Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning
Main Track

Correcting Hierarchical Plans by Action Deletion

Roman Barták[1] , **Simona Ondrčková**[1] , **Gregor Behnke**[2] , **Pascal Bercher**[3]
[1]Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
[2]University of Freiburg, Faculty of Engineering, Freiburg, Germany
[3]The Australian National University, College of Engineering & Computer Science, Canberra, Australia
{bartak, ondrckova}@ktiml.mff.cuni.cz, behnkeg@informatik.uni-freiburg.de,
pascal.bercher@anu.edu.au

Abstract

Hierarchical task network (HTN) planning is a model-based approach to planning. The HTN domain model consists of tasks and methods to decompose them into subtasks until obtaining primitive tasks (actions). There are recent methods for verifying if a given action sequence is a valid HTN plan. However, if the plan is invalid, all existing verification methods only say so without explaining why the plan is invalid. In the paper, we propose a method that corrects a given action sequence to form a valid HTN plan by deleting the minimal number of actions. This plan correction explains what is wrong with a given action sequence concerning the HTN domain model.

1 Introduction

Hierarchical planning is a popular form of automated planning (Bercher, Alford, and Höller 2019). It exploits the idea of identifying functional sequences of actions – sub-plans – and naming them as specific tasks that these action sequences solve. Tasks can further group into more complex (compound) tasks which define a hierarchy of tasks. In this hierarchy, each task may have several alternative ways to be achieved. From the other perspective, the hierarchical planning domain model specifies tasks and methods to decompose these tasks into simpler sub-tasks until obtaining primitive tasks – actions – that form a plan. This way, a human user describes control knowledge on achieving a specific goal (task) as a recipe to decompose the tasks into simpler sub-tasks. This approach speeds-up the planning process and gives better control over the plans – sequences of actions – generated by the planning engine. Thanks to these properties, hierarchical planning is prevalent in areas such as robotics (Kaelbling and Lozano-Pérez 2011) and computer games (Hoang, Lee-Urban, and Muñoz-Avila 2005; Neufeld et al. 2019).

Hierarchical plan verification checks that the action sequence is causally consistent, and it can be obtained by decomposition of any (or a specific) task. Recently, two methods have been proposed to verify if a given action sequence is a valid HTN plan. One method is based on translation to a Boolean satisfiability problem (Behnke, Höller, and Biundo 2017) and the other one uses parsing of grammars (Barták, Maillard, and Cardoso 2018; Barták et al. 2020; Barták et al. 2021). Using the parsing approach seems highly canonical given the close relationship between HTN models and formal grammars (Erol, Hendler, and Nau 1996; Höller et al. 2014; Höller et al. 2016; Barták and Maillard 2017). Grammar parsing is also a popular method in the hierarchical plan and task recognition problems (Vilain 1990; Avrahami-Zilberbrand and Kaminka 2005; Geib and Goldman 2009; Kabanza et al. 2013; Mirsky, Gal, and Shieber 2017). For valid plans, the verification techniques return the decomposition tree to justify the correctness of the plan. However, when the plan is not valid, all existing verification methods only say so but do not provide any additional information about the source of invalidity. One recent approach for dealing with unsuccessful plan verification proposes model changes that will make the given input plan a valid solution (Lin and Bercher 2021a; 2021b) as a basis for counter-factual explanations (Ginsberg 1986). These works are however pure complexity investigations.

In this paper, we address the issue with invalid HTN plans by studying how to correct a given action sequence to form a valid HTN plan. Specifically, we show how the correction can be done by deleting the minimal number of actions from the input action sequence. This is motivated by the following two scenarios. Firstly, as similarly suggested by Lin and Bercher (2021a) (they change the model's task hierarchy, whereas we change the plan) the necessary changes to the flawed input plan can be exploited as a basis for counterfactual explanations pointing out to the user why the plan is not a solution. Identifying the minimal number of actions that need to be deleted ensure a short and thus hopefully more comprehensible explanation. The second motivation comes from the area of goal recognition. Assume that we observe several agents' actions, we have a model of one agent's behavior, and we want to find out what task that agent is achieving. In addition to its goal-leading behavior, the agent also performs diversion actions (not coming from any task) putting the observer off track. Thus, we have to discern the actual goal-leading part of the plan from those diversion actions. Eliminating irrelevant actions to obtain a valid sub-plan according to the HTN model solves the above task-detection problem. We propose a technique based on parsing to select the longest valid hierarchical plan from a given sequence of actions. We furthermore prove that both the pursued form of plan verification and the proposed action deletion-minimizing plan correction is **NP-complete**.

2 Background on Classical and HTN Planning

Hierarchical planning has a long tradition in AI (Tate 1977; Wilkins 1984). The most widely used approach is known as Hierarchical Task Network (HTN) Planning (Erol, Hendler, and Nau 1996). In this paper, we use a later formalization (Barták, Maillard, and Cardoso 2018; Barták et al. 2020) that is closer to a grammar view. For the description of valid action sequences, that is, the causal precondition/effect relations between actions, it uses the classical STRIPS approach (Fikes and Nilsson 1971). The specific plan structures are then expressed via task decomposition methods. In this section, we recall the formal definitions.

2.1 Sequential Planning

Sequential planning deals with sequences of actions transferring the world from a given initial state to a state satisfying certain goal conditions. In the classical STRIPS formulation (Fikes and Nilsson 1971), world states are modeled as sets of propositions that are true in those states. Let P be a set of all propositions modeling properties of world states. A state is modeled by a set of propositions $S \subseteq P$ that are true in that state (every other proposition is false). We use the notation $S^+ = S$ to describe the propositions valid in S and $S^- = P \setminus S$ to describe the propositions not valid in S.

Actions are used to describe state transitions under the control of an agent. Let A be a set of actions. Each action $a \in A$ is modeled by three sets of propositions (B_a^+, A_a^+, A_a^-), where $B_a^+, A_a^+, A_a^- \subseteq P, A_a^+ \cap A_a^- = \emptyset$. The set B_a^+ describes positive preconditions of action a, that is, propositions that must be true right before the action a. Action a is applicable to state S if and only if $B_a^+ \subseteq S$. Sets A_a^+ and A_a^- describe positive and negative effects of action a, that is, propositions that will become true or false in the state right after executing the action a. If an action a is applicable to state S then the state right after the action a is $\gamma(S, a) = (S \setminus A_a^-) \cup A_a^+$, and $\gamma(S, a)$ is undefined otherwise. An action sequence $a_1, \ldots, a_n$ is *causally consistent* with respect to the set of propositions S_0 (called an *initial state*), if it holds $B_{a_1}^+ \subseteq S_0$ and for each $i \in \{2, \ldots, n\}$ $B_{a_i}^+ \subseteq \gamma(...\gamma(\gamma(S_0, a_1), a_2), ..., a_{i-1})$. In other words, the precondition of each action is satisfied in the state right before the action.

2.2 Hierarchical Task Networks

Hierarchical planning puts an additional structure to plans. Specifically, actions in the plan are obtained from higher-level (compound) tasks that decompose to sub-tasks until primitive tasks – actions – are obtained. This way, the tasks provide recipes to achieve particular goals. Our formalization (Barták, Maillard, and Cardoso 2018; Barták et al. 2020) is loosely based on the original Hierarchical Task Networks formalization (Erol, Hendler, and Nau 1996) of hierarchical planning.

The recipe for a compound task T is represented as a task network – a set of sub-tasks to solve the task T together with the set of constraints between the sub-tasks. Let T be a compound task and $(\{T_1, ..., T_k\}, C)$ be a task network, where C are its *constraints* (defined later). We can describe the decomposition method as a derivation (rewriting) rule saying that T decomposes to sub-tasks $T_1, ..., T_k$:

$$T \rightarrow T_1, ..., T_k \ \ [C]$$

We should highlight that opposite to rewriting rules in formal grammars, the order of tasks on the right side of the rule does not matter here. The sub-task order is described explicitly by the precedence constraints in C (see below). Note also that a compound task may have several decomposition methods describing alternative ways to fulfill it. In fact, choosing "the right" method is where the computational complexity comes in within hierarchical plan generation (Erol, Hendler, and Nau 1996; Alford, Bercher, and Aha 2015a; Alford, Bercher, and Aha 2015b) or hierarchical plan verification (Behnke, Höller, and Biundo 2015; Bercher et al. 2016), similar to choosing the right action in classical planning. The task for which no decomposition method exists is called a *primitive task*. It corresponds to an action in sequential planning (it has preconditions and effects). The compound tasks correspond to non-terminal symbols and the primitive tasks to terminal symbols in the formal grammar terminology.

Let S_0 be an initial state and G be a goal task. By solving the HTN planning problem given by S_0 and G we mean decomposing the goal task G via decomposition methods until a set of primitive tasks – actions – is obtained, and linearly ordering these actions to satisfy all the constraints obtained during decomposition and to be a causally consistent with respect to the initial state S_0. Let a_i be the i-th action in this action sequence. The state right after the action a_i is denoted S_i. We denote the set of actions to which a task T decomposes as $act(T)$. If U is a set of tasks, we define $act(U) = \bigcup_{T \in U} act(T)$. The index of the first action in the decomposition of T is denoted $start(T)$, that is, $start(T) = \min_{a_i \in act(T)} (i)$. Similarly, $end(T)$ means the index of the last action in the decomposition of T, that is, $end(T) = \max_{a_i \in act(T)} (i)$.

Let us now formalize the constraints C used in the set of available decomposition methods M. They are the ones available in the original HTN formalization (Erol, Hendler, and Nau 1996), but have been re-introduced in other hierarchical planning formalizations as well (Xiao et al. 2017). The constraints can be of the following three types, where the first is also known as an ordering constraint and the latter two are essentially state constraints:

- $t_1 \prec t_2$: a *precedence* (also called *ordering*) *constraint* meaning that in every plan the last action obtained from task t_1 is before the first action obtained from task t_2, $end(t_1) < start(t_2)$,
- *before*(U, p): a *precondition constraint* meaning that in every plan the proposition p holds in the state right before the first action obtained from tasks U, $p \in S_{start(U)-1}$,
- *between*(U, V, p): a *prevailing constraint* meaning that in every plan the proposition p holds in all the states between the last action obtained from tasks U and the first action obtained from tasks V. That is, for all i: $end(U) \leq i < start(V)$ holds $p \in S_i$.

deliver → t_1:*unload*, t_2:*move*, t_3:*move*, t_4:*load*

[$t_2 \prec t_4, t_4 \prec t_3, t_3 \prec t_1$,
before($\{t_2\}$,empty),
between($\{t_4\}$,$\{t_1\}$, loaded)]

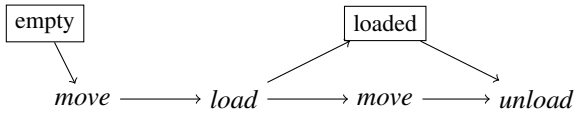

Figure 1: The figure shows (on top) a decomposition method for the task *deliver*. The order of sub-tasks on the right hand side of the method is irrelevant, as it might even be different from the imposed order in the constraints. At the bottom we depict the recipe described by the method graphically. Tasks are names without boxes, while the state constraints are depicted by boxes. The tasks they refer to are indicated with arrows. Example taken from (Barták et al. 2020)

The precedence constraints are used to define a specific order of sub-tasks. As an example, consider the *deliver* task. A decomposition method for it is depicted in Fig. 1. The task *deliver* may decompose to four sub-tasks: we first *move* the robot to a location, where we *load* an item to the robot, then the robot *moves* to the destination, where we *unload* the item. The precondition constraints are used in the same way as for actions. For example, before the robot starts moving to the place where the item is located, we may require the robot to be empty (note that this is not a precondition of task move, but it may be required for the move task used in the context of item delivery). The prevailing constraint is used to maintain some property between sub-tasks. For example, we may require the item to stay in the robot all the time between load and unload sub-tasks. Though the prevailing constraint was part of the original definition of HTN planning (Erol, Hendler, and Nau 1996), we are not aware of any HTN planning system that supports this constraint. However, they are used in more practical scenarios, for instance, for the generation of abstract solutions (de Silva, Padgham, and Sardina 2019). We regard this constraint as practically very useful, as demonstrated in the example above. Task interleaving allows the robot to do other tasks when doing the delivery task. For example, when going to the item's destination, the robot may visit another location, take a sample there, and then deliver the sample after delivering the item. Hence actions for the delivery task and for the take-sample task interleave in the final plan.

The *HTN plan verification problem* is formulated as follows: given a sequence of actions $a_1, a_2, \ldots, a_n$, and an initial state S_0, is the sequence of actions causally consistent with respect to S_0 and obtained from some compound task? Specifically, the action sequence must be obtained by decomposing any task and all its sub-tasks (no extra action is allowed in the plan). The action sequence must also satisfy all the constraints C from methods used in the decomposition. In some formulations, the goal task is given, and the action sequence must then be generated from that task.

3 HTN Plan Corrections

As of today, there exist two approaches for HTN plan verification. The system by Behnke, Höller, and Biundo (2017) translates the verification problem to a Boolean satisfiability problem. It does not support the before and between constraints[1] and requires the goal task to be given at input as well (because they verify whether the input plan is a solution to a given problem, rather than verifying whether it can be decomposed from some compound task). The parsing-based system by Barták, Maillard, and Cardoso (2018) supports all of the HTN constraints previously introduced. In the case of invalid plans, both approaches only report invalidity but no further details.

In the paper, we focus on what should be done with an invalid hierarchical plan. The core motivation is helping the user correct the plan by identifying and removing flaws in the plan. Let us first look at possible flaws in the hierarchical plan. Some action may miss its precondition. Missing precondition is the core type of flaw in classical planning (and easy to identify). However, in HTN planning, such an action may not be part of any task decomposition, so the missing precondition is not the problem. The presence of the action in the action sequence is the problem. Other flaws may happen due to the hierarchical structure. Some tasks may not be achievable via a specific method because some action is missing in the action sequence or some method constraint is violated. However, it is not clear if that task is needed as it may not be used in the root task spanning over the whole action sequence. Hence, rather than finding the violated constraints, we suggest telling the user how to modify the action sequence to achieve a valid hierarchical plan. There are basically two operations for the action sequence that cover all possible plan modifications – *action deletion* and *action addition*. Note that we can use deletion and addition operations also to change the order of actions. Recall that we are given a sequence of actions and an initial state, so if the action sequence is not a valid plan, we are looking for the closest action sequence, that is. The distance between action sequences can be defined as the minimal number of addition and deletion operations to transform one action sequence to the other one (note that such transformation is symmetrical). Hence, we want to modify the given action sequence as little as possible to get a valid action sequence. This approach seems like a reasonable suggestion in plan verification. If the action sequence is valid, then no modification is suggested. If the action sequence is invalid, then a minimal number of modifications are suggested to make the action sequence valid.

The above concept of plan modification can also be used in the task of the hierarchical plan and goal recognition (Höller et al. 2018). In this task, we observe actions, and we are trying to find what task is being achieved by these actions. We may observe actions not related to the task, so these actions should be deleted. We may miss an observation of some action (or that action has not yet been performed), so we need to add that action. Alternatively, we observed an action incorrectly, and we need to substitute it with the

[1] The before constraints can be compiled away.

correct action. This example illustrates that the concept of modifying an action sequence by action deletion and addition is generally helpful.

Action addition can be seen as a form of planning – what actions need to be added to form a valid plan. Hence, the plan correction problem would be undecidable if action addition is allowed. In this paper, we focus on modifying plans by action deletion only. This decision is motivated by assuming a complete action sequence that is not a hierarchical plan. We are looking for the largest (not necessarily contiguous) sub-sequence of actions that is a valid hierarchical plan. In other words, we want to delete the minimal number of actions from a given action sequence to obtain a valid plan. In terms of task recognition, we are looking for the task that spans over the largest number of observed actions. We formulate the problem of *HTN plan correction by action deletion* as follows: given a sequence of actions $a_1, a_2, \ldots, a_n$ and an initial state S_0, what is the largest (even non-contiguous) sub-sequence of actions causally consistent with respect to S_0 and obtained from some compound task? Notice that the HTN plan correction problem includes the HTN plan verification problem. If the plan is correct, then the correction algorithm returns that plan as this is the largest sub-sequence of actions that forms a valid plan. Note also, that the input action sequence may not be a plan at all as some actions' preconditions might be violated (the action sequence is not causally consistent with respect to S_0).

Although we are already given a sequence of actions to start with, it turns out that finding a maximally long sub-plan is still **NP-complete**. Before we show this, we will show that even just checking whether a given action sequence is a hierarchical plan is **NP-complete**. For this, we essentially have to show two things, namely that (1) the plan is executable and that (2) the plan can be obtained by some of the compound tasks. This is a variant of plan verification that was introduced by Barták, Maillard, and Cardoso (2018) and differs from the standard HTN plan verification as defined by Behnke, Höller, and Biundo (2015), in that the latter does not admit plans to be derived from *any* compound task in the *domain*, but from the *initial task network* of the *problem* only. Standard plan verification was proved to be **NP-complete**, but the hardness of the variant used here was still unknown. The hardness of our verification problem does not follow from the hardness of standard plan verification, although both are closely related. The hardness does, however, follow directly from the proof of one of the theorems by Behnke, Höller, and Biundo that was used as a basis to prove **NP-hardness** of plan verification in the absence of action preconditions or effects. We thus state this corollary and briefly explain why it holds.

Corollary 1 (Plan Verification Complexity)**.** *Given an HTN domain model, an action sequence $\vec{a} = a_1, \ldots, a_n$, and an initial state S_0, it is **NP-complete** to decide whether $\vec{a}$ is a hierarchical plan wrt. S_0 (i.e., deciding whether there exists a compound task in the model that decomposes into a task network that admits $\vec{a}$ as executable linearization) if the constraint sets only contain* precedence *constraints, and **NP-hard** otherwise.*

Proof. Membership: **NP** membership follows from the fact that standard plan verification as defined by Behnke, Höller, and Biundo is a special case of our plan verification variant (provided that the only constraints are *precedence* constraints, which correspond to ordering constraints in their formalism), as they have to verify that the input plan can be obtained by *one dedicated* compound task (the initial compound task), whereas we return true when there exists *any*. We can thus guess a compound task (or simply try all, as there are only linearly many) from the domain and test whether it can be decomposed into $\vec{a}$ using the technique by Behnke, Höller, and Biundo (2015) (cf. their Thm. 1). Return *yes* or *no* accordingly.

Hardness: Hardness follows directly from the proof of Thm. 2 by Behnke, Höller, and Biundo (2015). That theorem states the **NP**-hardness of VERIFYSEQ (Def. 12), the problem of determining whether for a given action sequence $\vec{a}$ there exists a solution task network tn_S, such that $\vec{a}$ is a linearization of tn_S. Their proof reduces from the NP-complete Vertex Cover (Karp 1972). In the planning problem they construct, the only possibility of obtaining the action sequence $\vec{a}$ is from their initial task t_I. In other words, given their planning domain, the only possible task that allows to obtain $\vec{a}$ is exactly the initial task t_I of their construction. Thus, this compound task is the only one allowing to generate $\vec{a}$ thus showing the claim. □

Knowing the complexity of our variant of the HTN verification problem is not just interesting from a theoretical point of view, but it also allows us to infer the complexity of the optimization problem we are solving in this paper:

Theorem 1 (Plan Correction Complexity)**.** *Given an HTN domain model, an action sequence $\vec{a}$, an initial state S_0, and a natural number $k > 0$, the problem of deciding whether k or fewer actions can be deleted from $\vec{a}$, such that the resulting sequence is a hierarchical plan, is **NP-complete** if the constraint sets only contain* precedence *constraints, and **NP-hard** otherwise.*

Proof. Membership: We first guess the number of actions k' we delete, $1 \leq k' \leq k$. Then guess k' numbers (positions in the action sequence) between 1 and $n = |\vec{a}|$. Note that although the input number k can be encoded logarithmically, we are not in risk of incurring an exponential runtime increase, since k can be bounded by n, the size of $\vec{a}$, whereas the entire sequence $\vec{a}$ is given in the input and thus linear in size. Delete the respective actions from $\vec{a}$ obtaining the sequence $\vec{a}'$ of length $n - k'$. Now check whether $\vec{a}'$ is a hierarchical plan, which is in **NP** if there are no state constraints (Cor. 1). Return *yes* if and only if the answer is *yes*.

Hardness: We reduce from our verification problem, which is **NP-hard** (Cor. 1). Thus, let $\vec{a} = a_1, \ldots, a_n$ be given, and the question (we reduce from) is whether $\vec{a}$ is executable in S_0 and can be obtained from some compound task. We reduce this to our problem by first assuming the exact same HTN domain, but we introduce one new action a_{n+1} without preconditions or effects that is not contained in any method. We construct the sequence

$\vec{a}' = a_1, \ldots, a_n, a_{n+1}$ and ask with $k = 1$ whether there exists a sub sequence of $\vec{a}'$ with k or less actions deleted from it, such that the resulting sequence is a hierarchical plan. Since a_{n+1} is not contained in any method, we know that $\vec{a}'$ cannot be a hierarchical plan, so we must delete at least one action, which must be a_{n+1}. Thus, the answer to the constructed problem is *yes* if and only if $\vec{a}$ was a hierarchical plan. □

Note that we expect that the problem will remain in **NP** if there are state constraints. The proof would be however overly technical (since the inheritance of all constraints had to be handled correctly), which will remain future work.

NP-completeness in the absence of state constraints was expected given that correcting a plan essentially contains verifying the new plan, and given that many variants of HTN change requests were shown to be **NP-complete** (Behnke et al. 2016).

4 Correction Algorithm

We shall now describe a novel algorithm that deals with invalid HTN plans and corrects them by deleting superfluous actions. What we do can be regarded as plan repair. There exist approaches for repairing HTN plans, but they do so because of an unexpected *execution* error, that is, due to an exogenous change to the current world state "outside of the model" (Warfield et al. 2007; Höller et al. 2020b; Goldman, Kuter, and Freedman 2020). This view poses different constraints, such as starting with a plan that was a solution in the first place, maintaining the actions already executed, and repairing the rest of the plan to fulfill the original goal task. Sometimes the repaired plans do not even need to be hierarchical plans (that is, they do not always have to lie within the decomposition hierarchy). Thus, none of these approaches is suited for our purpose, and we talk about plan correction rather than plan repair.

Our approach finds a minimal set of actions deleted from a given action sequence to obtain a valid hierarchical plan. We will base the algorithm on the parsing technique derived from the parsing-based validator (Barták et al. 2020) for the following reasons. It is currently the only technique that supports all constraints in the HTN model, and it uses a greedy approach that finds *all* tasks that decompose to some actions in the input action sequence. We need to find even more such tasks during plan correction as violations of some state constraints can be ignored if the constraints can be satisfied by deleting some actions.

The parsing-based plan validator works as follows. It first checks the causal consistency of the given action sequence. If it is inconsistent, then the plan is invalid. If the action sequence is causally consistent, the algorithm attempts to find a task that decomposes to the action sequence. It starts with a set $Tasks$ of primitive tasks – actions from the input action sequence. Then it finds a method to compose a new task. Specifically, if there is a method $T \rightarrow T_1, ..., T_k \ [C]$, such that all tasks T_i are in the set $Tasks$ and all the constraints C are satisfied, the task T is added to the set $Tasks$. This process is repeated until a task spanning over the whole action sequence is found, then the plan is valid, or no new task is introduced, then the plan is invalid.

When action deletion is allowed in the action sequence, we need the following extensions of the above parsing approach. First, when checking causal consistency and some action precondition is not satisfied, we find a preceding support action providing the precondition and delete all actions threatening this causal link (Weld 1994). Let $p \in B^+_{a_i}$ be a precondition of action a_i. We need to find a support for this precondition, which is either some action a_j, such that $j < i \wedge p \in A^+_{a_i}$, or the initial state, if $p \in S_0$ (then $j = 0$). Moreover, all actions a_k such that $j < k < i \wedge p \in A^-_{a_i}$ must be deleted to ensure that the causal link between a_j (or S_0) and a_i is not broken. We start with the support closest to a_i and if this support cannot be used (for example, because it deletes another precondition of the action) we look for another support earlier in the plan. To find supports we introduce a set called *timeline* that describes how actions in the plan modify a given proposition p:

$$tln(p) = \{(+i)|p \in A^+_{a_i}\} \cup \{(-i)|p \in A^-_{a_i}\} \cup \{0|p \in S_0\}$$

Basically, the timeline is a set of indexes of actions that change the validity of the proposition. Figure 2 gives the timelines for a simple package delivery plan. a, b, c are possible locations, and p, q are packages.

In the algorithm's initial stage, we generate primitive tasks for actions in the input action sequence and ensure that each such primitive task has supports for all its preconditions. Each task T is described as a 5-tuple $(T, beg, end, idx, del, sup)$, where:

- idx = indexes of actions to which the task T decomposes,
- beg = index of the first action to which T decomposes,
- end = index of the last action to which T decomposes,
- del = action indexes to be deleted to decompose T,
- sup = set of supports; each support is a pair (tln, j), where $j \in tln$ is an index of an action providing some condition of the task (if we have the timeline, we do not need the proposition p anymore) and that action is not included in the task decomposition (in idx).

Alg. 1 shows the initialization stage that converts the action sequence to a set of primitive tasks that will later be merged to compound tasks via methods. For each action, the algorithm finds a set of supports and a set of actions to be

timeline	Init	move$_1$(a,b)	load$_2$(b,p)	load$_3$(b,q)	move$_4$(b,c)	move$_5$(c,a)	unload$_6$(c,p)
empty	0		-2	-3			+6
loaded(p)			+2				-6
loaded(q)				+3			
on(b,p)	0		-2				
on(c,p)							+6
on(b,q)	0			-3			
at(a)	0	-1				+5	
at(b)		+1			-4		
at(c)					+4	-5	

Figure 2: Example of timelines describing effects of actions.

Algorithm 1 Initialization of primitive tasks

Input: initial state S_0 and action sequence $\vec{a} = \{a_1, \ldots, a_n\}$
Output: set of primitive tasks and their timelines
forall $p \in P$ **do**
  $tln(p) \leftarrow \{i \mid p \in A^+_{a_i}, a_i \in \vec{a}\} \cup \{(-i) \mid p \in A^-_{a_i}, a_i \in A\} \cup \{0 \mid p \in S_0\}$
$Tasks \leftarrow \emptyset$
forall $a_i \in \vec{a}$ **do**
  $idx \leftarrow \{i\}, beg \leftarrow i, end \leftarrow i, del \leftarrow \emptyset, sup \leftarrow \emptyset$
  $check \leftarrow \{tln(p) \mid p \in B^+_{a_i}\}$ `// support is needed for these preconditions`
  for $j = i - 1$ **downto** 1 **do**
    if $\exists tln \in check$, *s.t.* $(-j) \in tln$ **then**
      `// action` a_j `deletes condition`
      $del \leftarrow del \cup \{j\}$
    else
      forall $tln \in check$, *s.t.* $j \in tln$ **do**
        `// action` a_j `provides condition`
        $sup \leftarrow sup \cup \{(tln, j)\}$
        $check \leftarrow check \setminus \{tln\}$
  if $\{tln \mid tln \in check, 0 \notin tln\} = \emptyset$ **then**
    `// remaining preconditions are supported by the initial state`
    $Tasks \leftarrow Tasks \cup \{(a_i, beg, end, idx, del, sup)\}$
return $Tasks$

deleted as these actions break the causal links in supports. If no support is found for action, this action is not assumed (it is not used to compose tasks). Note that not every primitive task can actually be used in the plan as it is not checked yet if its supports are also supported. Action inclusion will be decided in the second stage of the algorithm, where primitive tasks will be merged to compound tasks via methods. Figure 3 shows how the primitive tasks look like after the initialization stage.

As mentioned above, the parsing-based approach to hierarchical plan validation progressively groups sub-tasks via methods to get compound tasks until a task spanning over all primitive tasks – actions – is found. We will now describe this grouping step for a decomposition method $T \rightarrow T_1, \ldots, T_k \;\; [C]$, where the sub-tasks $T_1, \ldots, T_k$ were already obtained, that is, $(T_i, beg_i, end_i, idx_i, del_i, sup_i) \in Tasks$. Each sub-task T_i describes over which actions it spans (idx_i) and which actions must be deleted (del_i) to make the sub-task decomposable to actions in the plan. To obtain a proper hierarchy, sub-tasks must decompose to disjoint sets of actions ($\forall i \neq j : idx_i \cap idx_j = \emptyset$) and deleted actions cannot be part of the task ($\bigcup_i idx_i \cap \bigcup_i del_i = \emptyset$). The method constraints are checked as follows. The *precedence* constraints are not influenced by action deletion and they can be checked immediately. The *before* constraints behave identically to preconditions of actions, so they are handled similarly to action preconditions. The *between* constraint requires some proposition to be true in all states between two actions a_{i_1} and a_{i_2} (these actions are defined by

Algorithm 2 Task composition

Input: A method $T \rightarrow T_1, \ldots, T_k \;\; [C] \in M$ with $[C] = [Prec, Before, Between]$, where $(T_i, beg_i, end_i, idx_i, del_i, sup_i) \in Tasks$
Output: $\emptyset$ if task T cannot be decomposed, or a new task otherwise.
if $\exists i \neq j$, *s.t.* $idx_i \cap idx_j \neq \emptyset$ **then return** $\emptyset$
if $\exists (i, j) \in Prec$, *s.t.* $beg_j \leq end_i$ **then return** $\emptyset$
$idx \leftarrow \bigcup_i idx_i$, $beg \leftarrow \min(idx)$, $end \leftarrow \max(idx)$, $del \leftarrow \bigcup_i del_i$
$check \leftarrow \{(tln, j) \mid (tln, j) \in \bigcup_i sup_i, j \notin idx\}$ `// supports to be checked`
forall $(U, p) \in Before$ `// find initial support for before conditions` **do**
  for $j = \min_{i \in U}(beg_i) - 1$ **downto** 1 **do**
    if $(-j) \in tln(p)$ **then**
      `// action` a_j `deletes condition`
      $del \leftarrow del \cup \{j\}$
    else if $j \in tln(p)$ **then**
      `// action` a_j `provides condition`
      $check \leftarrow check \cup \{(tln(p), j) \mid j \notin idx\}$
      continue with next *before* condition
  if $0 \notin tln(p)$ **then return** $\emptyset$ `// before condition cannot be satisfied`
forall $(U, V, p) \in Between$ `// find initial support for between conditions` **do**
  $st \leftarrow \max_{i \in U}(end_i)$
  for $j = \min_{i \in V}(beg_i) - 1$ **downto** 1 **do**
    if $(-j) \in tln(p)$ **then**
      `// action` a_j `deletes condition`
      $del \leftarrow del \cup \{j\}$
    else if $j \in tln(p) \wedge j \leq st$ **then**
      `// action` a_j `provides condition`
      $check \leftarrow check \cup \{(tln(p), j) \mid j \notin idx\}$
      continue with next *between* condition.
  if $0 \notin tln(p)$ **then return** $\emptyset$ `// between condition cannot be satisfied`
$sup \leftarrow \emptyset$ `// update all supports so no support is among deleted actions`
for $j = n$ **downto** 1`//` n `is the number of actions in the input plan` **do**
  if $j \in del$ **then continue**
  if $\exists (tln, i) \in check$, *s.t.* $j < i \wedge (-j) \in tln$ **then**
    `// action` a_j `deletes condition`
    $del \leftarrow del \cup \{j\}$
  else
    forall $(tln, i) \in check, s.t.\ j \leq i, j \in tln$ **do**
      `// action` a_j `provides condition`
      $sup \leftarrow sup \cup \{(tln, j) \mid j \notin idx\}$
      $check \leftarrow check \setminus \{(tln, i)\}$
if $idx \cap del \neq \emptyset$ **then return** $\emptyset$
if $\{(tln, i) \mid (tln, i) \in check, 0 \notin tln\} = \emptyset$ **then**
  return $(T, beg, end, idx, del, sup)$
else return $\emptyset$

two sets U and V of tasks). Hence, the proposition must be true in the state right after action a_{i_1}, which can again be handled similarly to action preconditions. If some action between a_{i_1} and a_{i_2} violates (deletes) the proposition, this action must be deleted (cannot be part of a plan for the task). All these state-based constraints are handled using the sets of supports as described above.

As the supports are initially checked for each constraint independently, and there are also supports connected to each sub-task, we need to update all the supports. The reason is that a particular action can be a support in one constraint while deleted to satisfy another constraint. This update is realized by going right to left in the action sequence, and if some support action is deleted, then another support is looked for in the corresponding timeline. If no support is found, then the task cannot be accepted as some state constraint is violated, and this violation cannot be corrected by action deletion. Finally, note that if some support is part of the task, this support is granted (the action providing the support cannot be deleted), so the support can be omitted from further checks. The above process is described in Alg. 2.

When composing a new task, only conflicting actions are deleted to ensure that the task can decompose to some subset of actions from the input action sequence. This minimizes the number of action deletions. The compound task can be assumed in the plan only if all its supports are included.

Suppose we found a compound task T, whose set of supports sup is empty, or all current supports can be substituted by other actions from the task or by the initial state. In that case, we have a root task that decomposes to a subset of actions from the input plan (identified by the set idx), and this subset forms a valid plan. This property is checked by Alg. 3. It may happen that $idx \cup del \neq \{1, \ldots, n\}$ so there are still some actions not covered by the task and not required to be deleted. These actions can be omitted from the action sequence, and we still get a valid plan as these actions do not support any state-based constraint from the task. To correct the plan optimally, we are looking for a root task with the largest set idx. There might be more such tasks and the proposed technique finds them all thanks to its greedy character. Alg. 4 describes how to greedily generate all tasks until the root task with the minimal number of deleted actions is found.

Figure 3 shows a decomposition tree with the task $deliver(b, c, p)$. Preconditions of all involved actions (and before and between constraints) are satisfied either by actions in the task or by the initial state. Hence the support set is empty. Action $move_5(c, a)$ needs to be deleted because it violates the precondition $at(c)$ of action $unload_6(c, p)$. Action $load_3(b, q)$ is not included, but it can be deleted as it does not interfere with the task's actions. Hence task $deliver(b, c, p)$ is a possible root task.

Theorem 2 (Soundness and completeness)**.** *Alg. 4 is sound and complete in the sense of returning a task that decomposes to a valid plan that is obtained from the given action sequence by a minimal number (≥ 0) of action deletions.*

Proof. First, each action in the primitive task is executable if its supports are present in the plan and no action that breaks the causal link from the support is present. This property is easy to verify when the initialization stage introduces the primitive tasks. If a compound task includes a subtask whose supports were deleted during the composition, the algorithm finds other supports. If no such support exists, then the compound task cannot be accepted. For each root task, either the support set is empty, or the supports can be substituted by supports from the task or the initial state. This property justifies that the set of actions over which the root task spans is causally consistent.

The root task decomposes to actions described by indexes in its idx set as the task was obtained by the reverse application of the decomposition methods. Moreover, all the method constraints are satisfied. The precedence constraints were checked during task composition (and they are not affected by action deletions). The before and between constraints were satisfied using the same mechanism as for action preconditions described above. Hence, the root task de-

Algorithm 3 Support check

Input: set sup of supports, set idx of action indexes from the task
Output: $True$ if supports can be found among actions in the task or in the initial state; $False$ otherwise

```
forall (tln, k) ∈ sup do
    for j = k − 1 downto 1 do
        if j ∈ tln ∧ j ∈ idx then
            continue with next timeline.
        if (−j) ∈ tln ∧ j ∈ idx then
            return False
    if 0 ∉ tln then
        return False
return True
```

Algorithm 4 Plan repair

Input: initial state S_0, action sequence $\vec{a} = \{a_1, \ldots, a_n\}$, and an HTN domain model
Output: task T that decomposes to the largest subsequence of actions forming a valid plan or *Fail* if no such task exists

```
New ← Initialization(S_0, a⃗) // Alg.1
Tasks ← ∅, max_d ← n
while New ≠ ∅ do
    Tasks ← Tasks ∪ New, New ← ∅
    forall methods T → T_1, ..., T_k [C], s.t.
      (T_i, beg_i, end_i, idx_i, del_i, sup_i) ∈ Tasks do
        Tk ← Compose(T → T_1, ..., T_k [C]) // Alg.2
        if Tk = (T, beg, end, idx, del, sup) s.t. |del| ≤
          max_d ∧ Tk ∉ Tasks) then
            New ← New ∪ {Tk}
            if SupportCheck(sup, idx) then
                // root task; Alg.3
                max_d ← min(max_d, n − |idx|)
if max_d < n then
    return Tk = (T, beg, end, idx, del, sup) ∈ Tasks, s.t.
      SupportCheck(sup, idx) ∧ n − |idx| = max_d
else return Fail
```

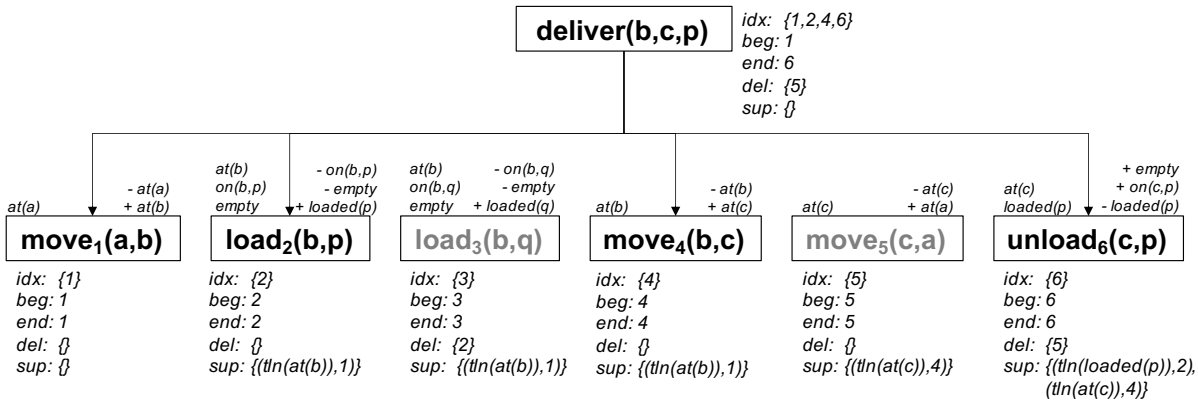

Figure 3: Decomposition tree. Action preconditions and effects are shown above the actions; Task data structure is below the actions.

composes to a sub-sequence of actions from the input action sequence, and this sub-sequence forms a valid plan. Hence the algorithm is sound.

The number of different tasks is finite (and this is true even if recursive methods are present) as the input action sequence's length is finite. Hence the algorithm must finish sometime as only new tasks are added during parsing. The algorithm also remembers the minimal number of deleted actions for all root tasks that decompose to a valid sub-plan of the input plan, and it returns one of these tasks. It may, however, happen that no root task is found. Because of the algorithm's greedy character, no sub-sequence of the input plan forms a valid HTN plan. □

5 Empirical Evaluation

As of now, there exists no other technique that can deal with failed plan verification. To evaluate our approach, we empirically compare the novel algorithm with the currently fastest HTN plan validation algorithm by Barták et al. (2020). We will call this validation algorithm *pure validator*, while our novel technique will be called *extended validator*. By this comparison, we shall see the new method's overhead over the technique that only identifies invalid plans but does not correct them. When the plan is valid, both algorithms give the same answer. Otherwise, the extended validator will provide the task that decomposes to the longest possible subset of actions as opposed to the pure validator, which simply states that "Plan is invalid".

All the experiments run under 64-bit Windows 10 on Intel Core i7 7700 processor and 16GB RAM. Both algorithms were implemented in C# 7 (from .NET 4.7), and they use the new PDDL-like representation HDDL (Höller et al. 2020a) that is also used by the PANDA planners and verifier (Behnke, Höller, and Biundo 2017), as well as for the International Planning Competition (IPC 2020) on HTN planning (Behnke et al. 2019). The code is available at https://github.com/siprog/PlanCorrectionKR2021. We compared both algorithms on four different domains: PCP, Transport, Kitchen, and Satellite. In each domain, we tested ten valid plans generated by the PANDA progression search and SAT-based planners. All of these plans were also used previously to compare pure validation techniques (Barták et al. 2020). We created five invalid plans for each domain by manually adding 1 to 5 additional actions that must be deleted to obtain a valid plan. In total, we used 60 problem instances.

The plan length for valid plans ranges from 2 to 49 actions (PCP 10-30, Transport 2-10, Kitchen 16-49, and Satellite 2-15). As the results in Fig. 4a show, both algorithms are comparable in speed, and the runtimes are very close to each other. This result was expected as no actions need to be deleted. This experiment also shows that the overhead to maintain timelines and supports is not significant. For the PCP domain, neither algorithm solved one of the instances within the given time limit (runtime is 300s in the graph).

For invalid instances, the plan length ranged from 15 to 53 actions (PCP: 27-31, Transport: 15-19, Kitchen: 49-53, and Satellite: 20-24). We have given both algorithms a time limit of 300 seconds. There are two main reasons of plan invalidity. Either the plan is not causally consistent, or no task decomposes to a given sequence of actions. The PCP and Transport domains represent the causally inconsistent plans – at least one action has violated preconditions. The pure validator identified that the plan is not causally consistent and stopped without ever trying to build the task structure on top of the actions. This observation explains runtimes close to zero as verifying causal consistency requires linear time with respect to the plan length. Figure 3 gives an example of a causally inconsistent action sequence. The extended validator is able to correct such plans by removing the extra actions. This approach naturally takes more time as the extended validator builds the task structure even if the action sequence is causally inconsistent.

The Satellite and Kitchen domains contain causally consistent plans, but no task decomposes to all actions. As we can see in Fig. 4b both algorithms have comparable runtimes there. This result is encouraging, taking into account that the extended validator also finds a valid sub-plan. This demonstrates that most information necessary to identify the flaws is already present in the task descriptions generated by the original validation algorithm. Moreover, the result confirms that the overhead to maintain the timelines and supports is not significant.

(a) Comparison of runtimes (seconds) as a function of plan length (valid plans).

(b) Comparison of runtimes (seconds) as a function of the number of actions to be deleted (invalid plans).

Figure 4: Comparison of runtimes for valid and invalid plans, respectively.

6 Conclusions

We proposed the first approach to correct invalid hierarchical plans by action deletion, thus dealing with failed plan verification in HTN planning. We proved that the problem of finding a corrected plan with the minimal number of deletions is **NP-complete** (and also that pure HTN verification is **NP-complete**) in the absence of state constraints and **NP-hard** otherwise, and we proposed a parsing-based algorithm that finds such a plan efficiently. Empirical comparison with the fastest HTN plan validator showed that the new method's overhead over the pure validation is minimal. Hence, the new technique can be used for HTN plan validation with the additional benefit of suggesting how to correct the invalid plan and explaining the reason for action deletion.

Acknowledgments

This research was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215, and by the joint Czech-German project registered under the number 21-13882J by the Czech Science Foundation (GAČR) and BE 7458/1-1 by Deutsche Forschungsgemeinschaft (DFG). Simona Ondrčková is (partially) supported by SVV project number 260 575.

References

Alford, R.; Bercher, P.; and Aha, D. 2015a. Tight bounds for HTN planning. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 7–15. AAAI Press.

Alford, R.; Bercher, P.; and Aha, D. 2015b. Tight bounds for HTN planning with task insertion. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, 1502–1508. AAAI Press.

Avrahami-Zilberbrand, D., and Kaminka, G. A. 2005. Fast and complete symbolic plan recognition. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, 653–658. Morgan Kaufmann Publishers Inc.

Barták, R., and Maillard, A. 2017. Attribute grammars with set attributes and global constraints as a unifying framework for planning domain models. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP 2017)*, 39–48. ACM.

Barták, R.; Ondrčková, S.; Maillard, A.; Behnke, G.; and Bercher, P. 2020. A novel parsing-based approach for verification of hierarchical plans. In *Proceedings of the 32th International Conference on Tools with Artificial Intelligence (ICTAI 2020)*, 118–125. IEEE.

Barták, R.; Ondrčková, S.; Behnke, G.; and Bercher, P. 2021. On the verification of totally-ordered HTN plans. In *Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)*.

Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of hierarchical plans via parsing of attribute grammars. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 11–19. AAAI Press.

Behnke, G.; Höller, D.; Bercher, P.; and Biundo, S. 2016. Change the plan – how hard can that be? In *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS 2016)*, 38–46. AAAI Press.

Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; Pellier, D.; Fiorino, H.; and Alford, R. 2019. Hierarchical planning in the ipc. In *Proceedings of the Workshop on the International Planning Competition*.

Behnke, G.; Höller, D.; and Biundo, S. 2015. 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 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 *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 20–28. AAAI Press.

Bercher, P.; Alford, R.; and Höller, D. 2019. A survey on hierarchical planning – one abstract idea, many concrete realizations. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 6267–6275. IJCAI.

Bercher, P.; Höller, D.; Behnke, G.; and Biundo, S. 2016. More than a name? on implications of preconditions and effects of compound HTN planning tasks. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*, 225–233. IOS Press.

de Silva, L.; Padgham, L.; and Sardina, S. 2019. HTN-like solutions for classical planning problems: An application to bdi agent systems. *Theoretical Computer Science* 763:12–37.

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.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2nd International Joint Conference on Artificial intelligence (IJCAI 1971)*, 608–620.

Geib, C. W., and Goldman, R. P. 2009. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence* 173(11):1101–1132.

Ginsberg, M. L. 1986. Counterfactuals. *Artificial Intelligence* 30(1):35–79.

Goldman, R. P.; Kuter, U.; and Freedman, R. G. 2020. Stable plan repair for state-space HTN planning. In *Proceedings of the 3rd ICAPS Workshop on Hierarchical Planning (HPlan 2020)*, 27–35.

Hoang, H.; Lee-Urban, S.; and Muñoz-Avila, H. 2005. Hierarchical plan representations for encoding strategic game ai. In *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005)*, 63–68. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proceedings of the 21st European Conference on Artificial Intelligence (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 *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS 2016)*, 158–165. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2018. Plan and goal recognition as HTN planning. In *Proceedings of the 30th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2018)*, 466–473. IEEE.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020a. HDDL: An extension to PDDL for expressing hierarchical planning problems. In

Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020), 9883–9891. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020b. HTN plan repair via model transformation. In *Proceedings of the 43th German Conference on AI (KI)*, 88–100. Springer.

Kabanza, F.; Filion, J.; Benaskeur, A. R.; and Irandoust, H. 2013. Controlling the hypothesis space in probabilistic plan recognition. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2306–2312. AAAI Press.

Kaelbling, L. P., and Lozano-Pérez, T. 2011. Hierarchical task and motion planning in the now. In *2011 IEEE International Conference on Robotics and Automation (IROS 2011)*, 1470–1477. IEEE.

Karp, R. M. 1972. Reducibility among Combinatorial Problems. In Miller, R. E.; Thatcher, J. W.; and Bohlinger, J. D., eds., *Complexity of Computer Computations*, The IBM Research Symposia Series. Springer US. 85–103.

Lin, S., and Bercher, P. 2021a. Change the world – how hard can that be? On the computational complexity of fixing planning models. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI 2021)*. IJCAI.

Lin, S., and Bercher, P. 2021b. On the computational complexity of correcting HTN domain models. In *Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)*.

Mirsky, R.; Gal, Y. K.; and Shieber, S. M. 2017. CRADLE: an online plan recognition algorithm for exploratory domains. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8(3):45:1–45:22.

Neufeld, X.; Sanaz.Mostaghim; Sancho-Pradel, D.; and Brand, S. 2019. Building a planner: A survey of planning systems used in commercial video games. *IEEE Transactions on Games* 11(2):91–108.

Tate, A. 1977. Generating project networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI 1977)*, 888–893.

Vilain, M. 1990. Getting serious about parsing plans: A grammatical analysis of plan recognition. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI 1990)*, 190–197. AAAI Press.

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

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

Wilkins, D. E. 1984. Domain-independent planning representation and plan generation. *Artificial Intelligence* 22(3):269–301.

Xiao, Z.; Herzig, A.; Perrussel, L.; Wan, H.; and Su, X. 2017. Hierarchical task network planning with task insertion and state constraints. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 4463–4469. IJCAI.