HTN Plan Repair via Model Transformation

Daniel Höller[1,4], Pascal Bercher[2,4], Gregor Behnke[3,4], and Susanne Biundo[4]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2] The Australian National University, Canberra, Australia
[3] University of Freiburg, Freiburg, Germany
[4] Ulm University, Institute of Artificial Intelligence, Ulm, Germany
hoeller@cs.uni-saarland.de, pascal.bercher@anu.edu.au, behnkeg@cs.uni-freiburg.de, susanne.biundo@uni-ulm.de

Abstract. To make planning feasible, planning models abstract from many details of the modeled system. When executing plans in the actual system, the model might be inaccurate in a critical point, and plan execution may fail. There are two options to handle this case: the previous solution can be modified to address the failure (plan repair), or the planning process can be re-started from the new situation (re-planning). In HTN planning, discarding the plan and generating a new one from the novel situation is not easily possible, because the HTN solution criteria make it necessary to take already executed actions into account. Therefore all approaches to repair plans in the literature are based on specialized algorithms. In this paper, we discuss the problem in detail and introduce a novel approach that makes it possible to use unchanged, off-the-shelf HTN planning systems to repair broken HTN plans. That way, no specialized solvers are needed.

Keywords: HTN Planning · Plan Repair · Re-Planning.

1 Introduction

When generating plans that are executed in a real-world system, the planning system needs to be able to deal with execution failures, i.e. with situations during plan execution that are not consistent with the predicted state. Since planning comes with several assumptions that may not hold the the real system, such situations may arise for several reasons like non-determinism in the problem, exogenous events, or actions of other agents. When we speak of an execution failure, we mean that the outcome of an action is not like anticipated by the planning model (e.g. due to the given reasons).

Two mechanisms have been developed to deal with such situations: Systems that use *re-planning* discard the original plan and generate a new one from the novel situation. Systems using *plan repair* adapt the original plan so that it can deal with the unforeseen change. In classical planning, the sequence of already executed actions implies no changes other than state transition. The motivation for plan repair in this setting has e.g. been *efficiency* [20] or *plan stability* [18], i.e. finding a new plan that is as similar as possible to the original one.

In hierarchical task network (HTN) planning [17, 8], the hierarchy has wide influence on the set of solutions and it makes the formalism also more expressive than classical planning [21, 22]. The hierarchy can e.g. enforce that certain actions might only be executed in combination. By simply re-starting the planning process from the new state, those implications are discarded, thus simple re-planning is no option and plans have to be repaired, i.e., the implications have to be taken into account. Several approaches have been proposed in the literature, all of them use special repair algorithms to find the repaired plans.

In this paper we make the following contributions (some of the work has been presented before in a workshop version of the paper [25]):

– We discuss the issues that arise when using a re-planning approach that re-starts the planning process from the new state in HTN planning.
– We survey the literature on plan repair in HTN planning.
– Based on a transformation for plan and goal recognition [23], we introduce a transformation-based approach that makes it possible to use unchanged HTN planning systems to repair broken HTN plans.

Outline. We first introduce HTN planning and specify the plan repair problem (Sec. 2), discuss issues with repairing HTN plans (Sec. 3), summarize related work (Sec. 4), and give our transformation (Sec. 5) and its properties (Sec. 6).

2 Formal Framework

This section introduces HTN planning and specifies the repair problem.

2.1 HTN Planning

In HTN planning, there are two types of tasks: *primitive* tasks equal classical planning actions, which cause state transitions. *Abstract* tasks describe more abstract behavior. They can not be applied to states directly, but are iteratively split into sub-tasks until all tasks are primitive.

We use the formalism by Geier and Bercher [19, 22]. A classical planning problem is defined as a tuple $P_c = (L, A, s_0, g, \delta)$, where L is a set of propositional state features, A a set of action names, and $s_0, g \in 2^L$ are the initial state and the goal definition. A state $s \in 2^L$ is a *goal state* if $s \supseteq g$. The tuple $\delta = (\mathit{prec}, \mathit{add}, \mathit{del})$ defines the preconditions prec as well as the add and delete effects (add, del) of actions, all are functions $f : A \to 2^L$. An action a is applicable in a state s if and only if $\tau : A \times 2^L$ with $\tau(a, s) \Leftrightarrow \mathit{prec}(a) \subseteq s$ holds. When an (applicable) action a is applied to a state s, the resulting state is defined as $\gamma : A \times 2^L \to 2^L$ with $\gamma(a, s) = (s \setminus \mathit{del}(a)) \cup \mathit{add}(a)$. A sequence of actions $(a_0 a_1 \ldots a_l)$ is applicable in a state s_0 if and only if for each a_i it holds that $\tau(a_i, s_i)$, where s_i is for $i > 0$ defined as $s_i = \gamma(a_{i-1}, s_{i-1})$. We call the state s_{l+1} the resulting state from the application. A sequence of actions $(a_0 a_1 \ldots a_l)$ is a solution if and only if it is applicable in s_0 and results in a goal state.

An HTN planning problem $P = (L, C, A, M, s_0, tn_I, g, \delta)$ extends a classical planning problem by a set of abstract (also called compound) task names C, a set of decomposition methods M, and the tasks that need to be accomplished which are given in the so-called initial task network tn_I. The other elements are equivalent to the classical case. The tasks that need to be done as well as their ordering relation are organized in *task networks*. A task network $tn = (T, \prec, \alpha)$ consists of a set of identifiers T. An identifier is just a unique element that is mapped to an actual task by a function $\alpha : T \to A \cup C$. This way, a single task can be in a network more than once. $\prec : T \times T$ is a set of ordering constraints between the task identifiers. Two task networks are called to be *isomorphic* if they differ solely in their task identifiers. An abstract task can by decomposed by using a (decomposition) method. A method is a pair (c, tn) of an abstract task $c \in C$ that specifies to which task the method is applicable and a task network tn, the method's subnetwork. When decomposing a task network $tn_1 = (T_1, \prec_1, \alpha_1)$ that includes a task $t \in T_1$ with $\alpha_1(t) = c$ using a method (c, tn), we need an isomorphic copy of the method's subnetwork $tn' = (T', \prec', \alpha')$ with $T_1 \cap T' = \emptyset$. The resulting task network tn_2 is then defined as

$$\begin{aligned} tn_2 =& ((T_1 \setminus \{t\}) \cup T', \prec' \cup \prec_D, (\alpha_1 \setminus \{t \mapsto c\}) \cup \alpha') \\ \prec_D =& \{(t_1, t_2) \mid (t_1, t) \in \prec_1, t_2 \in T'\} \cup \\ & \{(t_1, t_2) \mid (t, t_2) \in \prec_1, t_1 \in T'\} \cup \\ & \{(t_1, t_2) \mid (t_1, t_2) \in \prec_1, t_1 \neq t \wedge t_2 \neq t\} \end{aligned}$$

We will write $tn \to^* tn'$ to denote that a task network tn can be decomposed into a task network tn' by applying an arbitrary number of methods in sequence.

A task network $tn = (T, \prec, \alpha)$ is a solution to a planning problem P if and only if 1. all tasks are primitive, $\forall t \in T : \alpha(t) \in A$, 2. it was obtained via decomposition, $tn_I \to^* tn$, 3. there is a sequence of the task identifiers in T in line with the ordering whose application results in a goal state.

2.2 Plan Repair Problem in HTN Planning

Next we specify the plan repair problem, i.e., the problem occurring when plan execution fails (that could be solved by plan repair or re-planning), please be aware the ambiguity of the term *repair* naming the problem and a way to resolve it. A plan repair problem consists of three core elements: The original HTN planning problem P, its original solution plus its already executed prefix, and the execution error, i.e., the state deviation that occurred during executing the prefix of the original solution.

Most HTN approaches that can cope with execution failures do not just rely on the original solution, but also require the modifications that transformed the initial task network into the failed solution. How these modifications look like depends on the planning system, e.g., whether it is a progression-based system [28, 24, 26] or a plan-space planner [11, 16]. To have a general definition, we include the so-called decomposition tree (DT) of the solution. A DT is a tree representation of the decompositions leading to the solution [19]. Its nodes represent tasks; each abstract task is labeled with the method used for decomposition, the children in the tree correspond to the subtasks of that specific method. Ordering constraints are also represented, such that a DT *dt* yields the solution *tn* it represents by restricting the elements to *dt*'s leaf nodes.

Definition 1 (Plan Repair Problem). *A plan repair problem can now be defined as a tuple* $P_r = (P,\ tn_s,\ dt,\ exe,\ F^+,\ F^-)$ *with the following elements.* P *is the original planning problem.* $tn_s = (T, \prec, \alpha)$ *is the failed solution for it,* dt *the DT as a witness that* tn_s *is actually a refinement of the original initial task network, and* $exe = (t_0, t_1, \ldots t_n)$ *is the sequence of already executed task identifiers,* $t_i \in T$*. Finally, the execution failure is represented by the two sets* $F^+ \subseteq L$ *and* $F^- \subseteq L$ *indicating the state features that were (not) holding contrary to the expected state after execution the solution prefix* exe*.*

Notice that not every divergence of the state predicted by the model and the actual state during execution prevents further execution of the plan. A technique detecting whether repair is necessary is e.g. described by Bercher et al. [10].

Though they have been introduced before, we want to make the terms replanning and plan repair more precise.

Definition 2 (Re-Planning). *The old plan is discarded, a new plan is generated starting from the current state of the system that caused the execution failure.*

Definition 3 (Plan Repair). *The system modifies the non-executed part of the original solution such that it can cope with the unforeseen state change.*

3 About Re-Planning in HTN Planning

In classical planning, a prefix of a plan that has already been executed does not imply any changes to the environment apart from the actions' effects. It is therefore fine to discard the current plan and generate a new one from scratch from the (updated) state of the system. HTN planning provides the domain designer a second means of modeling: the hierarchy. Like preconditions and effects, it can be used to model either physics *or* advice. Figure 1 illustrates the Toll Domain. A car moves in a road network. The red square indicates the city center (the toll area). Whenever the car takes a road segment starting inside the center, a toll has to be paid at a position marked with a credit card. Since the car may want to use a segment more than once (e.g. because the driver wants to visit certain shops in a specific ordering), it is not sufficient to mark *which* segments have been used, they need to be counted. For simplicity, we assume that the toll area is left at the end (i.e. the final position is outside). An HTN domain is given in Figure 2. It contains five methods. Actions are given in boxes, abstract tasks are non-boxed. The *driveTA* action is only applicable inside the toll area, *drive* only outside it and the *payToll* action only at positions marked with a credit card. Whenever *driveTA* is added to the plan, an instance of the *payToll*

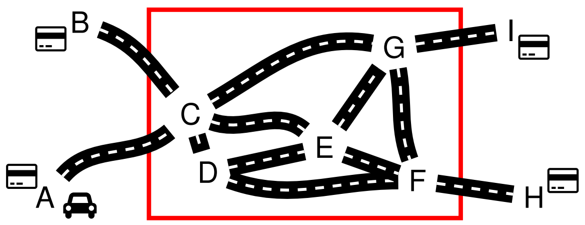

Fig. 1. The Toll Domain.

task is added and the toll for that single segment is paid when the toll area is left. The domain is a simple example for a context-free language-like structure. When described in STRIPS, one has to commit to a maximum number of visits or encode it using a richer classical model (e.g. supporting numeric variables).

Consider a car starting at position A and driving to H. A planning system could come up with the following plan:

$$\begin{gathered} \mathit{drive}(A, C), \mathit{driveTA}(C, G), \mathit{driveTA}(G, F), \\ \mathit{driveTA}(F, H), \mathit{payToll}(), \mathit{payToll}(), \mathit{payToll}() \end{gathered}$$

Consider an execution failure after the first $\mathit{driveTA}$ action: being at location G: the driver gets aware that the road to F is closed. Re-planning is triggered. The planning system comes up with the following new plan:

$$\begin{gathered} \mathit{driveTA}(G, E), \mathit{driveTA}(E, F), \mathit{driveTA}(F, H), \\ \mathit{payToll}(), \mathit{payToll}(), \mathit{payToll}() \end{gathered}$$

The driver executes the plan and reaches H, but while four segments are used, the toll gets only paid three times.

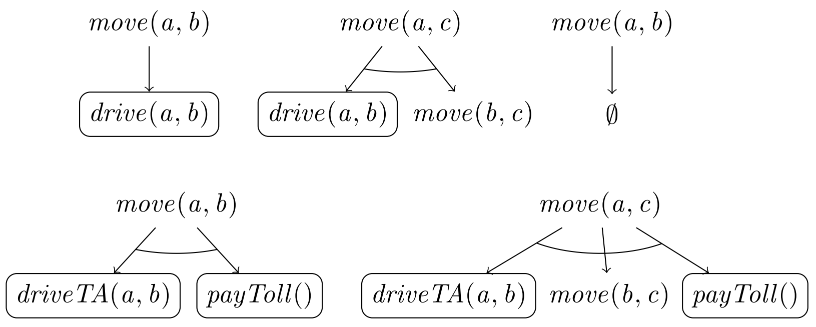

Fig. 2. Sketch of an HTN model for the Toll domain.

As we have seen, the hierarchy assures that certain properties hold in every plan and the domain designer might rely on these properties. There are different ways to ensure them during a repair process:

1. The responsibility can be shifted to the domain designer, i.e., the domain must be created in a way that the planning process can be started from any state of the real-world system. This leads to a higher effort for the domain expert and it might also be more error-prone, because the designer has to consider possible re-planning in every intermediate state of the system.
2. The reasoning system that triggers planning and provides the planning problem is responsible to incorporate additional tasks to make the system safe again. This shifts the problem to the creator of the execution system. This is even worse, since this might not even be a domain expert, and the execution system has to be domain-specific, i.e., the domain knowledge is split.
3. The repair system generates a solution that has the properties assured by the hierarchy. This solution leads to a single model containing the knowledge, the planning domain; and the domain designer does not need to consider every intermediate state of the real system.

Since it represents a fully domain-independent approach, we consider the last solution to be the best. This leads us to a core requirement of a system that solves the plan repair problem: regardless of whether it technically uses plan repair or re-planning, it needs to generate solutions that start with the same prefix of actions that have already been executed. Otherwise, the system potentially discards "physics" that have been modeled via the hierarchy. Therefore we define a solution to the plan repair problem as follows.

Definition 4 (Repaired Plan). *Given a plan repair problem $P_r = (P, tn_s, dt, exe, F^+, F^-)$ with $P = (L, C, A, M, s_0, tn_I, g, \delta)$, $tn_s = (T, \prec, \alpha)$ and $exe = (t_0, t_1, \ldots t_n)$, a repaired plan is a plan that*

1. *can be executed in s_0*
2. *is a refinement of tn_I,*
3. *has a linearization with a prefix $(\alpha(t_0), \alpha(t_1), \ldots \alpha(t_n))$ followed by tasks executable despite the unforeseen state change, resulting in a goal state.*

4 HTN Plan Repair: Related Work

Before we survey practical approaches on plan repair in HTN planning, we recap the theoretical properties of the task. Modifying existing HTN solutions (in a way so that the resulting solution lies still in the decomposition hierarchy) is undecidable even for quite simple modifications [5] and even deciding the question whether a given sequence of actions can be generated in a given HTN problem is NP-complete [6, 7]. Unsurprisingly, the task given here – finding a solution that starts with a given sequence of actions – is undecidable [6].

We now summarize work concerned with plan repair or re-planning in hierarchical planning in chronological order.

One of the first approaches dealing with execution failures in hierarchical planning is given by Kambhampati and Hendler [27]. It can be seen as *plan repair*, since it repairs the already-found solution with the least number of changes. There are two properties we want to point out regarding our classification: (1) Though they assume a hierarchical model, (i.e., they also feature abstract tasks and decomposition methods for refining them), the planning goals are not defined in terms of an initial task network, but as a state-based goal. Abstract tasks use preconditions and effects so that they can be inserted as well. The plan that is repaired is a primitive plan, but it was generated by a hierarchical planner. (2) They do not base their work on an execution error, such as an unexpected change of a current situation, but instead assume that the problem description changes, i.e., the initial state and a goal description.

Drabble et al. [15] introduced algorithms to repair plans in case of action execution failure and unexpected world events by modifying the existing plan.

Boella and Damiano [14] propose a repair algorithm for a reactive agent architecture. Though they refer to it as re-planning, it can be seen as plan repair according to our classification. The original problem is given in terms of an initial plan that needs to be refined. Repair starts with a given primitive plan. They take back performed refinements until finding a more abstract plan that can be refined into a new primitive one with an optimal expected utility.

Warfield et al. [31] propose the RepairSHOP system, which extends the progression-based HTN planner SHOP [29] to cope with unexpected changes to the current state. Their *plan repair* approach shows some similarities with the previous one, as they backtrack decompositions up to a point where different options are available that allow a refinement in which the unexpected change does not violate executability. To do this, the authors propose the *goal graph*, which is a representation of the commitments that the planner has already made to find the executed solution.

Bidot et al. [12] propose a *plan repair* algorithm to cope with execution failures. The same basic idea has later been described in a more dense way relying on a simplified formalism [13]. Their approach also shows similarities to the previous two, as they also start with the failed plan and take planning decisions back, starting with those that introduced failure-associated plan elements, thereby re-using much of the planning effort already done. The already executed plan elements (steps and orderings) are marked with so-called *obligations*, a new flaw class in the underlying flaw-based planning system.

The previous plan repair approach has been further simplified by Bercher et al. [9, 10]. Their approach uses obligations to state which plan elements must be part of any solution due to the already executed prefix. In contrast to the approaches given before, it starts with the initial plan and searches for refinements that achieve the obligations. Technically, it can be regarded *re-planning*, because it starts planning from scratch and from the *original* initial state while ensuring that new solutions start with the executed prefix. It was implemented in the plan-space-based planning system PANDA [11] and practically in use in the described assembly scenario, but never systematically evaluated empirically.

The most recent approach for HTN *plan repair* is the one by Barták and Vlk [4]. It focuses on *scheduling*, i.e., the task of allocating resources to actions and determine their execution time. In case of an execution error (a changed problem specification), another feasible schedule is found via backjumping (conflict-directed backtracking).

All these approaches address execution failures by a specialized algorithm. In the next section, we propose a novel approach that solves the problem *without* relying on specialized algorithms. Instead, it encodes the executed plan steps and the execution error into a standard HTN problem, which allows to use standard HTN solvers instead.

5 Plan Repair via Domain Transformation

Technically, the task is similar to *plan recognition as planning* (PGR) and we heavily build on the transformation-based PGR approach by Höller et al. [23]. The encoding of actions used by Höller et al. is similar to the one introduced by Ramírez and Geffner [30] in plan recognition in the context of classical planning.

Let $P_r = (P, tn_s, dt, exe, F^+, F^-)$ be the plan repair problem, $P = (L, C, A, M, s_0, tn_I, g, \delta)$ with $\delta = (prec, add, del)$ the original HTN planning problem, $exe = (a_1, a_2, \ldots, a_m)$ the sequence of already executed actions[5], and $F^+, F^- \in 2^L$ the sets of the unforeseen positive and negative facts, respectively. We define the following HTN planning problem $P' = (L', C', A', M', s'_0, tn'_I, g', \delta')$ with $\delta' = (prec', add', del')$ that solves the plan repair problem.

First, a set of new propositional symbols is introduced that indicate the position of some action in the enforced plan prefix. We denote these facts as l_i with $0 \leq i \leq m$ and $l_i \notin L$ and define $L' = L \cup \{l_i \mid 0 \leq i \leq m\}$.

For each task a_i with $1 \leq i < m - 1$ in the prefix of executed actions, a new task name a'_i is introduced with

$$\begin{aligned} prec'(a'_i) &\mapsto prec(a_i) \cup \{l_{i-1}\}, \\ add'(a'_i) &\mapsto add(a_i) \cup \{l_i\} \text{ and} \\ del'(a'_i) &\mapsto del(a_i) \cup \{l_{i-1}\}. \end{aligned}$$

The last action in the executed prefix a_m needs to have additional effects, it performs the unforeseen state change.

$$\begin{aligned} prec'(a'_m) &\mapsto prec(a_m) \cup \{l_{m-1}\}, \\ add'(a'_m) &\mapsto \big(add(a_m) \setminus F^-\big) \cup F^+ \cup \{l_m\} \text{ and} \\ del'(a'_m) &\mapsto del(a_m) \cup F^- \cup \{l_{m-1}\}. \end{aligned}$$

The original actions shall be ordered after the prefix, i.e., $\forall a \in A : prec'(a) \mapsto prec(a) \cup \{l_m\}$. The new set of actions is defined as $A' = A \cup \{a'_i \mid 1 \leq i \leq m\}$.

[5] To simplify the following definitions, the definition is slightly different from Def. 1, where it is a sequence of identifiers mapped to the tasks. The latter makes it possible to identify which decomposition resulted in an action, which is not needed here.

To make the first action of the prefix applicable in the initial state, the symbol l_0 is added, i.e., $s'_0 = s_0 \cup \{l_0\}$. We enforce that every solution starts with the entire prefix, i.e. $g' = g \cup \{l_m\}$.

Having adapted the non-hierarchical part of the problem, the newly introduced actions now need to be made reachable via the hierarchy. Since they simulate their duplicates from the prefix of the original plan, the planner should be allowed to place them at the same positions. This can be enabled by introducing a new abstract task for each action appearing in the prefix, replacing the original action at each position it appears, and adding methods such that this new task may be decomposed into the original or the new action. Formally, the transformation is defined in the following way.

$$\begin{aligned}
C' &= C \cup \{c'_a \mid a \in A\}, c'_a \notin C \cup A,\\
M^c &= \{(c, (T, \prec, \alpha')) \mid (c, (T, \prec, \alpha)) \in M\}, \text{ where}\\
&\qquad \forall t \in T \text{ with } \alpha(t) = k \text{ we define } \alpha'(t) = \begin{cases} k, & \text{if } k \in C\\ c'_k, & \text{else.}\end{cases}\\
M^a &= \{(c'_a, (\{t\}, \emptyset, \{t \mapsto a\})) \mid \forall a \in A\},
\end{aligned}$$

So far the new abstract tasks can only be decomposed into the original action. Now we allow the planner to place the new actions at the respective positions by introducing a new method for every action in $exe = (a_1, a_2, \ldots, a_m)$, decomposing a new abstract task c'_{a_i} into the executed action a_i:

$$M^{exe} = \{(c'_{a_i}, (\{t\}, \emptyset, \{t \mapsto a'_i\})) \mid a_i \in exe\}$$

The set of methods is defined as $M' = M^c \cup M^a \cup M^{exe}$.

Figure 3 illustrates the method encoding. On the left, a method m is given that decomposes an abstract task c into another abstract task c' and an action a. When we assume that a is contained in the prefix once, the given approach will result in three new methods in the new model that are given on the right. In the original method m, the action a is replaced by a new abstract task c_a (the resulting method is named m_1). When a is contained in other methods, it is replaced in the same way as given here. The abstract task c_a can be decomposed using one of the two methods m_2 and m_3. They replace c_a either by the original action a or by the newly introduced copy a'. That way, a' can be added into the solution at exactly the positions where a has been possible before.

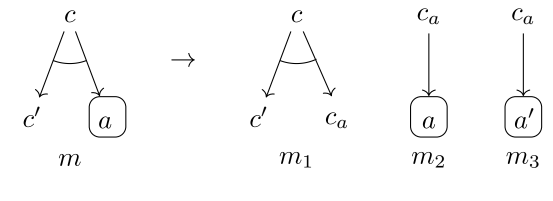

Fig. 3. Encoding of methods.

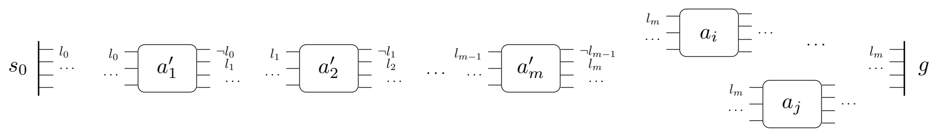

Fig. 4. Schema of the overall plan generated from our transformation.

Figure 4 shows the schema of plans generated from our transformation: The structure of preconditions and effects results in a totally ordered sequence of actions in the beginning that is equal to these actions already executed. The last action (a'_m) has additional effects that realize the unforeseen state change. Afterwards the planner is free to generate any partial ordering of tasks as allowed by the original domain. The newly introduced goal feature forces the prefix to be in the plan.

Now we come back to our example. Starting the planner after the execution failure on the transformed model, it might now come up with the following plan that is executed starting with the third action. Now the toll is paid correctly.

$$\begin{gathered} drive(A, C)', driveTA(C, G)', driveTA(G, E), \\ driveTA(E, F), driveTA(F, H), \\ payToll(), payToll(), payToll(), payToll() \end{gathered}$$

Like the approach given by Bercher et al. [9], our transformation is a mixture between re-planning and repair. The planning process is started from scratch, but the system generates a solution that starts with the executed prefix and incorporates constraints induced by the hierarchy. Since it enforces the properties by using a transformation, the system that generates the actual solution can be a standard HTN planning system. For future work, it might be interesting to adapt the applied planning heuristic to increase plan stability (though this would, again, lead to a specialized system).

A problem class related to HTN planning where our transformation may also be used is HTN planning with task insertion [19, 2, 8] (TIHTN planning). Here, the planning system is allowed to add actions apart from the decomposition process. This makes the issues described in Section 3 less obvious, since the sequence of already executed actions might simply be considered *inserted*. However, a TIHTN planner needs to include all actions that are enforced via the hierarchy into the plan. Consider an action that can only be executed once and that has already been executed. When the prefix is considered inserted and planning is done from scratch, the planner needs to insert it again and no executable plan is found. Using our techniques (with minor changes) will prevent such situations.

6 Theoretical Properties of the Transformation

We do not give an empirical evaluation of the encoding. Instead we analyze the theoretical properties of the transformation.

We first show that the common sub-classes of HTN planning are closed under the transformation, i.e., that the transformed problem is in the same class as the original problem. We consider the following sub-classes:

1. Full HTN planning – The full formalism without restrictions.
2. Totally ordered HTN planning [17, 1] – The subtasks of all methods as well as the initial task network are totally ordered.
3. Tail-recursive HTN planning [3, 1] – In this class, there exists a pre-order on all tasks in the domain (not to be confused with the partial order on tasks within task networks). For all methods, the following holds: All subtasks except the distinct last task (if there is one) have to be smaller than the decomposed task with respect to the pre-order. The last task may not be greater. Intuitively, there are layers of tasks and all but the last task are in lower layers. The last task may be in the same layer. It is always possible to put primitive tasks on a single lowest layer.
4. Acyclic HTN planning – There is no recursion, i.e. (one or more steps of) decomposition of a task can not end up with the same task.

Afterwards, we show that the size of the transformed problem is – in the worst case – quadratic when considering the executed prefix part of the input (as done in our definition, cf. Def. 1). Please be aware, however, that the transformed problem can become arbitrary large in comparison to the original planning problem when the prefix is not considered part of the input.

6.1 Closure Properties

We first show closure properties of the given HTN sub-classes.

Full HTN Models. Since the model resulting from the transformation is a common HTN model, it is obvious that this class is closed under the transformation.

Totally Ordered HTN Models. When we have a look at the transformation, we see that (1) the ordering of modified methods is exactly as it was before and that (2) new methods contain a single subtask (making them totally ordered, i.e. maximally restricted in the ordering). Surely we do not change properties related to method ordering and the transformation resulting from a totally ordered model as input will also be totally ordered.

Tail-recursive HTN Models. The newly introduced abstract tasks form a decomposition layer between the original abstract tasks and the original and newly introduced actions: when such a new abstract task is reached, it is not possible to reach any abstract task anymore, only actions are reachable from it. Given there was an ordering of the tasks with the given properties before the transformation, we can define one on the transformed model by inserting a new "layer" between the original actions and the original abstract tasks and put the new abstract tasks on this layer. So if there was a pre-order with the given properties on all tasks in the first place (making the problem tail-recursive) the new model will still possess such a pre-order, i.e. it is still tail-recursive.

Acyclic Models. We do not introduce recursive decomposition structures. When a model is non-recursive before, it will also be non-recursive afterwards.

6.2 Model Size

Next we show that the transformation is in the worst case quadratic in the input. Let n be the size of the HTN model and m the size of the already executed prefix. We consider both model and prefix as part of the input. For each step in the prefix, the transformation adds to the model:

1. a single state feature,
2. a single new action,
3. up to two methods (with constant size), and
4. at most a single abstract task.

Due to the (rather artificial) case that the actions included in the prefix dominate the size of the HTN model (e.g. if they have every state feature as precondition/effect and the elements defining the hierarchy, i.e. abstract tasks and methods are small in comparison to the actions) the size of the transformation is bounded by $n \times m$ (caused by adding a new action m times). In practice, it should be much smaller, though.

Even in this artificial kind of domain, the size of the encoding will only be quadratic for a certain prefix length. When the prefix is very small, the size of the input model will dominate the size of the resulting model, if it becomes very large, the prefix dominates the size. Only when they are of similar size we end up with quadratic model size.

7 Conclusion

In this paper we introduced a novel approach to repair broken plans in HTN planning. We discussed that simply re-starting the planning process is no option since this discards constraints implied by the hierarchy. Instead, systems need to come up with a new plan that starts with the actions that have already been executed. All systems in the literature tackle the given problem by specialized algorithms. We provided a transformation-based approach that enables the use of unchanged HTN planning systems.

Acknowledgments

Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 232722074 – SFB 1102 / Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 232722074 – SFB 1102.

References

1. Alford, R., Bercher, P., Aha, D.W.: Tight bounds for HTN planning. In: Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS). pp. 7–15. AAAI Press (2015)
2. Alford, R., Bercher, P., Aha, D.W.: Tight bounds for HTN planning with task insertion. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI). pp. 1502–1508. AAAI Press (2015)
3. Alford, R., Shivashankar, V., Kuter, U., Nau, D.S.: HTN problem spaces: Structure, algorithms, termination. In: Proceedings of the 5th Annual Symposium on Combinatorial Search (SoCS). pp. 2–9. AAAI Press (2012)
4. Barták, R., Vlk, M.: Hierarchical task model for resource failure recovery in production scheduling. In: Proceedings of the 15th Mexican International Conference on Artificial Intelligence (MICAI). pp. 362–378. Springer (2017)
5. Behnke, G., Höller, D., Bercher, P., Biundo, S.: Change the plan – How hard can that be? In: Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS). pp. 38–46. AAAI Press (2016)
6. Behnke, G., Höller, D., Biundo, S.: On the complexity of HTN plan verification and its implications for plan recognition. In: Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS). pp. 25–33. AAAI Press (2015)
7. Behnke, G., Höller, D., Biundo, S.: This is a solution! (. . . but is it though?) – Verifying solutions of hierarchical planning problems. In: Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS). pp. 20–28. AAAI Press (2017)
8. Bercher, P., Alford, R., Höller, D.: A survey on hierarchical planning – One abstract idea, many concrete realizations. In: Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI). pp. 6267–6275. IJCAI Organization (2019)
9. Bercher, P., Biundo, S., Geier, T., Hörnle, T., Nothdurft, F., Richter, F., Schattenberg, B.: Plan, repair, execute, explain – How planning helps to assemble your home theater. In: Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS). pp. 386–394. AAAI Press (2014)
10. Bercher, P., Höller, D., Behnke, G., Biundo, S.: User-centered planning. In: Biundo, S., Wendemuth, A. (eds.) Companion Technology – A Paradigm Shift in Human-Technology Interaction, pp. 79–100. Cognitive Technologies, Springer (2017)
11. Bercher, P., Keen, S., Biundo, S.: Hybrid planning heuristics based on task decomposition graphs. In: Proceedings of the 7th Annual Symposium on Combinatorial Search (SoCS). pp. 35–43. AAAI Press (2014)
12. Bidot, J., Schattenberg, B., Biundo, S.: Plan repair in hybrid planning. In: Proceedings of the 31st German Conference on Artificial Intelligence (KI). pp. 169–176. Springer (2008)
13. Biundo, S., Bercher, P., Geier, T., Müller, F., Schattenberg, B.: Advanced user assistance based on AI planning. Cognitive Systems Research **12**(3-4), 219–236 (2011)
14. Boella, G., Damiano, R.: A replanning algorithm for a reactive agent architecture. In: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA). pp. 183–192. Springer (2002)
15. Drabble, B., Dalton, J., Tate, A.: Repairing plans on-the-fly. In: Proceedings of the NASA workshop on Planning and Scheduling for Space. pp. 13.1–13.8 (1997)

16. Dvořák, F., Barták, R., Bit-Monnot, A., Ingrand, F., Ghallab, M.: Planning and acting with temporal and hierarchical decomposition models. In: Proceedings of the 26th International Conference on Tools with Artificial Intelligence (ICTAI). pp. 115–121. IEEE (2014)
17. Erol, K., Hendler, J.A., Nau, D.S.: Complexity results for HTN planning. Annals of Mathematics and Artificial Intelligence **18**(1), 69–93 (1996)
18. Fox, M., Gerevini, A., Long, D., Serina, I.: Plan stability: Replanning versus plan repair. In: Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS). pp. 212–221. AAAI Press (2006)
19. Geier, T., Bercher, P.: On the decidability of HTN planning with task insertion. In: Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI). pp. 1955–1961. IJCAI/AAAI (2011)
20. Gerevini, A., Serina, I.: Fast plan adaptation through planning graphs: Local and systematic search techniques. In: Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS). pp. 112–121. AAAI Press (2000)
21. Höller, D., Behnke, G., Bercher, P., Biundo, S.: Language classification of hierarchical planning problems. In: Proceedings of the 21st European Conference on Artificial Intelligence (ECAI). pp. 447–452. IOS Press (2014)
22. Höller, D., Behnke, G., Bercher, P., Biundo, S.: Assessing the expressivity of planning formalisms through the comparison to formal languages. In: Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS). pp. 158–165. AAAI Press (2016)
23. Höller, D., Behnke, G., Bercher, P., Biundo, S.: Plan and goal recognition as HTN planning. In: Proceedings of the 30th IEEE International Conference on Tools with Artificial Intelligence (ICTAI). pp. 466–473. IEEE Computer Society (2018)
24. Höller, D., Bercher, P., Behnke, G., Biundo, S.: A generic method to guide HTN progression search with classical heuristics. In: Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS). AAAI Press (2018)
25. Höller, D., Bercher, P., Behnke, G., Biundo, S.: HTN plan repair using unmodified planning systems. In: Proceedings of the 1st ICAPS Workshop on Hierarchical Planning (HPlan). pp. 26–30 (2018)
26. Höller, D., Bercher, P., Behnke, G., Biundo, S.: HTN planning as heuristic progression search. Journal of Artificial Intelligence Research **67**, 835–880 (2020)
27. Kambhampati, S., Hendler, J.A.: A validation-structure-based theory of plan modification and reuse. Artificial Intelligence **55**, 193–258 (1992)
28. Nau, D.S., Au, T., Ilghami, O., Kuter, U., Murdock, J.W., Wu, D., Yaman, F.: SHOP2: An HTN planning system. Journal of Artificial Intelligence Research **20**, 379–404 (2003)
29. Nau, D.S., Cao, Y., Lotem, A., Muñoz-Avila, H.: The SHOP planning system. AI Magazine **22**(3), 91–94 (2001)
30. Ramírez, M., Geffner, H.: Plan recognition as planning. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI). pp. 1778–1783. AAAI Press (2009)
31. Warfield, I., Hogg, C., Lee-Urban, S., Muñoz-Avila, H.: Adaptation of hierarchical task network plans. In: Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference (FLAIRS). pp. 429–434. AAAI Press (2007)