A Resource Based Framework for Planning and Replanning

Roman van der Krogt* Mathijs de Weerdt†
Cees Witteveen

Delft University of Technology,
P.O.Box 5031, 2600 GA Delft,
The Netherlands
{r.p.j.vanderkrogt, m.m.deweerdt, c.witteveen}@ewi.tudelft.nl

Abstract

The aim of this paper is to combine standard planning and replanning methods into a rigorous unifying framework, extending an existing logic-based approach to resource-based planning. In this Action Resource Framework (ARF), actions and resources are the primitive concepts. Actions consume and produce resources. Plans are structured objects composed of actions and resource schemes and an explicit dependency function specifying their interrelationships.

Previous plans can be used both for creating new plans and for modifying plans. Since we are often interested in reusing only a part of these previous plans, we extend the Action Resource Formalism with *incomplete plans*. To efficiently represent such incomplete plans we use the notion of a *gap*. We maintain a library of incomplete plans (with gaps), and we present operators to insert plan parts from this plan library into the current plan. We prove this set of plan operators to be complete.

Generalizing the refinement planning approach, we present a template algorithm for both planning and replanning using the ARF and its plan library and plan operators. Finally, we show that existing (re)planning methods and heuristics nicely fit into this framework.

1 Introduction

Planning is an important ability required for intelligent agents to accomplish goals they have set for themselves, or to reach goals they have accepted from others. Usually, a planning problem is specified using a description of (i) the *current* (or *initial*) *state* the agent is in, (ii) the set of *actions* (together with their prerequisites and consequences) the agent is capable to perform, and (iii) the *goals* the agent is aiming at, specified as a set of states. The planning *problem* then is to find the right sequence (or partial order) of actions leading the agent from the initial state to one of the desired states specified by the goals.

*Supported by the Freight Transport Automation and Multi-Modality program of the TRAIL research school, Delft University of Technology.

†Sponsored by the Seamless Multi-modal Mobility Program of the TRAIL research school, Delft University of Technology.

1.1 Existing approaches

Currently, many systems and methods exist that try to tackle the planning problem (e.g. [1, 2, 4, 6, 9, 17, 29]). The techniques used by these planners can be quite different. For example, planners like HSP [3] and FF [18] construct plans by using a *heuristic search* method. Other, so-called *least-commitment* planners, keep track of all dependencies and constraints among actions within their partial plan representation. No matter what method is used, however, these systems have some serious drawbacks.

First of all, almost all of these planning systems rely on the tacit assumption that planning problems can be solved *off-line*: the goal and the initial state are assumed to remain unchanged, at least during the planning process. For a large number of domains, however, this assumption does not hold due to the dynamical nature of planning problems. For example, in typical transportation problems, planners cannot rely on plans that are constructed off-line: the feasibility of the plan can be easily and seriously affected by, e.g., traffic accidents, broken equipment or last-minute changes in orders. Hence, in these cases, a practical planning approach should also pay attention to *replanning* in order to to repair a plan that has become infeasible, possibly already during the planning phase.

Secondly, most planning systems assume that a planning agent has to start from scratch. Often, however, this assumption is not realistic: agents are able to use results of their previous planning experiences, or knowledge given to them by a (human) domain expert and, therefore, use one or more existing –maybe incomplete or not completely suitable– plans P' as a starting point. Such initial plans may be *transformed* and *combined* into a suitable plan. Therefore, the standard approach to planning, i.e., to start from an empty plan, seems to be just a *limiting case* of standard practice and usually needs to be generalized to include the adaptation of existing plans.

Fortunately, there are planning systems that at least partially deal with these issues. For example, the Systematic Plan Adaptor (SPA) [17] and CHEF [16] systems meet the second objection by addressing plan adaptation. These *case-based planners* maintain a database of past problems and their solution plans, and choose an appropriate starting plan whenever they face a new planning problem. The selection of the starting plan is based on the similarity to the new planning problem. This plan then is modified to match the current goal and initial state requirements. Case-based planning has been successfully applied to a wide range of problems, including logistics (e.g. the PRODIGY planner [26]), software reuse (e.g. MRL [21]) and navigation planning (e.g. ROBBIE [13]). However, none of these systems explicitly deals with replanning. (Although they could be used for it by taking the current, invalid plan as a starting point and modify it.)

Other systems focus on the replanning aspect. Systems such as GPG [14], O-Plan [8] and more recently the system described by Ferrer [11] are all *static* replanners, i.e., whenever their system is used to solve a discrepancy between the original planning problem and the current state of the world, it assumes that the world remains static. Only after a solution is found, the state of the world is examined again for changes.

The term *continual planning* [7, 23, 15] is used to refer to systems in which planning, replanning and execution are all continuously interleaved. These sys-

tems provide a comprehensive approach to planning in dynamic environments. However, here planning and replanning are seen as two different issues and no past knowledge is used.

1.2 Research contributions

The goal of this paper is, first of all, to bring together ideas from both standard planning and replanning approaches into a unifying (re)planning approach. We base this framework upon an existing logic-based framework for resource based planning. The outcome is a unified framework to represent planning and replanning.

Secondly, the use of plan libraries is discussed as a method to repair plans based on previous planning experiences. To efficiently store plans in a library, we introduce the concept of plans with *gaps*. Plan operators are introduced that can manipulate the current plan using plans from the library.

Thirdly, we show that refinement strategies can be built on top of this framework to supply computational support for (re)planning and plan repair. To show the validity of the framework, we specify how two existing approaches can be described by it.

This paper is organized as follows. First, we give a concise introduction to an existing resource-based planning framework [24]. We then use this formalism to deal with replanning and we introduce plan transformation operators that are able to modify resource-based plans. We show that all necessary plan adaptations can be performed by a single plan adaptation operator. By assuming that an agent is able to use a *plan library*, these operators can be used to transform an initial (inadequate) plan into an adequate plan. This framework unifies both planning and replanning approaches. We would like to find a suitable sequence of these plan modification steps to obtain an adequate plan form the initial plan. Therefore, we present a method to use existing planning techniques in this framework, and illustrate this by showing that the FF-approach [18] and SPA [17] algorithm for plan adaptation can be reformulated in this framework.

2 The action resource formalism

In current planning formalisms such as STRIPS [12] and PDDL [22], several propositions are required to describe the properties of one object. When a resource in the planning domain unexpectedly becomes unavailable, the consequences for the set of propositions is not always directly clear. Therefore, we introduce a more object-oriented view to planning where the relevant planning entities, called *resource facts*, are the primitive concepts. These resource facts constitute an encapsulation of the properties of a real entity, such as a truck, a cab, a person, or a package. This resource oriented view is the starting point of the *Action Resource Framework*, abbreviated ARF, and improves previous work on resource based planning [5, 24].

We start with a concise overview of the main elements of this framework. Subsequently, we extend the framework by providing a more sophisticated notion of plans, and we formalize the notion of incomplete plans in the form of plans with *gaps*.

2.1 Basic notions

In the ARF two basic notions are distinguished: *resource (facts)* and *actions*. Goals and plans are derived notions that are defined using resource facts and actions.[1]

An atomic *resource fact* is the concise description of an object that is relevant to an agent with respect to the planning problem at hand. Such a resource is either a description of a physical object such as a truck or a block, or an abstract conceptual notion such as the right to do something.[2] Syntactically, a resource fact is denoted by a *predicate name* together with a complete specification of the *sorts* of all its *attributes* together with their *values*. The predicate name serves to indicate the *type* of resource mentioned in the fact. For example, we could represent a bicycle by a resource of type *bicycle*, having attributes like its identifier *id* and a location *loc*. Then $bicycle(1 : id, \mathrm{A} : loc)$ is a resource fact describing a cycle located in A with identifier 1. Since similar resources (i.e., resources with the same type and the same values for their attributes) may occur multiple times in the same plan, if needed, we will add to each resource a unique *occurrence identifier*, a special attribute that is used exclusively to distinguish between different occurrences of similar resources in a plan. A resource of type t with i as (the value of) its identifier is denoted as $t_i(\ldots)$.

Values of attributes may be ground (i.e., constant), but may also be variables or functions. In the latter case, a resource fact describes a *set* of ground resource facts (instances) of the same resource type. For example, the following resource refers to all bicycles in location A: $bicycle_2(i : id, \mathrm{A} : loc)$.

To specify one specific ground resource fact denoted by such a *general resource fact*, we introduce the notion of a *substitution*. A substitution θ replaces variables occurring in a resource r by terms of the appropriate sort. We write $r\theta$ to denote the resource r' that results from replacing the variables occurring in r according to θ. A substitution is *ground* if it replaces variables by ground terms, i.e., terms that do not contain variables. If R is a set of general resources, $R\theta$ is a shorthand for $\{r\theta \mid r \in R\}$.

A set of *goals* G is specified by a set of general resources $G = \{g_1, \ldots, g_n\}$. We say that a set of goals G is *satisfied* by a given set of resources R, abbreviated by $R \models G$, if there exists a ground substitution θ such that $G\theta \subseteq R$, i.e., there is a set of ground instances of the goals that is provided by the resources in R. Two resources r_1 and r_2 are called *compatible*, denoted by $r_1 \equiv r_2$, when they are equal except for the value of their occurrence attribute.

Resource facts are used to specify the state of the world (as far as it is relevant) by enumerating the set of resource facts that are true at a certain point of time. Possible *transitions* from one state to another are described by *actions*. An action is a basic process that consumes and produces resources. An action o has a set of input resources $in(o)$ that it consumes, and a set of output resources $out(o)$ it produces. Furthermore, an action may contain a specification $param(o)$ of some variables occurring in the set of output resources as *parameters* of the action. To ensure that output resources are uniquely defined, these resources may only contain variables that already occur in the

[1] In the original formalism [5], actions are called 'skills' and plans are called 'services'. We have chosen to use the more generally accepted terms 'actions' and 'plans'.

[2] Abusing language, in the sequel we use the notions of a resource and a resource fact interchangeably.

input resources or in the set of the parameters.

Example. An example of an action is:

$$pedal(d : loc) : \ bicycle(i : id, s : loc), road(s : loc, d : loc) \Rightarrow \\ bicycle(i : id, d : loc), road(s : loc, d : loc)$$

This action specifies how a person can travel (by using a cycle) from a source location s to a destination d. This action requires a cycle with identifier i at a source location s, and a road between s and d and "produces" a cycle at the destination and the road again (to make it available to other actions).[3]

An action o can be applied to a set of (ground) resources R if there exists a ground substitution θ such that $in(o)\theta \subseteq R$. Application of this action to R results in consuming the set $in(o)\theta$ of input resources while producing the set $out(o)\theta$: starting with R, the set $R' = (R \setminus in(o)\theta) \cup out(o)\theta$ is produced.

Let O be a set of actions.[4] The set of consumed resources by actions in O is specified by $cons(O) = \bigcup_{o \in O} in(o)$, while the set of resources produced is $prod(O) = \bigcup_{o \in O} out(o)$. The set of all resources mentioned in O is $res(O) = cons(O) \cup prod(O)$.

2.2 Plans

In general, a single action applied to an initial set of resources is not sufficient to achieve a desired state. Often, actions have to be applied in a partial order to produce the desired effect. A specification of the ordering of actions, however, is not sufficient. We also need to specify for each consumed resource, which produced resource it is *dependent* upon. Such a partially ordered set of actions together with a specification of the resource dependency relation is called a *plan*. To specify how actions are interrelated, we use the notion of a *dependency function*:

Definition 1. *Let O be a set of actions. A* dependency function *is an injective function $d : cons(O) \to prod(O) \cup \{\bot\}$ specifying (in a unique way) for each resource r to be consumed which resource r' produced by another action is used to provide r and undefined (denoted by $\bot$) else, i.e., if r is not produced by an action in O).*[5]

As we mentioned before, plans are composed of partially ordered actions. Since a dependency function d specifies an immediate dependency of input resources of an action on output resources of another action, d can only specify a *valid* dependency if *(i)* the resources r and $d(r)$ are compatible and *(ii)* d generates a *partial order* on the set of actions occurring in the plan.

[3] Note that in this particular model of the world we allow only one agent to use the road at a time. In a more realistic situation, we could attach a *capacity* attribute to a road predicate and allow it to be consumed by a number of actions until its capacity limit has been reached. However, such a more realistic model would make the example unnecessary complicated.

[4] In the remainder of the paper, we assume that for each o, o' in O we have $\mathrm{var}(o) \neq \mathrm{var}(o')$ when $o \neq o'$.

[5] Sometimes we need the inverse d^{-1} of d, which is defined as follows: for every $r' \in prod(O)$ such that $d(r) = r'$, $d^{-1}(r') = r$, and for every $r' \in prod(O)$ not occurring in $\mathrm{ran}(d)$, $d^{-1}(r') = \bot$.

The first requirement is met if there exists a substitution θ such that for any two resources r and r', $d(r) = r'$ implies $r\theta \equiv r'\theta$. That is θ is a *unifier* for every pair of resources $(r, d(r))$. In particular, we are looking at a *most general unifier* (mgu) θ with this property.

The second condition requires that there are no loops in the dependency relation between actions generated by d: we say that o directly depends on o', abbreviated as $o' \ll_d o$, if resources $r \in in(o)$ and $r' \in out(o')$ exist such that $d(r) = r'$. Let $<_d = \ll_d^+$ be the transitive closure of $\ll_d$. Then the second condition simply requires $<_d$ to be a (strict) partial order on O.

Now a plan can be defined as follows.

Definition 2. *A* plan P *over a set of actions* O *is a triple* $P = (O, d, \theta)$ *where* d *is a dependency function specifying dependencies between compatible relations and generating a partial order* $<_d$ *on* O*, while* θ *is the mgu of all dependency pairs* $(r, d(r))$ *where* $r, d(r) \in res(O)$*. The empty plan* $(\emptyset, \emptyset, id)$ *is denoted by* $\not\Diamond$*.*

Given a plan $P = (O, d, \theta)$, the set $In(P) = \{r\theta \mid r \in cons(O), d(r) = \bot\}$ is the set of input resources of P, i.e., the set of resources not depending on other resources in the plan. Analogously, the set $Out(P) = \{r\theta \mid r \in prod(O), d^{-1}(r) = \bot\}$ of output resources of P is the set of resources that are not consumed by actions in the plan. Furthermore, we define $prod(P) = prod(O)\theta$ and $cons(P) = cons(O)\theta$. Note that of course resources from $prod(P)$ may be consumed by *other* plans to produce resources.

Finally, we use $||P||$ to denote the size of P which equals the sum of the total number of all (occurrences of) resources mentioned in P and the number of actions occurring in P.

Plans are used to transform a set of resources into another set of resources. Given a set of goals G and an initial set of resources I, a plan $P = (O, d, \theta)$ is *applicable* to I if there exists a substitution σ such that $In(P)\sigma \subseteq I$. P *realizes* a set of goals G if there exists a substitution σ such that

$$G\sigma \subseteq (I \setminus In(P)\sigma) \cup Out(P)\sigma$$

The tuple (I, P, σ, G) where σ is a substitution of variables occurring in the goals G and input resources $In(P)$ is called an *embedded plan*. An embedded plan (I, P, σ, G) is called *adequate* when $In(P)\sigma \subseteq I$ and $G\sigma \subseteq I \setminus In(P)\sigma \cup Out(P)\sigma$. A tuple (I, P, G) is called adequate, denoted by $I \models_P G$, if, for some substitution σ, (I, P, σ, G) is adequate.

Example. Consider the plan shown in Figure 1. Here, the dependency function d is specified by $d(bicycle(i_2, f_2)) = bicycle(i_1, t_1)$ and undefined for the remaining input resources. Note that the substitution θ should ensure that the two cycle resource in the plan are similar. Let θ be the mgu $\theta = \{i_2 := i_1, f_2 := t_1\}$. This plan $P = (\{o_1, o_2\}, d, \theta\}$ can be used with an initial set of resources $I = \{road(\mathrm{A} : loc, \mathrm{B} : loc), road(\mathrm{B} : loc, \mathrm{C} : loc), bicycle(1 : id, \mathrm{A} : loc)\}$ to satisfy a goal $G = \{bicycle(1 : id, \mathrm{C} : loc)\}$ by using a substitution $\sigma = \{f_1 = \mathrm{A}, t_1 = \mathrm{B}, i_1 = 1, f_2 = \mathrm{B}, t_2 = \mathrm{C}\}$ since $In(P)\sigma = I$ and $G\sigma \subset Out(P)\sigma$.

2.3 Plans with gaps

The plans discussed above are perfect plans: whenever an instance of $In(P)$ has been determined, an instance of $Out(P)$ is guaranteed to be produced by

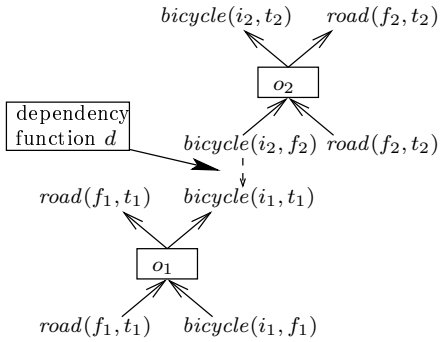


Figure 1: An example plan.

executing the actions as specified in P. It may occur, however, that at some places in the plan we do not exactly know how to transform a set of resources into another set. In these cases a plan contains one or more *undefined actions*. Like a real action, an undefined action u specifies a relation between a set of its inputs $in(u)$ and its output resources $out(u)$. Such an action u is called undefined with respect to a set of actions $\mathcal{O}$ if there is no (single) action $o \in \mathcal{O}$ and substitution θ such that $in(u) \supseteq in(o)\theta$ and $out(u) \subseteq out(o)\theta$. We call such an action u a *gap* (over $\mathcal{O}$).

Definition 3. *$P = (O \cup U, d, \theta)$ is a* plan with gaps *over $\mathcal{O}$ if P is a plan over $O \cup U$, $O \subseteq \mathcal{O}$ and every action $u \in U$ is a gap over $\mathcal{O}$.*

The idea of a plan with gaps is that it can be extended to a plan without gaps by substituting (other) plans for undefined actions. To this end we need the notion of *fitting* into a plan:

Definition 4. *Let $P = (O \cup U, d, \theta)$ be a plan with a set U of gaps over $\mathcal{O}$, and let $P' = (O', d', \theta')$ be a plan. If, for some $u \in U$ and a substitution σ, P' realizes $out(u)\sigma$ using (a subset of) $in(u)\sigma$, then P'* fits into *P and $P'' = (O \cup O' \cup (U \setminus \{u\}), d'', \theta\theta'\sigma)$ is a plan with gaps over $\mathcal{O}$ using P' as a sub plan, if d'' satisfies the following:*

$$d''(r) = \begin{cases} d(r) & \text{if } r \in cons(P) \setminus in(u) \text{ or } d(r) \in prod(P) \setminus out(u) \\ d'(r) & \text{if } r \in cons(P') \text{ and } d'(r) \neq \bot \\ r' & \text{if } r' \in Out(P'),\ d(r) \in out(u) \text{ and} \\ & r'\theta'\sigma \equiv d(r)\theta\sigma \\ r'' & \text{if } r \in In(P'),\ r' \in in(u) \text{ and} \\ & r'\theta\sigma \equiv r\theta'\sigma \text{ and } d(r') = r'' \\ \bot & \text{otherwise.} \end{cases}$$

This concludes the introduction to ARF and its extension to plans with dependency functions and plans with gaps. In the next section we use this (extended) ARF framework to discuss planning and replanning problems, showing that both can be defined as subproblems of a more general plan transformation problem, and we show how such transformations can be accomplished.

3 Planning and replanning

Using the ARF terminology, traditional planning problems can be easily defined. A planning problem is a tuple $\Pi = (\mathcal{O}, I, G)$ where

1. I is a finite set of ground resources specifying the initial situation,
2. $\mathcal{O}$ is the set of possible actions an agent is capable to execute, and
3. G is a finite set of general resources specifying the goals.

A *solution* to Π is a plan $P = (O, d, \theta)$ such that

1. $O \subseteq \mathcal{O}$, i.e., it contains actions the agent can execute, and
2. (I, P, G) is adequate, i.e., $I \models_P G$.

Typically, traditional planning approaches implement various ways of searching for a (partially ordered) sequence of actions that leads the agent from a state I to a state that fulfills G. As remarked in the introduction, the search for a plan from scratch should be considered a limiting case of planning. For, in most cases, agents start from some, although not completely adequate plan P' instead of starting from the empty plan $\lozenge\!\!\!\!\!\!\diamond$, and then try to *transform* it into an adequate plan. Therefore, we consider planning to be an activity where existing plans are transformed and combined into adequate plans. This perspective also enables us to consider planning and replanning as almost identical problems. First of all, note that a *replanning* problem occurs if an agent is able to achieve a set of goals G using a plan P with initial resources I (i.e., the triple (I, P, G) is adequate), but due to changes the agent discovers that its actual set of available resources is I', the realizable part of its plan is P', the actual set of goals is G' and the triple (I', P', G') is no longer adequate. Hence, its current plan P' has to be adapted.

Now observe that both instances of the planning and the replanning problem can be defined by

(i) the availability of an adequate triple (I, P, G),

(ii) the existence of an inadequate triple (I', P', G'), and

(iii) the goal of obtaining an adequate triple (I', P'', G').

In a *planning* problem, (i) denotes the availability of a plan P that has been used previously in a resource context (I, G). This plan $(P' = P)$ is proposed in the current resource context (I', G') as a starting point (ii). However, by (iii), P has to be transformed to an adequate plan P''.

In a *replanning* problem, (i) means that there was a valid plan for the initial resource context (I, G), but due to changes in the initial state, the action set and/or the goal specification, we end up with an invalid plan P' (ii) that has to be transformed into a valid plan P'' in the (changed) context (I', G') (iii).

Obviously, in most cases, such a transformation consists of several smaller transformation steps. Therefore, both planning and replanning can be described as constructing *sequences* of plan transformation steps. Of course, these transformation steps are guided by the available knowledge of the agent. Here, we propose to represent this knowledge by a set of available *free* plans in the form

of a *plan library* that can be used by the agent to transform existing plans.[6] A transformation step then involves the application of some plan transformation operator on the existing plan and some free plan selected from the plan library to compose a new plan.

In the next subsections we first discuss a number of such plan transformation operators and we show that this set of operators is complete with respect to (re)planning using a plan library. Thereafter, we discuss some details of the plan library.

3.1 Plan operators

We introduce two simple plan operators (addition and deletion) that are used to transform a plan P using some set $\{P_j\}_{j=1}^n$ of given plans (the plan library). We show that assuming some simple properties of the plan library, they form a *complete* set, that is every (source) plan P can be transformed into a target plan P' applying only these operators. Then we introduce a new plan transformation operator (replacement) and we show that this operator is able to simulate the two operators already introduced. Moreover, we provide a lower bound L for the number of plan transformations required –whatever operators used – given a source plan P, a target plan P' and a plan library $\{P_j\}_{j=1}^n$ and we prove that there always exists a sequence of transformations requiring no more than $2L$ applications of the replacement operator.

3.1.1 Addition

Analogously to the action concatenation operator used in traditional planning to construct a plan by composing actions, the addition operator $\oplus$ is an operator that *glues* two plans together by connecting input resources to output resources. Clearly, $\oplus$ needs the specification of a *glue function* g, like the dependency function d we used in a plan, to specify how exactly the new dependencies between input and output resources in both plans are created. This glue function g is a partial function overriding the specifications of the existing dependency functions d_1 and d_2 in both plans.[7] More precisely, if P_1 and P_2 are two plans to be added, whenever $r \in In(P_i)$ occurs in $\mathrm{dom}(g)$, $g(r) \in Out(P_j)$ for $i \neq j \in \{1, 2\}$.

Definition 5 (Addition). *Let $P_1 = (O_1, d_1, \theta_1)$ and $P_2 = (O_2, d_2, \theta_2)$ be plans and $g : In(P_1) \cup In(P_2) \rightarrow Out(P_1) \cup Out(P_2)$ a partial dependency function mapping input resources to output resources. Then $P_1 \oplus_g P_2$ is defined as the plan $P = (O, d, \theta)$ where*

1. *$O = O_1 \cup O_2$,*
2. *$d = (d_1 + d_2) \dagger g$ is a valid dependency function, and*[8]
3. *$\theta = \theta_1\theta_2\theta_g$ where θ_g is the mgu of the set of pairs $\{(r\theta_1\theta_2, g(r)\theta_1\theta_2) \mid r \in \mathrm{dom}(g)\}$.*

[6] Free plans are plans not embedded in a context.

[7] Formally, the result of overriding f by g, denoted as $f \dagger g$ is the function h where $h(x) = g(x)$ if $x \in dom(g)$ and $h(x) = f(x)$ else.

[8] That is, $<_d$ is a strict partial order.

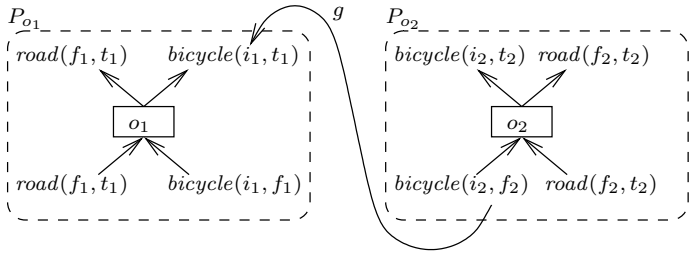


Figure 2: Adding two actions to obtain the plan from Figure 1.

We discuss two extreme cases:

1. If $\mathrm{dom}(g) = \emptyset$, P is the simple *concurrent combination* of P_1 and P_2, that is $P_1 \oplus_g P_2 = (O_1 \cup O_2, d_1 + d_2, \theta_1 + \theta_2)$;

2. If $\mathrm{dom}(g) \neq \emptyset$ and $P_2 = \diamondsuit$, the resulting plan P is a *sequential refinement* of P_1, that is $<_d$ extends the partial order $<_{d_1}$.

Remark. Let $var(P)$ denote the set of variables mentioned in the actions occurring in the set O of actions of P. We always assume that whenever two plans P_1 and P_2 are combined, $\mathrm{var}(P_1) \cap \mathrm{var}(P_2) = \emptyset$.

Example. Suppose that we would want to create the plan P from Figure 1 from two plans P_{o_1} and P_{o_2}, each containing just the action o_1 and o_2, respectively. This situation is depicted in Figure 2. A glue function g that would achieve the desired result is $g(bicycle(i_2, f_2)) = bicycle(i_1, t_1)$. Now, $P = P_{o_1} \oplus_g P_{o_2}$.

We state a simple property of the addition operator:

Proposition 6. *If $P_1 = (O_1, d_1, \theta_1)$ is a subplan of $P = (O, d, \theta)$ then there always exists a dependency function g such that $P = P_1 \oplus_g P_2$ for some subplan $P_2 = (O \setminus O_1, d_2, \theta_2)$ of P.*

Proof. Note that d_2 can be specified as the restriction of the dependency function d to the set $O \setminus O_1$. Then g is the partial dependency function consisting of those pairs (r, r') that do occur in d but not in d_1 or d_2. □

3.1.2 Deletion

Addition extends a plan and is needed, e.g., in order to obtain new (goal) resources that are available in a plan from the plan library. Sometimes, however, the resources we need are already available in a plan P, but are not available as output resources because they are consumed by some action. In that case we like to free those resources and even are prepared to delete some actions depending on these resources.[9] We therefore define an operator $\ominus$ that is able (i) to free resources and (ii) to delete actions by updating the dependency function using

[9] The deletion operator can be applied to a plan in the plan library, or directly to the current plan to be transformed.

a partial dependency function g in a plan P and by specifying the set of actions to be removed from P.

More specifically, $\ominus$ takes two plans P_1 and P_2 where P_2 is a subplan of P_1 specifying the actions to be removed from P_1, and a partial dependency function g. The purpose of g is to specify which dependencies occurring in d have to be removed additionally, in order to free resources in P_1. Hence, the domain of g consists of resources occurring in the input set $in(o)$ of some actions in the given plan P_1, mapping them to $\perp$ and thereby overriding the definition of d.

Definition 7 (Deletion). *Let $P_1 = (O_1, d_1, \theta_1)$ be a plan and $P_2 = (O_2, d_2, \theta_2)$ be a subplan of P_1. Let g be a partial function such that $(r, \perp) \in g$ and $\mathrm{dom}(g) \subseteq \mathrm{dom}(d_1)$. The plan $P = P_1 \ominus_g P_2$ then is defined as the plan $P = (O, d, \theta)$ where*

1. *$O = O_1 \setminus O_2$; that is all actions in O_2 are removed.*
2. *$d = ((d_1 - d_2) \dagger h) \dagger g$ where h is the dependency function defined as $h(r) = \perp$ whenever $d(r) \in res(O_2)$.*
3. *θ is the mgu of all pairs $(r, d(r'))$ occurring in d.*

We distinguish two extreme cases:

1. $P_2 = \lozenge\!\!\!\!\!\!\diamond$. Then $\ominus_g$ acts as an operator only freeing up resources by removing dependencies between resources in plans, without removing actions from P.
2. When $\mathrm{dom}(g) = \emptyset$ and $P_2 \neq \lozenge\!\!\!\!\!\!\diamond$, the resulting plan is the plan P_1 after removing the actions and dependencies from P_2.

Example. Let P be the plan of Figure 1. The plans P_{o_1} and P_{o_2} mentioned in the previous example may be obtained from P as follows: $P_{o_1} = P \ominus_\emptyset (o_2, \emptyset, id)$ and $P_{o_2} = P \ominus_\emptyset (o_1, \emptyset, id)$.

As expected, there exists a partial duality between the operators $\oplus$ and $\ominus$. Formally, the relation between $\ominus_\emptyset$ and the operator $\oplus$ can be described as follows.

Proposition 8. *Suppose that a plan P consists of two sub plans P_1 and P_2 such that P can be written as $P = P_1 \oplus_g P_2$ for some gluing function g. Then it holds that $P \ominus_\emptyset P_2 = P_1$.*

Proof. Immediately from the definitions of $\oplus$ and $\ominus$. □

3.2 Completeness of the addition and deletion operators

An important question is whether the two plan operators we just discussed are complete in the sense that every existing (inadequate) plan can be transformed into an adequate plan by applying just a sequence of plan addition and plan deletion operators.

Clearly, this completeness property is relative to the properties of the plan library that can be used to perform addition and deletion. We therefore assume the following properties to hold for the set of plans $\{P_j\}_{j=1}^m$ occurring in the plan library:

Assumption 1

Every plan is completely *accessible*, that is, for every plan in the plan library, its complete internal structure is available to the planner and the planner is always capable to retrieve any subplan from it.

Assumption 2

Every action o that belongs to the set of actions $\mathcal{O}$ that a planning agent A is capable of, occurs in at least one plan occurring in its plan library.

We show that if these two (mild) assumptions are met, every existing plan P can be transformed into an adequate plan P' using only the addition and deletion operators.

First of all we define the notion of a (sub) plan of a given plan generated by some subset of actions:

Definition 9. *Let $P = (O, d, \theta)$ be a plan and $O' \subseteq O$. Then the subplan of P generated by O', denoted by $P_{O'}$ is the plan $(P', d \restriction res(O'), \theta')$ where $d \restriction res(O')$ is the function d restricted to $res(O')$ and θ' is the mgu of all the pairs $(r, d(r))$ with $r \in \mathrm{dom}(r \restriction res(O'))$.*

First, we show a simple result stating that by using a series of deletion operators we always can extract from any plan P containing an action o a subplan P_o containing exactly the action o.

Proposition 10. *Let P be a plan containing an action o. Then there exists a sequence of plan operators resulting in the plan P_o.*

Proof. Since P contains o, P can be written as the composition of three plans:

1. The subplan $P_1 = (O_1, d_1, \theta_1)$ of P generated by all the actions the input resources occurring in $in(o)$ are dependent upon, i.e., the subplan below o;
2. The subplan P_2 of P generated by all the actions (except o) the resources occurring in $in(o)$ are *not* dependent upon i.e., the subplan generated by $(O \setminus O_1) \setminus \{o\}$.
3. The plan P_o exactly consisting of the action o.

By Assumption 1, these plans can be specified if P is an existing plan, since P_1, P_2 and P_o are subplans of P. By Proposition 6, there exist some $g_1, g_2 \subseteq d$ such that P can be written as $P = (P_1 \oplus_{g_1} P_o) \oplus_{g_2} P_2$. But then, by Proposition 8, $P_o = (P \ominus_\emptyset P_2) \ominus_\emptyset P_1$. □

The following corollary is an immediate generalization of the preceding proposition:

Corollary 11. *Let $P = (O, d, \theta)$ be a plan and $O' \subseteq O$ an arbitrary subset of actions. Then there exists a sequence of plan deletion operators that applied to P results in a plan P' just containing O' as a set of independently executable actions, that is $P' = (O', \emptyset, id)$.*

Proof. By the previous proposition, given a set $O' \subset O$ and a plan $P = (O, d, \theta)$, for each $o_i \in O'$ we can obtain a plan P_{O_i} generated by o_i. Then let $P' = P_{o_1} \oplus_\emptyset P_{o_2} \oplus_\emptyset \ldots \oplus_\emptyset P_{o_m} = (O', \emptyset, id)$. □

Using this proposition and corollary it is not difficult to prove the result that the set $\{\oplus, \ominus\}$ is a complete set of plan operators under the assumptions stated:

Proposition 12. *Suppose (I, P, G) is inadequate. If there exists an adequate plan (I, P', G) and there exists a plan library $\{P_j\}_{j=1}^n$ such that every action o occurring in P' also occurs in some P_j, then there exists at least one series of plan transformations which, when applied to P and the set $\{P_j\}$, results in the plan P'.*

Proof. [Sketch] Let $P = (O, d, \theta)$ and $P' = (O', d', \theta)$. Let $O_s = O \cap O'$. By the previous corollary, there exists a sequence of applications of the deletion operator and addition operator which, applied to P, results in a plan $P_{O_s} = (O_s, \emptyset, id)$. By the previous proposition and Assumption 2, there also exists a series of applications of deletion operators to plans (in the plan library) resulting in a set of plans P_o for every $o \in O' \setminus O$. Finally, using the addition operator, the plan P' can be easily assembled from the plan P_s and the set of plans $\{P_o\}_{o \in O' \setminus O}$. □

3.3 Replacement

Instead of using two separate plan operators, we would like to combine both operators into one single plan operator that is able to simulate them both. To this end we introduce the replacement operator $\otimes$ that serves to replace a subplan of a plan P by some other plan P'. Like the deletion operator, the replacement operator needs an additional parameter specifying the subplan to be replaced. In addition, the replacement operator also needs a parameter specifying how the new plan is glued to the existing plan. We therefore use an operator $\otimes$ with three parameters: the subplan to be deleted from P, a dependency function g to free up additional resources and a dependency function h specifying how the new plan P' should be glued to (free) resources in P.

Definition 13. *Let $P_1 = (O_1, d_1, \theta_1)$ and $P_2 = (O_2, d_2, \theta_2)$ be plans, P_1' a subplan of P_1 and let g, h be partial dependency functions. The resulting plan $P = P_1 \otimes_{P_1', g, h} P_2$ is defined as the plan $P = (O, d, \theta)$ where:*

1. $O = (O_1 \setminus O_1') \cup O_2$;
2. $d = ((((r, r') \in d_1 \mid r, r' \notin res(P_1')\}) \dagger g) + d_2) \dagger h$
3. *θ is the unifier of the dependency pairs in d.*

We now show that $\otimes$ is able to simulate both the addition and the deletion operator.

Proposition 14. *The replacement operator is able to simulate both the plan addition as well as the plan deletion operator.*

Proof. Let $P = P_1 \oplus_g P_2$. Then by definition of the replacement operator, $P = P_1 \otimes_{\not\emptyset,\emptyset,g} P_2$. Note that here g is used to glue the replacement for the empty plan to P_1.

Let $P = P_1 \ominus_g P_2$. Then $P = P_1 \otimes_{P_2,g,\emptyset} \not\emptyset$. That is, plan deletion is the same as replacing a subplan by an empty plan. □

3.4 A stronger completeness result for replacement

As an immediate corollary of Proposition 14 we obtain that the replacement operator is a complete operator. In fact, we can show a somewhat stronger result stating that the minimum number of applications of the replacement operator $\otimes$ needed to transform a plan P into another plan P' using a plan library $\mathcal{P} = \{P_j\}_{j=1}^n$ is less than twice the *minimum* number of plan transformations that are needed using an optimal set of plan transformation operators.[10]

To determine this lower bound, let us first define some additional notions. Let $P' = (O', d', \theta')$ be a plan and $\{P_j = (O_j, d_j, \theta_j)\}_{j=1}^n$ a set of plans constituting the plan library. A *labeling* for P' using $\{P_j\}$ is a function $\lambda : O' \to \bigcup_j O_j$ such that

1. $\lambda(o') = o_j$ implies that $o'\theta = o_j\theta$ for some substitution θ,
   i.e., the actions with their input and output resources are compatible;
2. $\lambda(o') = \lambda(o'')$ implies $o' = o''$,
   i.e. an action o_j occurring in some plan in the plan library cannot be used twice as a label for different actions in O' (hence λ is injective).

The size $size(\lambda)$ of a labeling λ is the cardinality of the set $\{P_j \mid \exists o' \in O'[\lambda(o') \in O_j]\}$, i.e., the number of different plans from the plan library mentioned by λ. Note that under the assumption that using a plan from the plan library implies the application of at least one operator, $size(\lambda)$ is a lower bound for the number of operations needed to obtain P' from the plans used in the plan library, since $size(\lambda)$ different plans are used from this library to compose P'.

The labeling λ for P' is said to be *minimal* with respect to a subset $T \subseteq \{P_j\}_{j=1}^n$ if a *minimum number* of plans in T is used to label the actions in P'. That is, as few plans from T as possible are involved in composing P' using $\{P_j\}_{j=1}^n$. Now we can define a lower bound on the number of transformations needed to transform a plan P to a plan P' as follows.

Proposition 15. *Let P be a source plan, $\{P_j\}_{j=1}^n$ be a plan library and P' a target plan. Then the minimum number of transformation steps needed to transform P to P' using plans in $\{P\} \cup \{P_j\}_{j=1}^n$ equals at least $size(\lambda) - 1$ where λ is a minimal labeling of P' with respect to $\{P_j\}_{j=1}^n$ using plans in $\{P\} \cup \{P_j\}_{j=1}^n$.*

Proof. [Sketch] Since λ is minimal with respect to $\{P_j\}_{j=1}^n$, if an action in O' is labeled with an action in some P_j from the library, the plan P_j used at least once in a transformation step. Therefore, there are at least $size(\lambda)-1$ (the number of plans required besides P) different transformation steps needed. □

Now our stronger completeness result can be stated as follows;

[10] Here, we make a trivial assume stating that, whenever a plan P_j from the plan library is *involved* in composing P' at least one application of a plan transformation operator is needed.

Proposition 16. *Let P be a source plan, $\{P_j\}_{j=1}^n$ be a plan library and P' a target plan. Then there exists a transformation from P to P' with at most $2m-1$ applications of the replacement operator $\otimes$, where m is a lower bound on the number of applications any set of plan transformation operators would need to transform P into P'.*

Proof. [Sketch] Let λ be a minimal labeling with respect to $\{P_j\}_{j=1}^n$, using $P \cup \{P_j\}_{j=1}^n$. Consider the actions o' in P' such that $\lambda(o') \in O$. Apply a replacement operator on P replacing the set of actions o not occurring in $ran(\lambda)$ by $\not\Diamond$ and returning a plan consisting of a set of independent actions. Then, for each plan P_j used in λ, apply the replacement operator once to remove all actions not used in the labeling and once to insert the remaining actions into the current plan. This requires $1+2(size(\lambda)-1) = 2size(\lambda)-1$ transformation steps. Since λ is minimal w.r.t. the set of plans in the plan library, $size(\lambda)$ is a lower bound for the number of applications needed (whatever set of transformation operators is used) and the proposition follows. □

Remark. Note that the plan operators we introduced require a current plan P to be transformed and another plan P' used to transform P. Hence, for an agent to be able to use these plan operators it must have access to a set of such plans P'. For this purpose, we assumed that the agent has a knowledge base containing plans to choose from. We called this knowledge base the *plan library*. We now discuss some aspects of such a plan library into more detail.

In its simplest form, a plan library can be conceived as a simple collection of plans. Often, however, it occurs that a number of plans share the same subplan or two plans are exactly equal except for some subplan in which they differ. In both cases, we could easily reduce the size of the plan library by using *plans with gaps* instead of complete plans. For example, suppose that $P_1, \ldots P_k$ share a common subplan P', we might replace every occurrence of P' in each P_i by an action $u \in U$ and add the subplan P' to the library, effectively reducing its size with $(k-1)(||P'|| - 1)$.

If the plan library may contain plans with gaps, we need additional constraints to hold for the library in order to guarantee that valid plans can be created out of plans with gaps. An intuitive constraint is that for each gap u that is present in a plan P from the library, another plan P' in the library must fit in the gap, i.e., a plan P' such that $in(u) \models_{P'} out(u)$. The resulting plan is the result of replacing u by P' in P and is written as $P \otimes_u P'$.

This requirement, however, does not solve our problem: for example, take a library with a plan A with a gap u_a and a plan B with a gap u_b where B fits u_a and A fits u_b. If there are no other plans that fit in either u_a or u_b, then it is not possible to create a valid ground plan out of these plans. The following requirement prevents such infinite regress to occur:

Definition 17. *A plan library L is* groundable *if for each plan P with a nonempty set U_P of gaps it holds that there there exists a finite subset of plans $P_1, P_2, \ldots P_k$ from L such that*

1. *for every $i = 1, 2, \ldots, k-1$ the plans $P'_1 = P$, $P'_{i+1} = P'_i \otimes_{u_i} P_i$ are well-defined, where each u_i is a gap occurring in $U_{P'_i}$;*
2. *$|U_{P'_k}| < |U_{P'_1}| = U_P$.*

4 A refinement (re)planning framework

To create and/or improve plans using the addition and deletion operators (or the replacement operator) and the plan library presented in the previous section, we have two options: Either we design new algorithms to implement these operators, or we show how existing algorithms and heuristics can be used to implement them. Choosing the last option, in this section we reformulate and generalize the classical refinement planning template algorithm [20] in order to use it in the ARF-replanning framework including the use of plan libraries. Hereafter, we present two examples of existing planning techniques that can be seen as instances of this new refinement template algorithm.

Remark. An important distinction between the ARF refinement framework and the classical planning framework is that in the ARF framework replanning is included. Consequently, if a replanning step is performed (during a refinement step), the set of possible candidates may be enlarged, while classically it is required that the size of the set of possible candidates monotonically decreases.

4.1 A template algorithm

As Kambhampati [20] argued, planning approaches do not only have in common the data structures they use, but they also share a common algorithmic structure. This common part, reformulated in ARF framework, is presented in the REFINE algorithm (Algorithm 1). The part that is different for each existing planning algorithm and depends on the specific planning method used is represented by the REFINESTEP (Algorithm 2).

The REFINE algorithm specifies how a solution *result* can be obtained from a partial plan P. Each time the function REFINE is executed and P is not a solution, a set $\mathcal{P}$ of refined versions of this plan is produced in a refinement step using the algorithm REFINESTEP. Then, iteratively an element P_i of $\mathcal{P}$ is selected, and REFINE is called recursively on P_i. This process stops once a solution has been found (or none exists) and the result (a successful plan or *fail*) is returned.

Algorithm 1 (REFINE $((I, P, \sigma, G), L)$)

Input: *An embedded plan* (I, P, σ, G) *and a plan library* L

Output: *A plan to attain* G *with* I *or 'fail'.*

begin

1. **if** $I \models In(P)\sigma$ *and* $Out(P)\sigma \models G\sigma$ *and* P *does not have any gaps* **then**
   - 1.1. **return** P;
2. **else**
   - 2.1. *result* := *fail*;
   - 2.2. $\mathcal{P}$:= REFINESTEP $((I, P, \sigma, G), L)$;
   - 2.3. **while** $\mathcal{P} \neq \emptyset$ *and result* = *fail* **do**
     - 2.3.1. *select a plan* P_i *of* $\mathcal{P}$;
     - 2.3.2. $\mathcal{P} := \mathcal{P} - \{P_i\}$;
     - 2.3.3. *determine a suitable* σ_i *for* P_i;
     - 2.3.4. *result* := REFINE$((I, P_i, \sigma_i, G), L)$;
   - 2.4. **return** *result*;

end

The RefineStep that is called in line 2.2 is described in Algorithm 2. This part of the planning algorithm is different for each existing planning method, we present a common template for these methods. Some instances (i.e., planning methods) may skip some of these steps, and others pay much more attention to one or more other steps. Different choices for each of the steps lead to different algorithms.

Abstracting from these differences, we present a common template for these methods. In the template algorithm we treat gaps in a plan identically to missing resources. The subgoals R are selected from all resources that need to be obtained: goals, missing input resources and gaps. For these selected subgoals one or more possible plans are selected from the plan library to attain these subgoals. The selected plans are combined with the original plan P. The result of one plan step is a set of one or more combinations of a selected plan with P.

Algorithm 2 (RefineStep $((I, P, \sigma, G), L)$)

Input: *An embedded plan (I, P, σ, G) and a plan library L*
Output: *A set of possible refinements $\mathcal{P}$*

begin

1. *select subgoals $R \subseteq (G\sigma - Out(P)\sigma) \cup (In(P)\sigma - I) \cup \bigcup_{u \in P} out(u)$;*
2. *select one or more subplans $P_i \in L$ to attain R;*
3. **for each** P_i **do**
   determine a subplan P_i' of P and functions g_i and h_i to combine P and P_i;
4. $\mathcal{P} := \left\{ P \otimes_{P_i', g_i, h_i} P_i \middle| P_i \right\}$;
5. **return** $\mathcal{P}$;

end

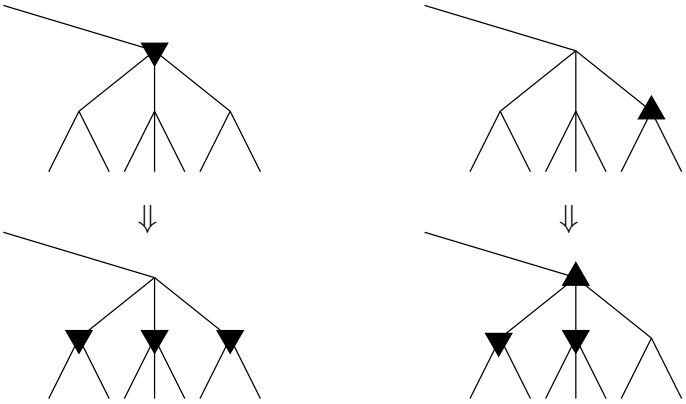
4.2 Examples of instances of the refine step template algorithm

To illustrate that existing planning methods can be reformulated in the ARF-framework, we describe two planning algorithms as instances of the RefineStep template algorithm.

4.2.1 Fast Forward

The planning algorithm Fast Forward (FF) [18] starts with the initial state and an empty plan. Repeatedly, the plan is extended with some actions, always adding them to the end of the plan. For each of the possible extensions of the plan (first with one action, then with two actions, etc.), a heuristic is used to calculate the value for the current state. The first possible extension leading to a state with a lower heuristic value is chosen. This heuristic mentioned uses a *relaxation* of the planning problem: it is assumed that actions *reproduce* all the resources they have just consumed (i.e. $out'(o) = out(o) \cup in(o)$). Therefore, a so-called relaxed plan can be constructed where all actions with satisfied inputs are added (possibly even in parallel) until the goal state is reached. The heuristic value is the cost of all actions that are needed to reach the goal state.

Although FF does not use the presented refinement framework, and is not about producing resources, it is in fact a form of refinement planning with the following refinement step. Given a plan that represents a set of possible



a. Extension b. Retraction

Figure 3: Extension and retraction as search in plan space. Extension replaces a plan tagged $\downarrow$ (represented by $\blacktriangledown$) by its children, tagged $\downarrow$. Retraction replaces a plan tagged $\uparrow$ ($\blacktriangle$ in the figure) by its parent (tagged $\uparrow$) and its siblings (tagged $\downarrow$).

solutions, a new plan is constructed by extending this partial plan at the end. The extension is selected using the heuristic described above. In Algorithm 3 this algorithm is presented as an instance of the template given in Algorithm 2. As a proof-of-concept, this algorithm has also been implemented [25].

Algorithm 3 (RefineStep-FF $((I, P, \sigma, G), L)$)

Input: *An embedded plan* (I, P, σ, G) *and a plan library* L

Output: *A set of possible refinements* $\mathcal{P}$

begin

1. *take all subgoals* $R = (G\sigma - Out(P)\sigma) \cup (In(P)\sigma - I)$;
2. *use the following heuristic to compute a plan Pi:*
   - 2.1. $h :=$ *the heuristic costs of producing* R *from the current end state;*
   - 2.2. $h' := \infty$;
   - 2.3. **while** $h' \geq h$ **do**
     - 2.3.1. *breadth-first select a sequence of actions* P_1 *from* L *to extend* P;
     - 2.3.2. *determine a sub plan* P_1' *of* P *and functions* g_1 *and* h_1 *to combine* P *and* P_1;
     - 2.3.3. $h' :=$ *the heuristic value of the end state of* $P \oplus_{P_1', g_1, h_1} P_1$;
     - 2.3.4. $\mathcal{P} := \left\{ P \otimes_{P_1', g_1, h_1} P_1 \right\}$;
3. **return** $\mathcal{P}$;

end

4.2.2 Plan adaptation

As a second example, we show that the SPA [17] algorithm for plan adaptation can also be reformulated in the ARF. This system is based on the least commitment approach [28].

After retrieving an initial plan from their plan library, the SPA system starts an adaptation routine that can either *extend* the plan (adding further constraints or actions), or *retract* decisions that have been made. This adaptation algorithm

performs a breadth-first search. The list of nodes to be expanded is kept in a list of the form $\langle P, \uparrow \rangle$ or $\langle P, \downarrow \rangle$, meaning that a plan P may be further refined (in case of a $\downarrow$) or that a decision may be retracted (signified by a $\uparrow$). Figure 3 shows what happens when a node is further refined or retracted from.

Note that the refinement strategy of SPA does not necessarily reduce the set of candidate plans in every step. Actually, the set of candidate plans may grow during the refinement. The refinement strategy for SPA is shown in Algorithm 4. There is a twofold difference with the standard template: (i) depending on whether the plan is tagged $\uparrow$ or $\downarrow$ SPA can remove parts of the plan or add them; (ii) in the case of plan retraction, step 4 includes an extra call to RefineStep-SPA. Since SPA requires the generation of sibling plans, we first calculate a set $\mathcal{P}'$ containing the parents (step 4.1), and calculate the children of these parents in step 4.3 (using a recursive call). Finally, step 4.4 combines the parents and their children, excluding the plan P as this is already processed.

Algorithm 4 (RefineStep-SPA $((I, P, \sigma, G), L)$)

Input: *An embedded plan (I, P, σ, G) and a plan library L*

Output: *A set of possible refinements $\mathcal{P}$*

begin

1. *select subgoals $R \subseteq (G\sigma - Out(P)\sigma) \cup (In(P)\sigma - I)$;*
2. *select one or more subplans $P_i \in L$ to attain R if P is tagged $\uparrow$, or the empty plan $\not\Diamond$ if P is tagged $\downarrow$;*
3. **for each** P_i **do**
   determine a subplan P_i' of P and functions g_i and h_i to add $P_i \neq \not\Diamond$ to P or determine subplans P_i' of P that can be removed from P to free R;
4. **if** *P is tagged $\uparrow$* **then**
   4.1. $\mathcal{P}' := \left\{ \langle P \otimes_{P_i', g_i, h_i} P_i, \uparrow \rangle \middle| P_i \right\}$;
   4.2. *determine a suitable σ_i for each $P_i \in \mathcal{P}'$;*
   4.3. $\mathcal{P}'' := \bigcup_{P_i' \in \mathcal{P}'}$ RefineStep-SPA$((I, \langle P_i', \downarrow \rangle, \sigma_i, G), L)$;
   4.4. $\mathcal{P} := \mathcal{P}' \cup \mathcal{P}'' \setminus \{P, \downarrow\}$;
5. **else**
   5.1. $\mathcal{P} := \left\{ \langle P \otimes_{P_i', g_i, h_i} P_i, \downarrow \rangle \middle| P_i \right\}$;
6. **return** $\mathcal{P}$;

end

Note that these instantiations of the RefineStep algorithm illustrate both the use of previously generated plans (from the plan library) as well as the transformation method to adapt them.

5 Conclusions and future work

Current planner work off-line and have no memory of what problems they have solved in the past. While there are systems that tackle some of these problems, we are not aware of a consistent integrating approach. In this paper we have discussed a conceptually rather simple framework, the Action Resource Formalism (ARF), where on-line planning is combined with a history of past experience (in the spirit of case-based planning).

In the ARF actions are seen as processes that consume and produce resources. A plan in this framework specifies the dependencies between these consumed and produced resources. This explicit notion of dependencies between actions is required for almost any planning algorithm. Another advantage of this formalism is the fact that the result of an action is seen as a set of products (i.e., resources). This makes it easier to reason about exchanging results in a multi agent context.

Using the ARF, we described how plans can be combined using an addition operator $\oplus$, and how we can remove parts from a plan using a deletion operator $\ominus$. The combination of these operators $\otimes$ was shown to be complete, provided that the right plans are available to the agent. We also showed that this operator work with constant overhead, i.e., it is able to describe plan transformations rather efficiently.

An agent not only needs this kind of plan operators, it also needs to store plans that are successful or domain knowledge given by a human expert. Slightly extending the ARF, we have discussed a way of dealing with past plan experience using a plan library. To efficiently represent existing plans in such a library we introduced the notion of a *gap*. Such gaps are undefined actions for which other plans can be substituted to create a complete plan. We showed which constraints should be placed on a plan library to ensure that all the plans in it can be completed.

The last part of the framework is an extension to Kambhampati's [20] refinement planning. This extension makes it possible to describe replanning algorithms as well. The strength of the framework was illustrated by showing how a planning technique as well as a technique that could be used for replanning fit into the framework. Such a uniform way to describe different types of planning algorithms is very useful for finding new variants and combining existing approaches. Furthermore, such a framework can help to compare different methods in a fair way.

In a multi-agent system, the plan library can also be used to represent services offered by other agents. Such a service can be modeled as a plan part that only describes the input-output relation of the service, and a large gap. If this gap is denoted with the name of the agent that offers this particular service, the planning agent can contact it when this plan part is selected from the library. In this way, services offered in a multi-agent system can be seamlessly incorporated in the planning process.

Task reduction (HTN) planning can also be integrated into the classical refinement planning approach [19, 27]. A refinement step can then also be a task reduction using one of the methods given in the plan library. In fact, this is quite similar to filling a gap in the plan. Using disjunctions of resources as described above, we hope to be able to show that HTN planning can also be seen as a special case of our refinement template.

Future work focuses on three main issues: (i) extending the current framework to allow a more efficient plan library representation , (ii) developing a continual algorithm that uses the framework to fully integrate planning, replanning and execution, and (iii) developing a method to create the plan library and maintain it. Further topics of interest are to show the relation with a disjunctive operator such as used in Graphplan [2] and the use of these operators in a multi-agent context.

It would also be interesting to see whether the planning template can be

extended to allow the combination of several refinement strategies. This would, e.g., allow different planners to cooperate on solving the same problem.

References

[1] Beek, P. V. and Chen, X. (1999). CPlan: A constraint programming approach to planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*. AAAI Press, Menlo Park, CA.

[2] Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300.

[3] Bonet, B. and Geffner, H. (2000). Planning as heuristic search. *Artificial Intelligence*. Special issue on Heuristic Search.

[4] Bonet, B. and Geffner, H. (2001). Heuristic search planner 2.0. *AI Magazine*, 22(3):77–80.

[5] de Weerdt, M. M., Bos, A., Tonino, J., and Witteveen, C. (2003). A resource logic for multi-agent plan merging. *Annals of Mathematics and Artificial Intelligence, special issue on Computational Logic on Multi-Agent Systems*, 37(1–2):93–130.

[6] Decker, K. S. and Lesser, V. R. (1992). Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems*, 1(2):319–346.

[7] DesJardins, M. E., Durfee, E. H., Ortiz, C. L., and Wolverton, M. J. (2000). A survey of research in distributed, continual planning. *AI Magazine*, 4:13–22.

[8] Drabble, B., Dalton, J., and Tate, A. (1997). Repairing plans on the fly. In *Proc. of the NASA Workshop on Planning and Scheduling for Space, Oxnard, CA*.

[9] Durfee, E. H. and Lesser, V. R. (1987). Planning coordinated actions in dynamic domains. Technical Report COINS-TR-87-130, Department of Computer and Information Science, University of Massachusetts, Amherst, MA. Also published as [10].

[10] Durfee, E. H. and Lesser, V. R. (1987). Planning coordinated actions in dynamic domains. In *Proceedings of the DARPA Knowledge-Based Planning Workshop*, pages 18.1–18.10.

[11] Ferrer, G. J. (2002). *Anytime Replanning Using Local Subplan Replacement*. Ph.D. thesis, University of Verginia.

[12] Fikes, R. E. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2):189–208.

[13] Fox, S. and Leake, D. (1995). Modeling case-based planning for repairing reasoning failures. In *Proceedings of the 1995 AAAI Spring Symposium on Representing Mental States and Mechanisms*, pages 31–38. AAAI Press, Menlo Park, CA.

[14] Gerevini, A. and Serina, I. (2000). Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS-00)*, pages 112–121. AAAI Press, Menlo Park, CA.

[15] Gratch, J. (1998). Reasoning about multiple plans in dynamic multi-agent environments. In *AAAI Fall Symposium on Distributed Continual Planning*.

[16] Hammond, K. J. (1990). Case-based planning: A framework for planning from experience. *Cognitive Science*, 14(3):385–443.

[17] Hanks, S. and Weld, D. (1995). A domain-independent algorithm for plan adaptation. *Journal of AI Research*, 2:319–360.

[18] Hoffmann, J. and Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of AI Research*, 14:253–302.

[19] Kambhampati, S. (1995). A comparative analysis of partial order planning and task reduction planning. *SIGART Bulletin*, 6(1):16–25.

[20] Kambhampati, S. (1997). Refinement planning as a unifying framework for plan synthesis. *AI Magazine*, 18(2):67–97.

[21] Koehler, J. (1994). An application of terminological logics to case-based reasoning. In Doyle, J., Sandewall, E., and Torasso, P., editors, *KR'94: Principles of Knowledge Representation and Reasoning*, pages 351–362. Morgan Kaufmann Publishers, San Mateo, CA.

[22] McDermott, D. (1998). PDDL – the planning domain definition language. Technical Report TR-98-003, Yale Center for Computational Vision and Control.

[23] Miksch, S. and Seyfang, A. (2000). Continual planning with time-oriented, skeletal plans. In *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-00)*, pages 511–515.

[24] Tonino, J., Bos, A., de Weerdt, M. M., and Witteveen, C. (2002). Plan coordination by revision in collective agent-based systems. *Artificial Intelligence*, 142(2):121–145.

[25] van der Krogt, R. P., de Weerdt, M. M., Planken, L. R., and Biesheuvel, A. (2003). Cabs planner. http://www.pds.twi.tudelft.nl/∼mathijs/.

[26] Veloso, M., Carbonell, J., Pérez, 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):81–120.

[27] Wang, X. and Chien, S. (1997). Replanning using hierarchical task network and operator-based planning. Technical report, Jet Propulsion Laboratory Nasa.

[28] Weld, D. S. (1994). An introduction to least-commitment planning. *AI Magazine*, 15(4):27–61.

[29] Wellman, M. P. (1992). A general-equilibrium approach to distributed transportation planning. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 282–289. AAAI Press, Menlo Park, CA.