A Resource Based Framework for Planning and Replanning

Roman van der Krogt, Mathijs de Weerdt, and Cees Witteveen
Parallel and Distributed Systems Group,
Delft University of Technology,
PO Box 5031, 2600 GA Delft, The Netherlands
{r.p.j.vanderkrogt, m.m.deweerdt, c.witteveen}@cs.tudelft.nl

Abstract

We discuss a rigorous unifying framework for both planning and replanning, extending an existing logic-based approach to resource-based planning. The primitive concepts in this Action Resource Framework (ARF) are actions and resources. Actions consume and produce resources. Plans are structures composed of actions, resource facts and an explicit dependency function specifying their interrelationships. In this framework, both planning and replanning are conceived as plan transformation processes accomplished by applying sequences of operations on plans. For this, we introduce operators for plan transformation and define the concept of a plan library. Using a refinement planning template, we show how some existing (re)planning methods and heuristics can be described as special cases of this framework. The advantage of the framework is that it offers a unifying view on planning and replanning.

1. Introduction

Usually, a planning problem is specified using a description of (i) the *current* (or *initial*) *state* an agent is in, (ii) the set of *actions* the agent is capable to perform, and (iii) the *goals* the agent is aiming at. 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 states specified by the goals. Currently, many systems and methods exist that try to tackle the planning problem (see [8] for an overview of state-of-the-art planning systems). These systems, however, have some serious drawbacks.

First of all, almost all of these planning systems rely on the tacit assumption that planning problems always can be solved *off-line*: the goal and the initial state are assumed to remain unchanged. Due to the dynamical nature of most planning problems, however, this assumption simply does not hold for a large number of domains. Even if the rate of change is rather low, it may not be possible to find a new plan for a complex domain in time when the plan is already being executed.

Secondly, most planning systems take for granted 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. The standard approach to planning 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) [5] system meets the second objection by addressing plan adaptation. This *case-based planner* maintains a database of past problems and their solution plans, and chooses an appropriate starting plan whenever it faces a planning problem. This plan then is modified to match the current goal and initial state requirements. Other systems focus on the replanning aspect to tackle the first problem. A system as GPG [3] finds the problems that exist in a plan (e.g. preconditions of actions that are not satisfied) and tries to replace parts of the plan such that these problems are solved. The term *continual planning* [2] is used to refer to systems in which planning, replanning and execution are all continuously interleaved.

This paper tries to overcome the mentioned problems by introducing an integrating framework that brings together ideas from both planning and replanning approaches. It is based upon an existing logic-based framework for resource based planning [9]. We show that refinement strategies can be built on top of this framework to supply computational support for (re)planning. We consider such a unifying framework for planning and replanning to have (at least) the following benefits. First of all, it should offer a common platform to develop new heuristics and algorithms. Secondly, it should offer possibilities to compare the quality of competing (planning) algorithms.

This paper is organized as follows. First, we give a concise introduction to an existing resource-based planning approach [9]. Next, we extend this formalism and use it to deal with replanning and we introduce plan transformation operators that are able to modify resource-based plans. 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. We present a method to use existing planning techniques in this framework, and show that the FF-approach [6] and SPA [5] algorithm for plan adaptation can be conceived as special cases of this (re)planning as plan transformation approach.

2. The action resource formalism

We introduce a framework for planning and replanning. This *Action Resource Framework*, abbreviated ARF, is based upon previous work [1, 9]. 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 a plan with gaps.

2.1. The ARF framework: basic notions

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

A *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, if *cycle* is a resource type that is used to describe bicycles, having attributes like its identifier id and a location *loc*, then $cycle(1 : id, \mathrm{A} : loc)$ is a resource fact describing a cycle located in A with identifier 1. To uniquely identify resource facts, a special attribute *identity* is used to distinguish it from other resources having the same type and possibly the same values of their attributes. Subsequently, we denote a resource of type t with identifier i as $t_i(\ldots)$.[3]

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 cycles in location A: $cycle_2(i : id, \mathrm{A} : loc)$.

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 identity 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 $out(o)$ that it produces. Furthermore, an action may contain a specification of some variables occurring in the set of output resources as *parameters* $param(o)$ of the action. To ensure that output resources are uniquely defined, these resources may only contain variables that already occur in the input resources or in the set of the parameters. An example of an action is:

$$\begin{aligned} pedal(d : loc) : \; & cycle(i : id, s : loc), road(s : loc, d : loc) \Rightarrow \\ & cycle(i : id, d : loc), road(s : loc, d : loc) \end{aligned}$$

This specifies how a person can travel from a source location s to a destination d. This action requires a cycle at the source, and a road between the two locations and "produces" a cycle at the destination and the road again (to make it available to other actions).[4] An action o can be applied to a set of (ground) resources R if a ground substitution θ exists 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 $R \setminus in(o)\theta \cup out(o)\theta$ is produced.

In general, a single action applied on 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

[1] In the original formalism [1], 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.

[3] The identifier of the resource fact has no connection with any identifier that the object being referred to may have. Thus, the resource fact $cycle_1(2 : id, \ldots)$ has identifier 1, but refers to a cycle with identifier 2.

[4] Note that this particular model of the world is rather unfortunate, as it allows only one agent to use the road at a time. However, a more realistic model would make the example unneccesary complicated.

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*. Let O be a set of actions.[5] We define plans over O as structured objects composed of actions in O. 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 in O is $res(O) = cons(O) \cup prod(O)$. 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 $d : cons(O) \to prod(O) \cup \{\perp\}$ specifying (in a unique way) for each resource r to be consumed which resource r' produced by another action is used to provide r (or $\perp$ if r is not produced by an action in O).*[6]

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 involved are compatible and (ii) d generates a partial order between the actions.

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 $<_d = \ll^+$ 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* free plan *P over a set of actions O is a triple $P = (O, d, \theta)$ where d is a dependency relation specifying dependencies between compatible relations and generates 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 $(\emptyset, \emptyset, \emptyset)$ is denoted by $\diamondsuit$.*

Given a plan $P = (O, d, \theta)$, the set of input resources of a plan, denoted by $In(P)$, is the set of resources $\{r\theta \mid d(r) = \perp\}$. The set of output resources denoted by $Out(P)$ is the set $\{r\theta \mid d^{-1}(r) = \perp\}$. Furthermore, we define $prod(P) = prod(O)\theta$ and $cons(P) = cons(O)\theta$.

[5]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 $o \neq o'$.

[6]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 $\mathrm{ran}(d)$, $d^{-1}(r') = \perp$.

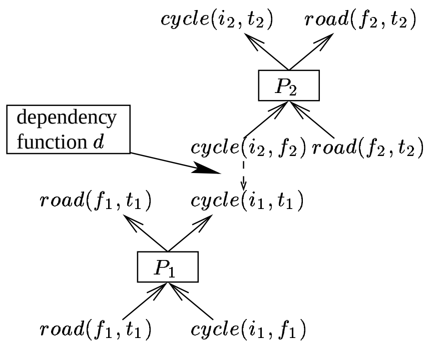

Figure 1. An example plan.

Free 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 a substitution σ exists 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 (I, P, σ, G) is adequate for a substitution σ.

An example of a plan is shown in Figure 1. It shows how two *pedal* actions can be combined. The substitution θ that unifies the two actions could be: $\theta = \{i_2 := i_1, f_2 := t_1\}$. This plan can be used with an initial set of resources $I = \{road(\mathrm{A} : loc, \mathrm{B} : loc), road(\mathrm{B} : loc, \mathrm{C} : loc), cycle(1 : id, \mathrm{A} : loc)\}$ to satisfy a goal $G = \{cycle(1 : id, \mathrm{C} : loc)\}$.

2.2. 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 if P is executed. Sometimes, however, we have to deal with less perfect plans. In these cases plans contain *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 $out(u)$. 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 a 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 undefined with respect to $\mathcal{O}$.*

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 gaps U over $\mathcal{O}$, and $P' = (O', d', \theta')$ 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:*

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

This concludes the introduction to ARF and its extension to plans with dependency functions and plans with gaps. In the next section we use the (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 done.

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 I is a finite set of ground resources specifying the initial situation, $\mathcal{O}$ is the set of possible actions an agent is capable to execute, and G is a finite set of general resources specifying the goals. A *solution* to Π is a plan $P = (O, d, \theta)$ such that (i) $O \subset \mathcal{O}$, i.e., the agent can execute it, and (ii) (I, P, G) is adequate, i.e., $I \models_P G$.

A *replanning* problem occurs when 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', or the actual set of goals is G' and the triple (I', P', G') is no longer adequate.

Note that instances of both 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).[7] 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''.

[7]Traditional planners choose the triple $(\emptyset, \diamondsuit, \emptyset)$ here.

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

Obviously, in most cases, such a transformation consists of several smaller steps. Therefore, both planning and replanning can be described as constructing *sequences* of plan transformation steps. Such 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*. In the next subsections we first discuss a number of plan transformation operators and 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 another plan P'. For a proof of the completeness of these operators, we refer the reader to [11].

3.1.1. Addition. Analogously to the action concatenation operator used in traditional planning, 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 in- and output resources in both plans are created. This gluing function g is a partial function overriding the specifications of the existing dependency functions d_1 and d_2 in both plans. (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 $h(x) = f(x)$ otherwise.)

Definition 5 *Let $P_1 = (O_1, d_1, \theta_1)$ and $P_2 = (O_2, d_2, \theta_2)$ be plans and $g : Out(P_1) \cup Out(P_2) \to In(P_1) \cup In(P_2)$ 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*[8]*, and*

3. *$\theta = \theta_1\theta_g + \theta_2\theta_g$ where θ_g is the mgu of the set of pairs $\{(r\theta_1\theta_2, g(r)\theta_1\theta_2) \mid r \in \mathrm{dom}(g)\}$.*[9]

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 $P_1 \oplus_g P_2 = (O_1 \cup O_2, d_1 + d_2, \theta_1 + \theta_2)$;

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

[9]Note that we always assume that whenever two plans P_1 and P_2 are combined, $\mathrm{var}(P_1) \cap \mathrm{var}(P_2) = \emptyset$.

2. If $\mathrm{dom}(g) \neq \emptyset$ and $P_2 = \not\Diamond$, the resulting plan P is a *sequential refinement* of P_1, that is $<_d$ extends the partial order $<_{d_1}$.

3.1.2. Deletion. Addition extends a plan, e.g., in order to obtain new (goal) resources available in another 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. We therefore define an operator $\ominus$ that 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, to specify which dependencies occurring in d have to be removed. 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 6 *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 it only contains pairs of the form $(r, \perp)$, where $r \in \mathrm{dom}(d_1)$. 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. $P_2 = \not\Diamond$. 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 g is the empty function and $P_2 \neq \not\Diamond$, the resulting plan is the plan P_1 after removing the actions and dependencies from P_2.

As expected, a partial duality exists between the operators $\oplus$ and $\ominus$: suppose that a plan P consists of two subplans P_1 and P_2 such that P can be written as $P = P_1 \oplus_g P_2$ for a gluing function g. Then $P \ominus_\emptyset P_2 = P_1$.

3.2. The plan library

Note that the plan operators require a plan P' to transform a plan 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 assume that the agent has a knowledge base containing plans to choose from. We call this knowledge base the *plan library*. We now discuss some aspects of such a plan library in 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.

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, it is not possible to create a ground plan out of these plans. The following requirement prevents such infinite regress to occur:

Definition 7 *A plan library L is* groundable *iff for each plan P with a nonempty set U_P of gaps it holds that there exists a finite subset of plans $P_1, P_2, \ldots P_{k-1}$ 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 and the plan library presented in the previous section, we have two options: either we can design new algorithms to implement these operators, or we can show how existing algorithms and heuristics can be used to implement them. Choosing the last option, we reformulate and generalize the refinement planning template algorithm [7] 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.

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

Kambhampati [7] argues that planning approaches have common data structures as well as a common algorithmic structure. The *refinement approach* he proposes can be used to distinguish between the common structure of planning and the particularities of the planning method at hand. The refinement planning algorithm, reformulated in the ARF framework, is presented in Algorithm 1 (REFINE). The part that is different for each existing planning algorithm and depends on the specific planning method used is represented by REFINESTEP (Algorithm 2).

The REFINE algorithm specifies how a solution *result* can be obtained from a partial plan P. In each step, REFINE generates refined versions of the plan P_i that is being worked on (the set $\mathcal{P}$) and then selects one of the refinements to process in the next step, backtracking if a solution cannot be found.

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 $\sigma_i$ for $P_i$;[10]
          2.3.4. $result :=$ REFINE$((I, P_i, \sigma_i, G), L)$;
      2.4. return $result$;
end
```

The REFINESTEP called in line 2.2 is described in Algorithm 2. This step differs for different planning strategies, therefore Algorithm 2 describes a template. A planning method is defined by the way it implements these steps.

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 $R \subseteq (G\sigma \setminus Out(P)\sigma) \cup (In(P)\sigma \setminus I) \cup \bigcup_{u \in P} out(u)$;
  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. $\mathcal{P} := \left\{ P \otimes_{P'_i, g_i, h_i} P_i \middle| P_i \right\}$;[11]
  5. return $\mathcal{P}$;
end
```

[10]For the sake of clarity, we omitted further details here.

[11]Here, we use $P \otimes_{P',g,h} P''$ as a short hand for $(P \ominus_g P') \oplus_h P''$.

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) [6] starts with the initial state and an empty plan. Repeatedly, the plan is extended with some actions, always adding 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 value for the current state is calculated. The first possible extension leading to a state with a lower heuristic value is chosen.

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 change to REFINESTEP: instead of selecting one or more plans from the library, FF uses its heuristic to select one plan P_i in line 2. The rest of the REFINESTEP function is not changed. As a proof-of-concept, this algorithm has also been implemented in the ARF [10].

4.2.2. Plan adaptation. As a second example, we show that the SPA [5] algorithm for plan adaptation can also be reformulated in the ARF. This system is based on the least commitment approach [12].

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 nodes that have to be expanded is kept in a list of pairs $\langle P, \uparrow \rangle$ or $\langle P, \downarrow \rangle$. Here, a $\downarrow$ means that a plan may be further refined, whereas $\uparrow$ means that a decision may be retracted.

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 even grow during the refinement. The (simplified) refinement strategy for SPA is shown in Algorithm 3. Note that line 4 differs slightly from the original algorithm for simplicity.[12]

Algorithm 3 **(**REFINESTEP-SPA $((I, P, \sigma, G), L)$**)**

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

begin
  1. select subgoals $R \subseteq (G\sigma \setminus Out(P)\sigma) \cup (In(P)\sigma \setminus I)$;
  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
       $\mathcal{P} := \left\{ \langle P \otimes_{P_i', g_i, h_i} P_i, \uparrow\rangle, \langle P \otimes_{P_i', g_i, h_i} P_i, \downarrow\rangle \middle| P_i \right\}$;
  5. else
       $\mathcal{P} := \left\{ \langle P \otimes_{P_i', g_i, h_i} P_i, \downarrow\rangle \middle| P_i \right\}$;
  6. return $\mathcal{P}$;
end
```

5. Conclusions and future work

This paper started by noting that a number of objections exist to the way planning is usually conceptualized in most planning systems. These planners 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, based on the Action Resource Formalism (ARF), where on-line planning is combined with a history of past experience (cf. case-based planning [4]). 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.

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 $\ominus$. Furthermore, we showed how to store successful plans or domain knowledge given by a human expert. Slightly extending the ARF, we discussed a way of dealing with past plan experience using a plan library.

The last part presented is an extension to Kambhampati's [7] 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 and a replanning technique 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.

[12] We do not ensure that the same node is not generated twice.

Future work will focus on four main issues: (i) extending the current framework to allow a more efficient plan library representation resembling the HTN approach, (ii) developing a continual algorithm, that uses the framework to fully integrate planning, replanning and execution, (iii) developing a method to fill the plan library and maintain it, (iv) extending the refinement template to deal with multi-agent planning. 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.

Acknowledgments Roman van der Krogt is supported by the Freight Transport Automation and Multi-modality program of the TRAIL research school for Transport, Infrastructure and Logistics. Mathijs de Weerdt is supported by the Seamless Multi-modal Mobility program of the same research school.

References

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

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

[3] A. Gerevini and I. Serina. 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, 2000.

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

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

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

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

[8] D. Long and M. Fox. International planning competition. http://www.dur.ac.uk/d.p.long/competition.html, 2002.

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

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

[11] R. P. J. van der Krogt, M. M. de Weerdt, and C. Witteveen. Integrating planning and replanning, manuscript. http://www.pds.twi.tudelft.nl/∼roman/, 2003.

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