An algorithm for replanning

R.P.J. van der Krogt A. Bos M.M. de Weerdt
C. Witteveen

Delft University of Technology
{r.p.j.vanderkrogt,a.bos,m.m.deweerdt,c.witteveen}@its.tudelft.nl

Abstract

An important aspect of agents is how they construct a plan to reach their goals. However, since most real-world agents live in a dynamic environment, often they will be confronted with situations where their plans are no longer feasible or optimal. In such situations, agents have to change their plan to deal with the new environment. In this paper we describe such a replanning process using a computational framework, consisting of so-called *resources* and *skills*, to represent the planned activities of an agent. An algorithm is introduced which can be used to replan the activities, taking the new environment into account.

1 Introduction

Often, agents have to achieve a number of given goals without a predefined way of how to do it. Therefore, they have to make a plan that consists of a number of actions, leading from the current state of the world to one of the states in which the goals have been accomplished. To assist them in planning tasks, a number of systems exists, e.g. Blackbox [5], Graphplan [1] and HSP [2]. These planners perform well under the assumptions they make: *deterministic effects* and *sole cause of change* [7]. The former states that the effects of all actions are known and static. The latter states that the world changes only because of the agent's actions. These assumptions are convenient during the planning phase, because they make the problem more manageable. However, once agents start executing their plan in a *dynamic environment* these assumptions no longer hold. Most likely, the agents are not the only one to live in the environment (violating the sole cause of change-assumption), and the agents' actions might have other effects than expected (making the effects non-deterministic), e.g. a robot's arm may get stuck and not move as expected.

This demonstrates the need for methods to *replan* an agent's actions in a dynamic environment. One way to replan is to give up the plan, and use standard planning tools to create a new plan. This is, however, a rather inefficient approach: not only will it likely take more time than adapting the current plan, it will also waste all the effort an agent has put in optimizing the current plan. Besides being inefficient, such an approach might also ruin agreements that have been made with

other agents. Therefore, we believe that using specialized plan revision methods (like in [3, 4]) to adapt the current plan into a plan which takes into account the new situation, is the preferred way of coping with a dynamic world. This paper introduces algorithmic methods for plan revision. Our method is not dependent on Strips as [4], but on a richer formalism based on skills and resources, as described in the next section. Therefore we focus on the availability of resources, while Hanks and Weld focus on the search through a graph of plans.

The remainder of this paper is organized as follows. First, in Section 2, we present a framework for representing an agent's plan and describe problems that may arise in a dynamic environment. Then, in Section 3, we give a detailed description of the algorithm. In the last section (Section 4) we discuss our future work.

2 A framework for planning and replanning

This section gives an overview of the framework we will use to model the plan of an agent. This is a modified version of the framework as described in [6]. We model processes, such as production or transportation, by *skills*. An application of a skill consumes a set of *resources* and produces a disjunct set of resources. These skills can be combined to reach certain *goals*. Such a combination is called a *plan*.

2.1 Resources, skills, goals and plans

We assume an enumerable set of resources $\mathcal{R}$. Resources can be used (and produced) exactly once. Each resource $r \in \mathcal{R}$ is identified by its *type* (a predicate symbol) and a unique identifier. We use predicate symbols to describe the type of a resource, like t_{BAL} for the type "truck in Baltimore", and $\frac{NYO}{BAL}$ for the type "free transport capacity from New York to Baltimore" To fully specify a resource, we give a predicate symbol followed by the unique identifier. For example, the term $t_{NYO}(34)$ represents "truck in New York number 34", and $t_{BAL}(36)$ for "truck in Baltimore number 36"[1]. A *resource scheme* is the set of all resources with the same predicate symbol. Furthermore we have an *extended resource scheme* which is a finite multi-set of resource schemes.

A *skill* is a rule of the form $RS_2 \leftarrow RS_1$ where RS_1 and RS_2 are extended resource schemes. We use $in(s)$ to denote RS_1 and $out(s)$ to denote RS_2. An example of a skill is given in Figure 1. This represents a drive of a truck from New York to Baltimore: from a truck currently in New York (t_{NYO}), it is possible to produce room for two loads from New York to Baltimore ($\frac{NYO}{BAL}$) and a truck in Baltimore (t_{BAL}). An application of a skill transforms a set of resources R_1 to a set of resources R_2. Such an application is specified by an instantiation θ changing each occurrence p of a predicate symbol in $in(s)$ and $out(s)$ to an occurrence of a fully specified resource, i.e. adding a unique identifier to each occurrence of p. The set of all skills will be denoted with $\mathcal{S}$.

$$\{\tfrac{NYO}{BAL}, \tfrac{NYO}{BAL}, t_{BAL}\} \leftarrow \{t_{NYO}\}$$

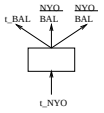

Figure 1: A skill (top) and its graphical representation

[1] Mind that these resources may refer to the same truck, but at a different place.

Let S be a set of skills, and let R_1 and R_2 be (disjunct) sets of resources. We say that R_2 *can immediately be produced* from R_1 *using* S, abbreviated by $R_1 \vdash_S R_2$, if there is a skill $s \in S$ and an instantiation θ such that $in(s)\theta \subseteq R_1$, i.e. all specific resource instances of the input of the skill occur in R_1, and $R_2 = (R_1 - in(s)\theta) \cup out(s)\theta$, i.e. the specific input instances are deleted from the original set of resources R_1, while the corresponding output instances of the skill are added to the remaining resources. We say that R_2 *is produced from* R_1 *using* S, if $R_1 \vdash_S^* R_2$ holds, where $\vdash_S^*$ denotes the reflexive, transitive closure of $\vdash_S$.

Skills are applied to a given set of resources R to realize a *goal scheme*. A goal scheme is an extended resource set denoting the types of the resources required. A goal scheme GS is *realizable* from R using an instantiation θ, if $R \vdash_S^* GS\theta$.

We represent a *plan* by a bi-partite Directed Acyclic Graph (DAG) $P = \langle N_R \cup N_S, E \rangle$, where $N_R \subseteq \mathcal{R}$ is a set of resource nodes, N_S is a set of skill nodes n_s where $s \in S$ and E a set of arcs. An example plan is given in Figure 2. For any two nodes $a \in N_R$ and $n_s \in N_S$, $(a, n_s) \in E$ means that resource a is used by an application of skill s, and $(n_s, a) \in E$ means that resource a is produced by an application of s. The set of input resources of P will be denoted by $In(P)$, whereas $Out(P)$ refers to the set of final products of P. We will use $d^+(n)$ to denote the out-degree of a node n and $d^-(n)$ to denote the in-degree of node n.

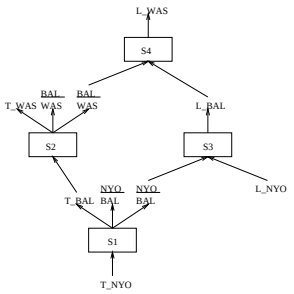

Figure 2: A plan

A *plan fragment* is a special kind of plan to define *services* an agent can provide. Using plan fragments, we need less actions to find a way to realize a set of goals and thus we make the problem more tractable. A *superskill* is the representation of a plan fragment PF as a skill which consumes $In(PF)$ and produces $Out(PF)$. The internals of PF are hidden in the superskill.

2.2 Problems

We assume that at forehand, an agent either selects or computes a plan that satisfies its initial goals. However, due to the dynamic nature of the world a previously constructed plan may become obsolete or inefficient. For example:

- *goal removal* Goals may be removed and therefore resources may become obsolete, for example because another agent has taken over the responsibility of goal satisfaction. An agent's plan then may be optimized: each of the skills that are used to produce these resources is a candidate to be removed.
- *goal addition* The agent is asked to satisfy a new goal. Possibly, the new goal is already produced as a by-product of the plan, but more likely, the plan has to be changed to include skills to reach the new goal. It is also possible that the new goal can only be produced if an existing goal is given up (thus freeing resources). If the latter is the case, the agent will have to decide if it will trade the old goal for the new one.
- *broken skills* Errors in the real world occur, causing skills to be "broken". That is, an error may cause a skill to no longer produce one or more of its

products (e.g., a reduction in the capacity of a truck). If these products are not used in the plan (they are pure by-products), the plan still satisfies all goals. Otherwise, we need to find an alternative way to produce the lost products. Or, one of the basic resources (e.g., a truck) of a plan may not be available due to some mishap. Again, alternative ways to realize the plan must be found.

As we have seen, an agent's plan is based on assumptions (e.g., availability of resources, and absence of errors) about the real world, that might be violated due to unanticipated events. It is impossible to consider all possible contingencies at forehand. Therefore, we assume that a plan is valid for most situations, but for those situations in which the plan fails to meet its goals, the agent has to adjust its plan to the new situation. The agent thereby has to take into account that the adjusted plan does not change the structure too much.

3 The replanning algorithm

Looking at the list of problems that might occur, we perceive that basically there are two possibilities: Either the need for a resource at some level of the planning graph disappears (due to a goal that is no longer needed, or due to a skill that no longer needs an input) or we need to create a resource at some level of the planning graph (because we need to include a new goal, because a skill requires an extra resource or because a skill doesn't produce a resource that is used by another skill). This means that if we create two basic algorithms, one for resource removal and one for resource addition, we can use these to build an algorithm for each of the problems mentioned in the previous section.

The basic solution for the first possibility is simple: Starting at the obsolete resource, we scan the plan for skills that are no longer required (because the only product that was used is a free resource now). Because of the removal of this skill, all resources that were used by this skill are now free, too. For these resources we also check whether this means some skills can be removed. We keep checking and removing skills like this, until there are no skills left that can be removed. This algorithm is shown in Figure 3.

```
REMOVE_SKILLS(P,r)
  1. if r is produced by a skill s then
       if ∀r' produced by s · d+(r') = 0 then
         DELETE(s)
         for all r'' consumed by s do
           REMOVE_SKILLS(P,r'')
```

Figure 3: An algorithm to remove unneeded skills

Dealing with the second possibility can be done with two different kinds of plan operations: we can *extend* the plan with a plan fragment or we can *replace a part* of the plan with a fragment. To gain information on how to include a fragment to the plan we introduce two functions: *Howto* which will return information on extending a plan with a plan fragment and *overlap* which will return how to replace part of the plan with a plan fragment. After detailing these functions in the next two sections, we present an algorithm which combines these functions to find a way to replace lost resources.

3.1 Howto

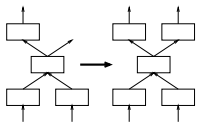

Figure 4: Extending a plan

If we need to provide a resource r in our planning graph, we can simply add the necessary skills to reach r (schematically depicted in Figure 4). For example, if we need to provide a truck in Baltimore, we could add a plan fragment that drives a truck from its current location to there. Possibly, some of the skills used in the plan fragment are not needed, since the original plan can provide the same resources as these skills. The *howto* function takes as input a plan P and a plan fragment PF and returns a tuple $\langle sk, res \rangle$, where sk is the set of skills that needs to be added to the plan, and res is the set of resources that have to be included in the plan before the fragment can be added (i.e. external resources required by the plan fragment PF, that can not be provided by the plan P).

The procedure (which is given in Figure 5) works as follows: We start by checking whether we need the plan fragment PF at all, i.e. if the goals PF produces are not already available in the plan P. In general this will not be the case, after

```
HOWTO(P,PF)
1. if all goals of PF reached in P then
       return⟨∅,∅⟩
2. ⟨sk,res⟩ = ⟨∅,∅⟩
3. for all skills s which produce goals in PF
   that are not available in P do
       let⟨sk',res'⟩ = CHECK_SKILL(s,P,PF)
       sk = sk ∪ sk'
       res = res ∪ res'
4. return⟨sk,res⟩
```

```
CHECK_SKILL(s,P,PF)
1. if s is marked then
       return⟨∅,∅⟩
2. MARK (s)
3. ⟨sk,res⟩ = ⟨s,∅⟩
4. for all resources r required by s do
       if r not available in P then
           if r produced by a skill s' in PF then
               let ⟨sk',res'⟩ =CHECK_SKILL(s',P,PF)
               sk = sk ∪ sk'
               res = res ∪ res'
5. return⟨sk,res⟩
```

Figure 5: Algorithm for *howto(P,PF)*

which we start checking the skills of PF from top to bottom. For each skill s, we check whether the resources it consumes ($in(s)$) are available in P. If there are any, we can use those and add the skill to the plan without a problem. If not, we will add the skills from PF that provide the missing resources, if there are. If there is no such skill in PF (the resource is an input for the plan fragment), we will add the resource to the list of missing resources. We will use $P \oplus sk$ to denote the extension of a plan P with a plan fragment PF, where $\langle sk, res \rangle = howto(P, PF)$ [2].

3.2 Overlap

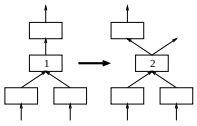

Figure 6: Replacing a skill

Besides extending a plan to provide a wanted resource, we can also try to change part of the plan (schematically depicted in Figure 6). For example, in the case of a broken truck we might consider using rail transport instead. To do this, we need to determine which skills in the original plan P can be removed, if a plan fragment PF is added. Some skills s that can be removed because of adding PF also produce side products that are used by other skills. These side products have to be provided in another way. We will now specify a function, *overlap*, which

[2] Since the structure of the plan fragment is lost, this takes a bit of reasoning. Saving extra data about the structure can prevent this.

will check if we can replace part of a plan P with a superskill S. The result of $overlap(P, S) = \langle sk, res \rangle$, where sk is the set of skills that can be removed from P, res the set of goals that will no longer be reached (more specifically: the resources that provide these goals) and S a superskill representation of a fragment.

The *overlap* algorithm (given in Figure 7) works as follows: For each output of the superskill that has a matching resource r in the plan, we walk through the planning graph to search for skills that can be removed and goals that will no longer be reached. We start with the skill s that produced r and check both the skills that rely on s, and the skills that s relies on. For each skill we check in this way, we will examine the produced and the consumed resources: A produced resource will be added to the set of needed resources if it is not produced by the superskill S and is a goal of P, for consumed resources $r \in In(P)$ we mark the corresponding input of S (if there is any). When this search ends, we will add the unmarked input resources of S to the set of needed resources too, completing the set. We will use $P \otimes sk$ to denote replacing part of a plan P with a superskill S in this way, where $\langle sk, res \rangle = overlap(P, S)$.

OVERLAP(P,S)
1. $\langle sk, res \rangle = \langle \emptyset, \emptyset \rangle$
2. **for all** *products of S find the corresponding resource(s)* $r \in P$ *and the skill* s *that produces it* **do**
   if s *is not marked* **then**
   let $\langle sk', res' \rangle =$SKILLS$(P, S, s)$
   $sk = sk \cup sk'$
   $res = res \cup res'$
3. $res = res \cup$ *unmarked resources of* S
4. **return**$\langle sk, res \rangle$

SKILLS(P,S,s)
1. $\langle sk, res \rangle = \langle s, \emptyset \rangle$
2. **for each** *unmarked resource* r *produced by* s **do**
   let $\langle sk', res' \rangle =$RESOURCE_UP$(P,S,r)$
   $sk = sk \cup sk'$
   $res = res \cup res'$
3. MARK(s)
4. **for each** *unmarked resource* r *consumed by* s **do**
   let $\langle sk', res' \rangle =$RESOURCE_DOWN$(P,S,r)$
   $sk = sk \cup sk'$
   $res = res \cup res'$
5. **return**$\langle sk, res \rangle$

RESOURCE_UP(P,S,r)
1. $\langle sk, res \rangle = \langle \emptyset, \emptyset \rangle$
2. **if** r *is produced by* S **then**
   MARK(r) *in* S, **return**$\langle \emptyset, \emptyset \rangle$
3. **if** r *is a goal of* P **then**
   $res = r$
4. MARK(r)
5. **if** r *is consumed by a skill* s *which is not marked* **then**
   let $\langle sk', res' \rangle =$SKILLS$(P, S, s)$
   $sk = sk \cup sk'$
   $res = res \cup res'$
6. **return**$\langle sk, res \rangle$

RESOURCE_DOWN(P,S,r)
1. **if** r *is an input of* S, *or* r *is an initial condition of* P **then**
   MARK(r) *in* S, **return**$\langle \emptyset, \emptyset \rangle$
2. MARK(r)
3. **return** SKILLS$(P, S,$ *skill that produces* r *in* $P)$

Figure 7: The *overlap* function

3.3 Combining *howto* and *overlap*

To solve the problem of including a resource in the plan, we will use an algorithm based on the *howto* and *overlap* functions. The algorithm will examine a number of possible fixes in a breadth-first way. We will use a priority queue, prioritized by the *cost* of a possible solution to decide which partial solution we will work on. We define the cost of a possible solution PF^*, which is a sequence of applied plan fragments, to solve a problem in plan P as follows: let $new(PF^*)$ be the number of new skills that will be added to P if we add PF^* to P, $changes(PF^*)$ be the

number of changes (additions and removals of skills) to P if we add PF^*, and $missing(PF^*)$ the number of resources that need to be found before we can add PF^*. Then, $cost(PF^*) = x_1 new(PF^*) + x_2 changes(PF^*) + x_3 missing(PF^*)$, where $x_1 + x_2 + x_3 = 1$. By setting the values of x_i we choose priorities for the different aspects of partial solutions. Suppose we want to favor solutions with few missing resources[3] and solutions with additions over solutions with changes[4]. Then, we use $x_3 > x_1 > x_2$.

The algorithm to find a way to produce a resource g is shown in Figure 8. It starts by checking if there is already a resource of the same resource scheme as g in the plan (step 1). If not, we will first add all plan fragments PF with $g \in out(PF)$ to the queue. We then select the first element in the queue (one of the partial solutions with least cost), find the ways to include a missing resource for that plan fragment, and queue the extended solutions. We then select the new front element of the queue and process it, until we have found a solution to our problem, or we cross some predefined threshold (to make sure we do not search forever trying to include a resource). This greedy algorithm can be used to build

```
NEWGOAL(P,g)
1. if ∃r∃R · r ∈ out(P) ∧ r ∈ R ∧ g ∈ R then
       use r to supply g
2. else
       for all PF where g ∈ out(PF) do
           let ⟨sk, res⟩ =OVERLAP(P, PF)
           if sk ≠ ∅ then
               QUEUE(P ⊗ sk,PF,res,COST(PF))
           else
               let ⟨sk, res⟩ =HOWTO(P, PF)
               QUEUE(P ⊕ sk,PF,res,COST(PF))
3. while no complete solution found and the
   threshold is not crossed do
       DEQUEUE(P,PF*,wanted,cost)
       select a random r ∈ wanted
       if ∃r'∃R · r' ∈ out(P) ∧ r ∈ R ∧ r' ∈ R
       then
           use r' to supply r
       else
           for all PF where r ∈ out(PF) do
               let ⟨sk, res⟩ =OVERLAP(P, PF)
               if sk ≠ ∅ then
                   QUEUE(P ⊗ sk,PF* + PF,res \
                   r,COST(PF* + PF))
               else
                   let ⟨sk, res⟩ =HOWTO(P, PF)
                   QUEUE(P ⊕ sk,PF* + PF,res \
                   r,COST(PF* + PF))
```

Figure 8: An algorithm that combines *howto* and *overlap* to include a new goal

a more optimized one, by taking into account domain specific knowledge.

The algorithms described above have been implemented and tested on a number of transportation problems. The program reads a planning graph and a number of plan fragments to start with, after which the user can select from a number of problems to happen. The program then searches through the list of fragments to fix the plan with. The program is able to extend plans with a number of fragments to reach new goals and can also replace skills by others.

4 Conclusions and future extensions

We have remarked that a plan that is created under the assumptions of deterministic effects and sole cause of change may not remain valid in a dynamic environment. Other agents and failing actions may bring an agent in a situation where his plan becomes obsolete or inefficient. If this happens the agent has to create a new valid plan. One approach is to start planning all over again, but a better approach is to

[3] The number of missing resources could be seen as a measurement for the remaining work.

[4] Additions won't interfere with the current plan, whereas changing the plan might.

adapt the current plan to fit the new situation. We have developed a framework to represent an agent's actions and introduced algorithms for removing and adding resources to the plan, based on the functions *remove_skills*, *howto* and *overlap*, that can adapt a plan to the new situation by extending or changing the plan. Although the revision algorithms correctly solve the problems of including and removing resources from the plan, there are a number of improvements possible:

- The framework presented does not yet include timing information. Often, an agent has to reach the goals before a certain deadline. To be able to deal with deadlines, the framework and the algorithm will have to be changed to take times into account.
- The cost function used, treats all goals and skills equally. More likely, different skills have different costs, and different goals have different profits. Also, the cost function used by the algorithm contains three constants, for which an optimal value (or range) has to be found.
- If there are multiple agents, it is possible that the help of another agent is required, or leads to a cheaper solution. A way to inquire and deal with this information will be needed in multi-agent environments.
- Special tools could be developed, which create a suitable set of plan fragments. The set of plan fragments should be large enough to be able to construct plan adaptations with a few plan fragments, but it should not be too large, since that will slow down the search.

References

[1] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. In *Proc. of the* 14^{th} *Int. Joint Conf. on A.I. (IJCAI-95)*, 1995.

[2] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *European Conference on Planning (ECP-99)*, 1999.

[3] A. Gerevini and I. Serina. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proc. of the* 5^{th} *Int. Conf. on AIPS*, 2000.

[4] S. Hanks and D.S. Weld. A domain-independent algorithm for plan adaptation. *JAIR*, Volume 2, 1995.

[5] H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98*, 1998.

[6] L.J. Moree, A. Bos, H. Tonino and C. Witteveen. Cooperation by iterated plan revision. In *Proceedings of the ICMAS-00*, 2000.

[7] D.S. Weld. Recent advances in AI planning. In *AI Magazine*, Volume 20, Number 2, 1999.