Proceedings of the 1st ICAPS Workshop on Hierarchical Planning

HTN Plan Repair Using Unmodified Planning Systems

Daniel Höller and **Pascal Bercher** and **Gregor Behnke** and **Susanne Biundo**
Institute of Artificial Intelligence, Ulm University, D-89069 Ulm, Germany
{daniel.hoeller, pascal.bercher, gregor.behnke, 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.

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. Such situations may arise for several reasons. Planning models used for deterministic planning have to abstract from many details of the modeled system and the model might be inaccurate in a critical point. Up to a certain amount of non-determinism in the modeled system, it might also be beneficial to use deterministic planners and deal with execution errors.

Two mechanisms have been developed to deal with such failures: 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 been *efficiency* (Gerevini and Serina 2000) or *plan stability* (Fox et al. 2006), i.e. finding a new plan that is as similar as possible to the original one.

In hierarchical task network (HTN) planning (Erol, Hendler, and Nau 1996), the hierarchy has wide influence on the set of valid solutions and it makes the formalism also more expressive than classical planning (Höller et al. 2014; 2016). 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 give an elaborate discussion on the issues that arise when using a re-planning approach that re-starts the planning process from the new state in HTN planning.
- We introduce a novel transformation-based 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.

Next, we introduce HTN planning, specify the formal problem, discuss issues arising when repairing HTN plans, summarize related work, and give our transformation.

2 Formal Framework

This section first introduces HTN planning and specifies the repair problem afterwards.

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 Höller et al. (2016). Here, a classical planning domain 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 $\delta = (prec, add, del)$ 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 $\tau : A \times 2^L \to \{true, false\}$ with $\tau(a, s) \Leftrightarrow prec(a) \subseteq s$ 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 $\gamma(a, s) = (s \setminus del(a)) \cup add(a)$. 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 will call the state s_{l+1} the resulting state from the application. A sequence of actions $(a_0 a_1 \dots a_l)$ is a solution if and only if it is applicable in the initial state 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 follows.

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

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 decomposing the initial task network, $tn_I \to^* tn$, (3) there is a sequence $(t_1 t_2 \dots t_n)$ of the task identifiers in T in line with the ordering constraints $\prec$, and the application of $(\alpha(t_1)\alpha(t_2)\dots\alpha(t_n))$ in s_0 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 this term. 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 errors 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 may depend on the underlying planning system, e.g., whether it is a progression-based system (Nau et al. 2003; Höller et al. 2018a) or a plan-space planner (Bercher, Keen, and Biundo 2014; Dvořák et al. 2014). To have a general definition, we include the so-called decomposition tree (DT) of a given solution tn. A DT is a tree-like representation of performed decompositions. It forms a witness for a decomposition leading to the solution (Geier and Bercher 2011). Its nodes represent tasks; each abstract task is labeled with the method used for decomposing it, the children in the tree correspond to the subtasks of that specific method. All ordering constraints are also represented, such that a DT dt yields the solution tn it represents by restricting the elements of dt to dt's leaf nodes.

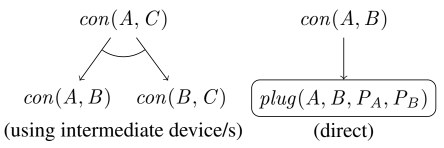

Figure 1: Core methods of an entertainment domain (example from Höller et al. 2018a).

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 $exe = (t_0, t_1, \dots t_n)$ 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.*

Though they have been introduced before, we want to make the terms re-planning 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 both physics *or* advice. Figure 1 shows (core parts of) a domain that models the task of assembling an entertainment system. The signal flow is thereby modeled via the hierarchy without using any state features. This can be done by the two given methods. When two devices A and C have to be connected (represented by the task $con(A, C)$), this can be done by using a third intermediate device B, or directly by performing a $plug$ action. That way, devices like a TV or DVD player

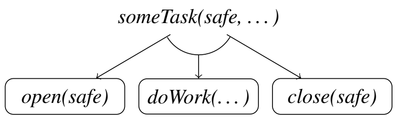

Figure 2: A sketch of a domain containing a pair of actions that have to be executed either both or none.

can be treated equal to cables or adapters and the hierarchy enforces the signal flow. Other things, like which plug fits into which port, or which port is free, can be represented in state. Clearly, this hierarchy represents physics, not advice.

Now imagine a situation where two devices shall be connected and re-planning is performed after half of the connections. Some cables have already been connected to ports and thus both are occupied. When re-planning does not include these circumstances, these cables are just treated as non-free and new cables are used. That way, resources are wasted and in worst case, no solution can be found.

Such situations might be considered during domain design. The domain might include an unplug action, or the recursive connection model can consider plugged cables between devices. However, it *has* to be addressed somehow.

Consider another domain where, for a certain action that causes a safety threat, a second action has to be performed to make the situation safe again, e.g. an action for opening a safe. Every safe that is opened must also be closed eventually. This can easily be modeled as an HTN domain. A sketch for such a domain is given in Figure 2. Though the given domain could also be modeled using some features in classical planning (e.g. by introducing a *closed* state feature and include it in the goal definition for every safe), please be aware that this is not always the case: Consider e.g. that one action needs to be done as many times as a second one. Then, there is no way to ensure it via state, since the state in planning is usually finite. It can, however, be modeled in the more expressive HTN formalism (Höller et al. 2016).

As we have seen in our examples, 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:

- 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 real-world system.
- 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.
- 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, and (3) has a linearization with a prefix equal to $(\alpha(t_0), \alpha(t_1), \ldots \alpha(t_n))$ followed by tasks executable despite the unforeseen state change.*

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 (Behnke et al. 2016) and even deciding the question whether a given sequence of actions can be generated in a given HTN problem is NP-complete (Behnke, Höller, and Biundo 2015; 2017). Unsurprisingly, the task given here – finding a solution that starts with a given sequence of actions – is indeed undecidable (Behnke, Höller, and Biundo 2015).

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 errors in hierarchical planning is given by Kambhampati and Hendler (1992). It can be characterized as *plan repair*, since they repair the already-found solution with the least number of changes. Though they assume a hierarchical model, the task hierarchy is just advice, i.e., the planning goals are not defined in terms of an initial task network, but as state-based goal. Abstract tasks use preconditions and effects so that they can be inserted as well. They do not base their work upon 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 goal description.

Drabble, Dalton, and Tate (1997) introduced algorithms to repair plans in case of action execution failure as well as unexpected world events by modifying the existing plan.

Boella and Damiano (2002) propose a technique that they refer to as re-planning, but the work can be seen as *plan repair* according to our classification. They propose a repair algorithm for a reactive agent architecture. 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. (2007) propose the RepairSHOP system, which extends the progression-based HTN planner SHOP (Nau et al. 2001) 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, Schattenberg, and Biundo (2008) propose a *plan repair* algorithm to cope with execution errors. The same basic idea has later been described in a more dense way relying on a simplified formalism (Biundo et al. 2011). 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 later been simplified further by Bercher et al. (2014; 2017). 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 already executed prefix. The approach was implemented in the plan-space-based planning system PANDA (Bercher, Keen, and Biundo 2014) and practically in use in the described assembly scenario, but never systematically evaluated empirically.

The most recent approach for HTN *plan repair* that we are aware of is by Barták and Vlk (2017). It focuses on *scheduling*, i.e., the task of allocating resources to actions and scheduling their execution time. In case of an execution error (a changed problem specification), they find another feasible schedule. They perform backjumping (i.e., conflict-directed backtracking) to find repaired solutions.

All these approaches address execution errors 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 our work on *Plan Recognition as Planning* (Höller et al. 2018b). The approach is based on two transformations, one of them enforces HTN plans to start with a prefix of observations.

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

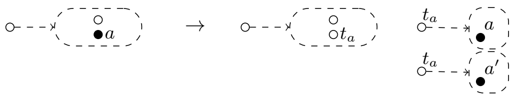

Figure 3: The original method (left) contains the action a that is part of the already executed prefix. This task is replaced by a new abstract task t_a (middle) and two new methods are added that decompose t_a either in a or in a' (right).

$(L', C', A', M', s'_0, tn'_I, g', \delta')$ with $\delta' = (prec', add', del')$ that solves the plan repair problem.

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

For each task a_i with $1 \le i < m - 1$ in the prefix of executed actions, a new task name a'_i is introduced with $prec'(a'_i) \mapsto prec(a_i) \cup \{l_{i-1}\}$, $add'(a'_i) \mapsto add(a_i) \cup \{l_i\}$ and $del'(a'_i) \mapsto del(a_i) \cup \{l_{i-1}\}$. The last action in the executed prefix a_m needs to have additional effects, it performs the unforeseen state change. $prec'(a'_m) \mapsto prec(a_m) \cup \{l_{m-1}\}$, $add'(a'_m) \mapsto (add(a_m) \setminus F^-) \cup F^+ \cup \{l_m\}$ and $del'(a'_m) \mapsto del(a_m) \cup F^- \cup \{l_{m-1}\}$. The original problem is placed after the prefix, i.e., $\forall a \in A$ holds that $prec'(a) \mapsto prec(a) \cup \{l_m\}$. And the new set of actions is defined as $A' = A \cup \{a'_i \mid 1 \le i \le m\}$. To make the first action of the prefix applicable in the initial state, the symbol l_0 is added, i.e., $s'_0 = s_0 \cup \{l_0\}$. To reuse the already executed actions, ensure that every solution starts with the entire prefix, i.e. $g' = g \cup \{l_m\}$.

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 done 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. A schema of the transformation is given in Figure 3. Formally, it is defined in the following way.

$$C' = C \cup \{c'_a \mid a \in A\}, c'_a \notin C \cup A,$$

$$M^c = \{(c, (T, \prec, \alpha')) \mid (c, (T, \prec, \alpha)) \in M\}, \text{ where}$$

$$\forall t \in T \text{ with } \alpha(t) = n \text{ and } \alpha'(t) = \begin{cases} n, & \text{if } n \in C \\ c'_n, & \text{else.} \end{cases}$$

$$M^a = \{(c'_a, (\{t\}, \emptyset, \{t \mapsto a\})) \mid \forall a \in A\},$$

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 $M' = M^c \cup M^a \cup M^{exe}$ and all elements of P' have been specified.

Like the approach given by Bercher et al. (2014), our transformation is technically a hybrid between re-planning (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).

6 Conclusion

In this paper we introduced a novel approach to repair broken plans in HTN planning. We elaborated that simply re-starting the planning process is no option since this would discard changes implied by the hierarchical part of the model. 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 modifying the applied planning system. We provided a compilation-based approach that enables the use of unchanged HTN planning systems. In future work, we want to empirically evaluate the feasibility of our approach.

Acknowledgments

This work was partly done within the technology transfer project "Do it yourself, but not alone: Companion-Technology for DIY support" of the SFB/TRR 62 funded by the German Research Foundation (DFG).

References

Barták, R., and Vlk, M. 2017. Hierarchical task model forresource failure recovery inproduction scheduling. In *Proc. of the 15th Mexican Int. Conf. on AI (MICAI 2016)*, 362–378. Springer.

Behnke, G.; Höller, D.; Bercher, P.; and Biundo, S. 2016. Change the plan – How hard can that be? In *Proc. of ICAPS 2016*, 38–46. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the complexity of HTN plan verification and its implications for plan recognition. In *Proc. of ICAPS 2015*, 25–33. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2017. This is a solution! (... but is it though?) – Verifying solutions of hierarchical planning problems. In *Proc. of ICAPS 2017*, 20–28. AAAI Press.

Bercher, P.; Biundo, S.; Geier, T.; Hörnle, T.; Nothdurft, F.; Richter, F.; and Schattenberg, B. 2014. Plan, repair, execute, explain – How planning helps to assemble your home theater. In *Proc. of ICAPS 2014*, 386–394. AAAI Press.

Bercher, P.; Höller, D.; Behnke, G.; and Biundo, S. 2017. *Companion Technology – A Paradigm Shift in Human-Technology Interaction*. Cognitive Technologies. Springer. chapter 5: User-Centered Planning, 79–100.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proc. of SoCS 2014*, 35–43. AAAI Press.

Bidot, J.; Schattenberg, B.; and Biundo, S. 2008. Plan repair in hybrid planning. In *Proc. of the 31st German Conf. on AI (KI 2008)*, 169–176. Springer.

Biundo, S.; Bercher, P.; Geier, T.; Müller, F.; and Schattenberg, B. 2011. Advanced user assistance based on AI planning. *Cognitive Systems Research* 12(3-4):219–236.

Boella, G., and Damiano, R. 2002. A replanning algorithm for a reactive agent architecture. In *Proc. of the 10th Int. Conf. on AI: Methodology, Systems, and Applications (AIMSA 2002)*, 183–192. Springer.

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*, 13–1–13–8.

Dvořák, F.; Barták, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. Planning and acting with temporal and hierarchical decomposition models. In *Proc. of the 26th IEEE Int. Conf. on Tools with AI (ICTAI 2014)*, 115–121. IEEE.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence* 18(1):69–93.

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

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proc. of IJCAI 2011*, 1955–1961. AAAI Press.

Gerevini, A., and Serina, I. 2000. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proc. of the 5th Int. Conf. on AI Planning Systems (AIPS 2000)*, 112–121. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proc. of ECAI 2014*, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proc. of ICAPS 2016*, 158–165. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018a. A generic method to guide HTN progression search with classical heuristics. In *Proc. of ICAPS 2018*. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018b. Plan and goal recognition as HTN planning. In *Proc. of the AAAI Workshop on Plan, Activity, and Intent Recognition (PAIR 2018)*.

Kambhampati, S., and Hendler, J. A. 1992. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence* 55:193–258.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 2001. The SHOP planning system. *AI Magazine* 22(3):91–94.

Nau, D. S.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: an HTN planning system. *JAIR* 20:379–404.

Warfield, I.; Hogg, C.; Lee-Urban, S.; and Muñoz-Avila, H. 2007. Adaptation of hierarchical task network plans. In *Proc. of the 20th Int. Florida AI Research Society Conf. (FLAIRS 2007)*, 429–434. AAAI Press.