator representations. Domain representation updates include *causes* of a failure and conclusions from reasoning, *R,* that includes the attributes which the failed action intends to change. These updates are used in the replaning phase. *R* is initially an empty set, and in this situation, planner constructs new plans without any reasoning. However, after telling the *cause* of the failure in the knowledge base and updating the reasoning set, more relevant actions could be selected by the planner in the replanning phase. Depending on the quality of the reasoning about the failure *cause*, the results of the update procedure can be more specific when semantic attachments can be made. For example, if an obstacle-related *cause* is detected by the path planner, only the attributes of *cause* which are related to the location of the object should be considered for replanning. However, if no such reasoning is available, irrelevant preconditions including attributes such as the color of the obstacle may also be considered.

3.1. Planning and Execution

The framework includes a temporal planner which is continually called whenever replanning is needed. The entire procedures for robust continual mission execution are given in Algorithm 1. There are four execution states: *start*, *executing*, *suspended* and *failed*. In state *start*, a new plan is constructed and the robots start executing this renewed plan. State *executing* is active whenever robots are in execution of the plan. State *failed* is activated when the monitoring process detects an action failure, and state *suspended* is activated after state *failed* until the execution is ceased.

The algorithm starts with state *start* for planning (line 22) using the existing domain representation. When a plan is constructed, it is sent to the robots. In state *executing*, the algorithm is suspended on WAITMESSAGE subroutine waiting for any interruption from the monitoring process (at line 7). Robots communicate with each other and each message contains the state of action execution for a specified action, perceptual information about the domain and the *cause* of failure if any. Receiving a message, Algorithm 1 resumes and calls the MONITOR subroutine (Algorithm 2). Algorithm 1 remains in state *executing* until a failure message is received from a robot in the domain or VALID(P, S, G) subroutine returns *false* when the corresponding plan becomes *invalid*. In this case, a *stop-execution* message is sent to all robots. Receiving a stop message, each robot stops the execution of its action.

Whenever the plan is no longer executable by the robots, the cause of the failure is encoded in the domain representation and state *failed* is activated. Then, the validity of the failed plan is checked for the updated state information. If the failed plan is still treated as valid from the planner's perspective,

Algorithm 1 DYNAMICTEMPORALPLANNING (S, G)

Input: current state S, goal state G
Output: returns success or failure: messages to robots are sent

```
1:  msg = NULL
2:  P = ∅,  R = ∅
3:  status = start
4:  for S ⊉ G do
5:    if status = executing then
6:      if VALID(P, S, G)
7:        msg = WAITMESSAGE()
8:        status = MONITOR(msg, S)
9:      else
10:       SENDALLROBOTSSTOPEXECUTION()
11:       status = suspended
12:   else if status = failed then
13:     if VALID(P, S, G) then
14:       UPDATEDOMAINDESCRIPTION(msg, Δ, S, P, R)
15:     status = suspended
16:   else if status = suspended then
17:     for all robots stop execution do
18:       msg = WAITMESSAGE()
19:       MONITOR(msg, S)
20:     status = start
21:   else
22:     P = PLANNER(Δ, S, G)
23:     if P = ∅ then
24:       return NOPLAN
25:     status = executing
26:     SENDPLANTOROBOTS(P)
27: return SUCCESS
```

Algorithm 2 MONITOR (msg, S)

Input: message msg, currrent state S
Output: status of planning, current state S

```
1:  status = executing
2:  a = msg.action
3:  state_a = msg.state
4:  percv = msg.perception
5:  if state_a = started then
6:    S = APPLYSTARTEFFECTS(a, S)
7:  else if state_a = finished then
8:    S = APPLYENDEFFECTS(a, S)
9:    S = MERGE(S, percv)
10: else if state_a = failed then
11:   S = MERGE(S, percv)
12:   status = failed
13: else if state_a = stopped then
14:   S = MERGE(S, percv)
15: return status
```

replanning cannot handle this failure case. In this case, UPDATEDOMAINDESCRIPTION subroutine is called to make the required domain representation updates, reason about the failure and handle it. The body of this subroutine is described in the following subsection given in Algorithm 3. After the domain representation is updated, Algorithm 1 waits for execution-stopped messages from all robots. After then, the state is changed to *start* and a new plan is constructed.

3.2. Execution Monitoring and Failure Recovery

Algorithm 2 is used to monitor the status of the existing plan and to parse messages coming from robots. Monitoring of the existing plan is performed by simulating the plan from the current world state. The plan is simulated by applying the start and end effects of the actions to the fact base. The perceptions are applied on the current state, when *finished*, *started* and *stopped* messages are received, the status is set to *executing*. However, when a *failed* message is received, the status is set to *failed* which activates replanning. When failures are detected, the required domain updates are applied in the knowledge base.

Algorithm 3 implements the appropriate domain updates. At lines (5) to (9), the failed action is *locked* by defining a new predicate (newPredicate) with the name of the operator (concatenated with string "_locked") and its parameters in the domain. This predicate is added to the preconditions of the failed operator. Then, a corresponding fact (newFact) is included in the domain description. At line (14), reasoning set is updated by adding the attributes which exist in the effects of the failed action in the domain description. This update makes the planner to primarily choose the actions that suppress the failure *cause*. At lines (15) to (19), a new pseudo operator (o_{pseudo}) is created for the failed action-cause pairs $(a_{failed} - c)$, if not created before (for avoiding from loops). Line (21) assigns the pseudo operator to the corresponding failed action and cause pair. This pseudo operator diverges from the original domain operators by its specific preconditions, effects and parameters for representing the failure *cause*. It has additional preconditions for restricting its selection (for preventing cycling plans) only when the corresponding failed action is *locked* and the related *cause* is given as an argument. It has also additional effects for unlocking the failed action.

The appropriate preconditions are generated by MAKEFAILUREPRECONDITIONS subroutine (Algorithm 4). This subroutine first searches for the domain operators which possess the related effects to the attributes of cause. Then, it creates a precondition list as a disjunction of the inequality statements between the attributes of cause and their values in the failed state S_{failed}. In addition to this operation, the attributes that do not exist in the reasoning set R are eliminated to manipulate the planner to choose relevant actions to overcome the failure in its first attempt. Therefore, these inequality statements ensure reactivating the failed actions when the status of cause is changed in a desired way. If the pseudo operator is created earlier, and the plan is still non-executable, that means the alternative action does not produce the desired effect for resolving the failure. In this case, lines (23) to (30) in Algorithm 3 remove the related effects of this alternative action from the list of the preconditions of the pseudo operator. Therefore, it is not considered as an alternative action in future plans. An inappropriate alternative action is not considered in a future plan for the resolution of the related failure.

Algorithm 3 UPDATEDOMAINDESCRIPTION (msg,Δ,S,P, R)

Input: message msg, planning domain Δ,
current state S, current failed plan P, reasoning set R
Output: updated planning domain Δ and reasoning set R

$c = msg.cause$
$a_{failed} = msg.action$
$o_{failed} = operator(a_{failed})$
$preList = \emptyset$
newPredicate. name = CONCAT($name(o_{failed})$, "_locked")
newPredicate. parameters = param((o_{failed}))
$pre(o_{failed}) = pre(o_{failed}) \cup \neg\, newPredicate$
newFact = GENERATEFACT (newPredicate, $arg(a_{failed})$)
$S = S \cup newFact$
$\Delta = \Delta \cup newPredicate$
$preList = (newParam = c)$
$preList = preList \cup newPredicate$
o_{pseudo} = GETPSEUDOOPERATOR(a_{failed}, c)
$R = R \cup attr(a_{failed})$
if $o_{pseudo} = NIL$ **then**
  $param(o_{pseudo}) = param(o_{failed}) \cup newParam$
  $pre(o_{pseudo}) = preList$
  $pre(o_{pseudo}) = pre(o_{pseudo})$
    ∪ MAKEFAILUREPRECONDITIONS($c, \Delta, S, \emptyset, R$)
  $del(o_{pseudo}) = del(o_{pseudo}) \cup newFact$
  $\Delta = \Delta \cup o_{pseudo}$
  SETPSEUDOOPERATOR($a_{failed}, c, o_{pseudo}$)
else
  a_{pre} = GETPREDECESSORACTION(P, o_{pseudo}, c)
  $o_{pre} = operator(a_{pre})$
  $executedList$ = GETEXECUTEDOPERATORS(a_{failed}, c)
  $executedList = executedList \cup o_{pre}$
  SETEXECUTEDOPERATORS($a_{failed}, c, executedList$)
  $pre(o_{pseudo}) = preList$
  $pre(o_{pseudo}) = pre(o_{pseudo}) \cup$
  MAKEFAILUREPRECONDITIONS($c, \Delta, S, executedList, R$)

Algorithm 4 MAKEFAILUREPRECONDITIONS (c,Δ, S, l, R)

Input: failure cause c, planning domain Δ, reasoning set R
current state S, executed operator list l
Output: disjunctive precondition list $prelist$

$prelist = \emptyset$
for each o_i in Δ but not in l **do**
  for each *effect* e_j in $add(o_i) \cup del(o_i)$ **do**
    for each *attribute* $attr_k$ in $attr(c)$ **do**
      if e_j affects $attr_k$ **and** $attr_k \in R$ **then**
        $value = value(attr_k, S)$
        $prelist = prelist \cup \neg(attr_k = value)$
return $prelist$

4. SIMULATION RESULTS

A failure resolution scenario is analyzed for presenting the solution in the realistic Webots simulator with two mobile robots. This scenario includes a temporary action execution failure case to test and validate the success of the proposed method. There are 6 operators available in the planning domain: operator (move-to-loc ?robot ?loc) is to forward robots to a destination, operator (move-to-obj ?robot ?obj) is to forward robots to a position nearby an object to act on it, operators (pick ?robot ?smallObj) and (drop ?robot ?smallObj) are for picking/dropping a small object by the grippers of the robots, operator (push ?robot ?largeObj) is used to change the position of a large object by dragging, and operator (paint ?robot ?obj ?color) to paint an object.

The overall goal in the planning problem is transferring the small red cylindrical objects to a target location behind the obstacle (Fig. 4). However, the robots are not informed about the existence of the obstacle which blocks the passageway.

During the execution of the move-to-loc action, one of the robots detects the obstacle and its path planner returns a failure since it is impossible to generate a path from the current position to the destination. The failure cause of the action move-to-loc is reported as obstacle with its estimated location. This new information is encoded in the knowledge base (P_r'). If there is a valid solution at this step, replanning is performed. At this moment, it is assumed that there is no knowledge of how to resolve this failure (i.e., operator push would not be selected by the planner since it does not have an effect "clear the pathway to reach at the target"). Even when replanning is not possible, the proposed generic domain representation update method enables creating an alternative plan by means of the UPDATEDOMAINDESCRIPTION subroutine. This subroutine adds a pseudo operator for the reported cause to the domain to transform to a new planning problem P_r''.

```
(def-adl-operator (move-to-loc ?loc)
    (pre
      …
      (not (move-to-loc_locked ?loc))
    )
    …
)
(def-adl-operator (pseudo1 ?obj ?loc)
    (pre
      …
        (and
          (= ?obj obstacle)
          (move-to-loc_locked ?loc)
          (or
              (not (xcoord ?obj obs_x))
              (not (ycoord ?obj obs_y))
          )
        )
    )
    (del (move-to-loc_locked ?loc))
)
```

Fig. 2. Domain updates for the resolution of the failure. The color attribute of the object is not included in the pseudo operator since reasoning set R excludes it.

MAKEFAILUREPRECONDITIONS subroutine searches for the operators in the domain representation and finds operators *push* and *paint* as the related operators to the cause (obstacle) since they have the effects to change the attributes of an object. A new pseudo operator pseudo1 is created with the preconditions related to both the failed action (move-to-loc) and the effects x and y coordinates. Note that the color attribute of the object is not included in the pseudo operator since reasoning set R excludes it. The domain representation is updated with the inclusion of a disabling fact (move-to-loc_locked obstacle) of the failed action and the new pseudo

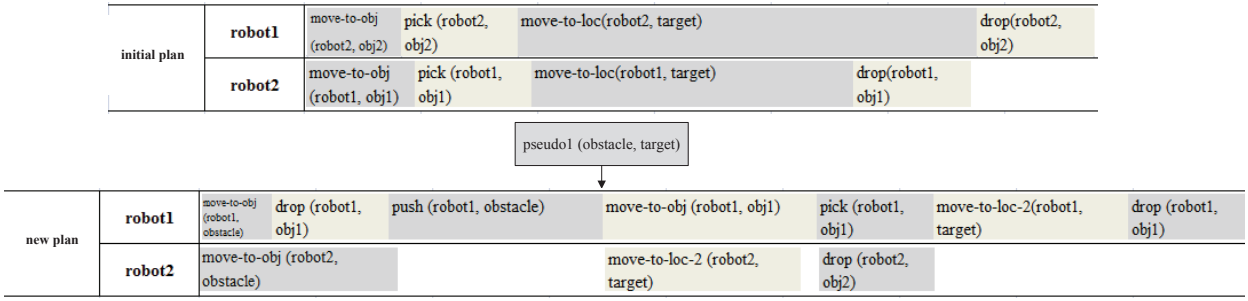

Fig. 3. The plans that are generated by the planner before (initial plan) and after (new plan) the failure

operator. All domain representation updates including the change made on the disabled action are illustrated in Fig. 2. The robots' plans for the given scenario are also illustrated in Fig. 3. The top row of the Fig. 3 shows the original plans of the robots. After detecting a failure, robots update their beliefs and the planner generates a new plan including the pseudo operator which enables the operator *push* to be included in the first robot's updated plan. Action paint is not included in the final plan since the color attribute of the object does not exist in the reasoning set *R*. Therefore, the planner chooses *push* action to overcome the failure. Since TLPlan generates makespan optimal plans, the durations of the actions, illustrated in Fig. 3, are also taken into consideration. Overall plan execution is illustrated in Fig. 4.

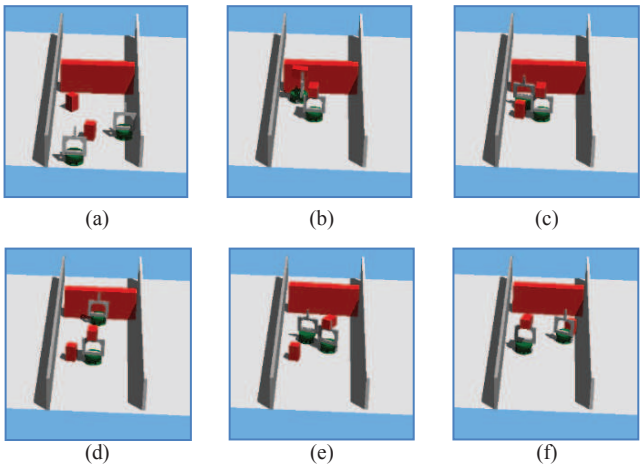

Fig. 4. The overall plan execution of moving the red objects to their destinations. (a-b) The initial plan is executed. (b) After detecting an obstacle, the necessary domain representation updates are applied and a new plan is generated. (c-f) The new plan involves pushing the obstacle to an appropriate location before the red objects are moved.

5. CONCLUSIONS

In this paper, we present a framework which successfully recovers from failures against environmental issues. The proposed method can efficiently handle temporary action execution failures by updating the robot's domain representation, and reasoning about the cause of failures. Different from our earlier study (Usug and Sariel-Talay 2011), the planner can generate alternative plans that recover from an encountered failure in a shorter time by an intuitive reasoning approach. The new plans are generated after appropriate domain representation updates. An example failure scenario is given in the Webots simulator; and the failure is successfully handled by applying our proposed method. As simulation results show, new valid plans are generated by replanning with the addition of new pseudo operators into the knowledge base and by reasoning to enable the planner choose actions that recover from the failure cause. The proposed method provides a new and efficient way to detect and recover from failures in dynamic environments. Our future work includes more advanced reasoning strategies to be included in the framework.

ACKNOWLEDGEMENT

This research is funded by a grant from the Scientific and Technological Research Council of Turkey (TUBITAK), Grant No. 111E-286.

REFERENCES

Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency: A forward chaining approach. In *Proceedings of IJCAI*, 417-424.

Brenner, M., and Nebel, B. 2009. Continual planning and acting in dynamic multiagent environments. In *AAMAS,* 19(3):297-331

Cushing W., and Kambhampati S. 2005. Replanning: A new perspective. In *Proceedings of ICAPS*.

Cushing, W.; Benton, J.; and Kambhampati, S. 2008. Replanning as a deliberative re-selection of objectives. *Arizona State University, Tech. Rep*.

de Jonge, F.; Roos, N.; Witteveen, C. 2009. Primary and secondary diagnosis of multi-agent plan. In *AAMAS*, 18(2): 267-294.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. In *Proceedings of ICAPS*, 212-221.

Göbelbecker, M.; Keller, T.; Eyerich, P.; Brenner, M.; Nebel, B. 2010. Coming up with good excuses: What to do when no plan can be found. In *Proceedings of The International Conference on Automated Planning and Scheduling (ICAPS)*, 81-88.

Karapinar S., Altan D. and Sariel-Talay S. 2012. A Robust Planning Framework for Cognitive Robots. *The AAAI-12 Workshop on Cognitive Robots (CogRob)*.

Micalizio, R. 2009. A distributed control loop for autonomous recovery in a multiagent plan. In *Proceedings of IJCAI*, 1760-1765.

Usug, U.C., and Sariel-Talay, S. 2011. Dynamic temporal planning for multirobot systems. *In Proceedings of the AAAI-11 Workshop on Automated Action Planning for Autonomous Mobile Robots (PAMR)*.

van der Krogt, R., and de Weerdt, M. 2005. Plan repair as an extension of planning. In *Proceedings of The International Conference on Automated Planning and Scheduling (ICAPS)*, 161-170.