A Look-Ahead Technique for Search-Based HTN Planning: Reducing the Branching Factor by Identifying Inevitable Task Refinements

Conny Olz[1] **and Pascal Bercher**[2]

[1] Ulm University
[2] School of Computing, The Australian National University
conny.olz@uni-ulm.de, pascal.bercher@anu.edu.au

Abstract

In HTN planning the choice of decomposition methods used to refine compound tasks is key to finding a valid plan. Based on inferred preconditions and effects of compound tasks, we propose a look-ahead technique for search-based total-order HTN planning that can identify inevitable refinement choices and in some cases dead-ends. The former occurs when all but one decomposition method for some task are proven infeasible for turning a task network into a solution, whereas the latter occurs when all methods are proven infeasible. We show how it can be used for pruning, as well as to strengthen heuristics and to reduce the search branching factor. An empirical evaluation proves its potential as incorporating it improves an existing HTN planner such that it is the currently best performing one in terms of coverage and IPC score.

Introduction

Hierarchical Task Network (HTN) planning aims to find an executable sequence of actions that is a refinement of some initial abstract tasks (Erol, Hendler, and Nau 1996; Ghallab, Nau, and Traverso 2004; Alford, Bercher, and Aha 2015). HTN planning problems can be solved in various ways (Bercher, Alford, and Höller 2019), but one of the most successful ones at the moment is phrasing it – just like in classical planning – as a heuristic search problem (Höller and Behnke 2021). In a progression search-based HTN planner search nodes consist of a current state and a task network, which is a collection of primitive and/or abstract tasks (Nau et al. 2003; Höller et al. 2020b). When a search node is selected from the fringe new search nodes are generated by always processing the *first* task of the task network, i.e. primitive ones are applied and change the current state, compound ones are decomposed according to their decomposition methods producing usually multiple new search nodes.

As a consequence of this strict *progression* procedure, inevitable decomposition choices for compound tasks that occur later in the current search node (non-first tasks) cannot be chosen earlier. So, if we knew due to some reasoning process which (single) decomposition method of some compound task had to be chosen, we could still only choose it once its compound task has become the first, i.e., after all its predecessor tasks have been processed. Performing such a unique choice early on in the back part of the search node can however have a great impact on the choices that have to be made in the front part, most notably, performing these inevitable decompositions early on will make heuristics more accurate, because they do not become "distracted" by the other method choices that have been shown infeasible.

For the setting of totally ordered HTN planning, we propose a look-ahead technique that detects cases when only one method of a compound task may lead to a solution so that we can decompose them right away – even if they are non-first tasks. Additionally, as a special case, we can detect dead-ends when all decomposition methods for a task are detected infeasible. The first case is put into practice by generalizing the standard HTN progression algorithm by allowing to decompose non-first tasks right away (for all tasks with only one feasible method). Our look-ahead technique exploits so-called mandatory preconditions of compound tasks and methods as well as (positive and negative) effects (state changes) that will occur due to the decomposition of a compound task, which can be computed in polynomial time (Olz, Biundo, and Bercher 2021). Our technique, therefore, does not rely on any of the HTN heuristics' typical problem relaxations: delete-relaxation, ordering-relaxation, task insertion, or task sharing (Alford et al. 2014; Bercher et al. 2017; Höller et al. 2020b; Höller, Bercher, and Behnke 2020; Höller and Bercher 2021). The technique thus nicely *complements* existing heuristics as it relies on different problem relaxations thus strengthening current reasoning techniques. It further reduces the size of the search space by detecting previously unrecognized dead-ends, while also reducing the branching factor and making heuristics more informed due to eliminating provably infeasible method choices.

We implemented the look-ahead technique as well as the generalized progression procedure on top of the PANDA progression planner (Höller and Behnke 2021; Höller et al. 2020b). We conducted a comprehensive empirical evaluation on the benchmark set of the International Planning Competition (IPC) 2020, which shows that our proposed technique outperforms the state of the art in terms of solved instances and IPC score.

Related Work

Reasoning on an abstract level has been done before. Loosely related work is on the *downward-unlinearizable* cri-

terion for partially ordered (p.o.) task networks (Yang 1990), which tries to identify situations in which an abstract p.o. plan can not be turned into a solution because of unresolvable ordering constraints. Also in the context of p.o. task networks *external conditions* (i.e., constraints similar to method preconditions) were used to prioritize which abstract task is decomposed next during search to heuristically avoid backtracking (Tsuneto, Hendler, and Nau 1998). We however deal with totally ordered task networks in the first place, so these results do not transfer.

Summary information (including among others pre-, in-, and postconditions) is a concept similar to external conditions. They have been used to reason about abstract plans in a non-recursive, temporal, partial-order multi-agent setting (Clement, Durfee, and Barrett 2007), as well as to improve the performance of Belief-Desire-Intention (BDI) systems, for example, by detecting conflicts between an agent's top-level goals (de Silva, Meneguzzi, and Logan 2020).

Marthi, Russell, and Wolfe (2007) present a top-down (depth-first) search algorithm that also takes advantage of preconditions and effects of compound tasks (which they call high-level actions, HLAs). One main difference to our approach is that these preconditions and effects are given in their input. The authors do not explain how they could be inferred or verified. Another difference of major importance is that they do not adhere the standard HTN semantics. They assume that the hierarchy does not generate primitive non-solutions since in their case primitive actions just vanish if their preconditions are not met in a state. This changes the standard HTN semantics heavily and likely also the complexity of the plan existance problem so that results are not fully comparable anymore.

The lifted planner HyperTensioN (Magnaguagno, Meneguzzi, and de Silva 2021, 2022) finds *preconditions* similar to Olz, Biundo, and Bercher's executability-relaxed preconditions (Olz, Biundo, and Bercher 2021) (which we exploit in this paper) in a preprocessing step called "Pullup". HyperTensioN exploits those preconditions in a lifted and blind (depth-first) progression search, whereas we exploit them to improve a heuristic progression search, specifically, among others, heuristic accuracy. In particular, HyperTensioN uses them in two ways: 1) *Before search* for model reduction, if preconditions of tasks or methods (no matter where they appear in the domain) are never satisfiable, the task or method gets removed completely. This should be compared to preprocessing procedures like, e.g., the PANDAPIGROUNDER (Behnke et al. 2020). It already compiles predefined method preconditions into the model (by means of artificial actions) and performs some reachability analyses based on the hierarchy, delete-relaxation, and task insertion. The grounder could potentially be improved by incorporating inferred preconditions. However, this is out of scope of this paper as we are not aiming at domain model reductions, but on how search can be improved. 2) *During search* for each search node HyperTensioN checks methods' preconditions of the *first* tasks (i.e., the first task in the sequence of tasks of the respective task network) with respect to the current search node's *initial* state before decomposition to identify dead-ends. In contrast, we check (also for each search node during search) method applicability of *all* tasks, including non-first ones to ignore those that are inapplicable. In order to do so, we propagate different kinds of (inferred) effects through the entire task network to check for executability issues of any intermediate state. Therefore we may find more dead-ends than HyperTensioN because of inapplicable actions in the middle and back of the task network. Additionally, this can lead to early refinements, which HyperTensioN is not capable of. Examples 1 and 2 illustrate both cases.

In SAT-based planners the encoding is based on a tree representing all possible refinements up to some depth limit together with the executability constraints of the primitive tasks in the leaf nodes. The problem is solved incrementally by increasing the depth bound. The lifted SAT-based planner Lilotane (Schreiber 2021) incorporates some reachability analysis to prune unreachable actions or methods based on preconditions similar to the ones used by us. To check their satisfiability, mentioned effects (effects that occur somewhere in the children of a task) were computed for every new depth layer. It is a weaker check compared to our approach since the interplay with delete effects is not taken into account and the problem is considered as a whole with the initial state and all possible refinement choices, whereas in our case tasks are already decomposed and applied.

Theoretical Background

Here we provide the definitions for total-order HTN planning, dead-ends, and a recap of (inferred) preconditions and effects of compound tasks and methods.

HTN Planning Formalism

We consider totally ordered (t.o.) HTN planning and use a formalism based on the ones by Geier and Bercher (2011) and Behnke, Höller, and Biundo (2018). T.o. HTN planning domains are tuples $\mathcal{D} = (F, A, C, M)$, consisting of a finite set of facts F, *primitive tasks* A, *compound tasks* C (also called *abstract tasks*), and *decomposition methods* $M \subseteq C \times T^*$[1]. We denote the union of both types of tasks as $T = A \cup C$. Primitive tasks or actions $a = (prec, add, del) \in A$ are tuples, which describe their *preconditions* $prec(a) \subseteq F$ and *effects* $add(a), del(a) \subseteq F$ (the *add and delete effects, resp.*). We say that an action $a \in A$ is *applicable* in a state $s \in 2^F$ if $prec(a) \subseteq s$. In that case, applying it to s results in the successor state $\delta(s, a) = (s \setminus del(a)) \cup add(a)$. This can be generalized to sequences of actions $\bar{a} = \langle a_0 \dots a_n \rangle$ with $a_i \in A$, which are applicable in a state s_0 if a_0 is applicable in s_0 and for all $1 \leq i \leq n$, a_i is applicable in $s_i = \delta(s_{i-1}, a_{i-1})$. Besides primitive tasks, there are also compound tasks in HTN planning. Such a task $c \in C$ can be viewed as abstraction of primitive and/or compound tasks, which are further specified by methods $m = (c, \bar{t}) \in M$ that *decompose* a compound task c within a task network $tn_1 = \langle \bar{t}_1\, c\, \bar{t}_2 \rangle \in T^*$ into a task network $tn_2 = \langle \bar{t}_1\, \bar{t}\, \bar{t}_2 \rangle$ (denoted by $tn_1 \rightarrow_{c,m} tn_2$), where *task networks* are (possibly empty) finite sequences of tasks. We denote a (possi-

[1] In reference to Kleene star, T^* denotes the set containing the empty and all finite-length sequences of tasks of T.

ble empty) sequence of methods transforming tn into tn' by $tn \rightarrow tn'$ and call tn' a refinement of tn. A solution to a *t.o. HTN planning problem* $\Pi = (\mathcal{D}, s_I, tn_I, g)$ – containing the domain $\mathcal{D}$, an *initial state* $s_I \in 2^F$, an *initial task network* $tn_I \in T^*$, and a goal description $g \subseteq F$ – is a sequence of actions $tn = \langle a_0 \dots a_n \rangle \in A^*$ if and only if $tn_I \rightarrow tn$, tn is applicable in s_I, and results in a goal state $s \supseteq g$.

From Olz, Biundo, and Bercher (2021) we take also the following definitions. The set of *executability-enabling states* of a compound task $c \in C$ is $E(c) = \{s \in 2^F \mid \exists\, \bar{a} \in A^* : c \rightarrow \bar{a}$ and $\bar{a}$ is applicable in $s\}$ and the set of all states into which the execution of c in a state $s \in 2^F$ can result is given by $R_s(c) = \{s' \in 2^F \mid \exists\, \bar{a} \in A^* : c \rightarrow \bar{a}, \bar{a}$ is applicable in s and results in $s'\}$.

A *dead-end* is a search node from which there does not exist a path in the search tree to a solution. Formally, a tuple $(s, tn) \in 2^F \times T^*$ is a dead-end of a t.o. HTN planning problem $\Pi = (\mathcal{D}, s_I, tn_I, g)$ if and only if $(\mathcal{D}, s, tn, g)$ does not have a solution.

Preconditions and Effects of Compound Tasks

Our look-ahead technique is based on inferred preconditions and effects of compound tasks. Compound tasks (in the used formalization) do not have preconditions or effects, instead they are just placeholders for task networks that substitute them during planning.[2] By investigating into which action sequences compound tasks may be decomposed one can infer which state features must hold before the execution of any refinement, and likewise which state features are 'produced' by all of the refinements. The relevant definitions, as introduced in our previous work (Olz, Biundo, and Bercher 2021), are reproduced herein as necessary for this paper.

State-independent positive and negative effects (cf. Def. 4) of a compound task c are facts that are added or deleted, resp., by the successful execution of a refinement of c, independent of the state in which the task is executed, i.e.,

$$\mathit{eff}^+_*(c) := (\bigcap_{s \in E(c)} \bigcap_{s' \in R_s(c)} s') \setminus \bigcap_{s \in E(c)} s$$
$$\mathit{eff}^-_*(c) := \bigcap_{s \in E(c)} (F \setminus \bigcap_{s' \in R_s(c)} s')$$

if $E(c) \neq \emptyset$, otherwise $\mathit{eff}^{+/-}_*(c) := \mathit{undef}$.

Possible state-independent effects (cf. Def. 5) of a compound task c are not guaranteed to hold (or not hold, resp.,) after *every* refinement of c but after at least one:

$$\mathit{poss\text{-}eff}^+_*(c) := \bigcup_{s \in E(c)} (\bigcup_{s' \in R_s(c)} s' \setminus s)$$
$$\mathit{poss\text{-}eff}^-_*(c) := \bigcup_{s \in E(c)} ((\bigcup_{s' \in R_s(c)} (F \setminus s')) \cap s)$$

if $E(c) \neq \emptyset$ and $\mathit{poss\text{-}eff}^{+/-}_*(c) := \mathit{undef}$ otherwise.

[2] There are also hierarchical planning formalizations, which permit to specify preconditions and/or effects for compound tasks in the model (see, e.g., some overviews by Bercher et al. (2016) and Olz, Biundo, and Bercher (2021)), but we can say little about these formalizations because they do not have common semantics. Such preconditions and effects are also not reflected in current standard benchmark domains as described by HDDL (Höller et al. 2020a), which was used for the IPC 2020 on HTN planning.

Mandatory preconditions (cf. Def. 6) of c are facts that hold in every state for which there exists an executable refinement. So, they are required in every state in which a refinement of c shall be executed: $\mathit{prec}(c) := \bigcap_{s \in E(c)} s$ if $E(c) \neq \emptyset$ and $\mathit{prec}(c) := \mathit{undef}$ otherwise.

Since these definitions rely on *executable refinements* determining the preconditions and effects is computationally hard, ranging from **PSPACE**- to **EXPTIME**-complete – it is basically as hard as solving the respective planning problem in the first place (Olz, Biundo, and Bercher 2021). As we want to use these preconditions and effects for our look-ahead technique, using these *exact* preconditions and effects is not reasonable for the goal of speeding up search – it would be too costly and would not pay off. However, previously, we also introduced a variant of such preconditions and effects that can be computed in polynomial time based on what we called *precondition-relaxation*, which essentially analyzes preconditions and effects while completely ignoring executability (Olz, Biundo, and Bercher 2021). Formally, the *precondition-relaxation* of a domain $\mathcal{D} = (F, A, C, M)$ is the domain $\mathcal{D}' = (F, A', C, M)$ with $A' = \{(\emptyset, \mathit{add}, \mathit{del}) \mid (\mathit{prec}, \mathit{add}, \mathit{del}) \in A\}$. Then, $\mathit{eff}^{\emptyset+}_*(c)$, $\mathit{eff}^{\emptyset-}_*(c)$, $\mathit{poss\text{-}eff}^{\emptyset+}_*(c)$ and $\mathit{poss\text{-}eff}^{\emptyset-}_*(c)$ are *precondition-relaxed effects* (cf. Def. 9) and are defined just like the original ones but based on the precondition-relaxed variant of $\mathcal{D}$.

There is also a tractable variant of preconditions, which are called *executability-relaxed preconditions* (cf. Def. 10). A fact $f \in F$ is such a precondition of c, denoted $f \in \mathit{prec}^{\emptyset}(c)$, if and only if for all primitive refinements (i.e., ignoring executability) $\langle a_0 \dots a_n \rangle$ of c there exists an action a_i with $f \in \mathit{prec}(a_i)$ and there does not exist an action a_j with $j < i$ and $f \in \mathit{add}(a_j)$, where $i, j \in \{0 \dots n\}$. In other words, a fact is a precondition of a compound task if it is needed by at least one primitive task in every refinement of it and no other task adds it beforehand.

Procedures to infer precondition-relaxed effects and executability-relaxed preconditions in polynomial time were described in the proofs of Theorems 6 (on the poly-time decidability of possible effects) and Corollary 7 (guaranteed effects) as well as of Theorem 7 (on the poly-time decidability of preconditions) by Olz, Biundo, and Bercher (2021). We also mentioned that one can define inferred preconditions and effects of *decomposition methods* (rather than of compound tasks) as follows: Given a method $m = (c, tn)$ introduce a new compound task $c_{\widetilde{m}}$ together with a new method $\widetilde{m} = (c_{\widetilde{m}}, tn)$. In essence, $c_{\widetilde{m}}$ represents tn as it has only this single method. Then the preconditions and effects of m are equal to the ones of $c_{\widetilde{m}}$. We will exploit this later in our proposed technique. Very recently, we introduced conjunctive possible effects (Olz and Bercher 2023a), which are a generalization of the possible ones. They could potentially be used to further improve our technique in the future.

Look-Ahead Technique

We now describe our proposed look-ahead technique to detect inevitable decomposition choices and dead-ends and how it can be integrated into a search-based HTN system.

Algorithm 1: Look-ahead algorithm

Input: A planning problem $\Pi = (\mathcal{D}, s_I, tn_I, g)$, task network $tn = \langle t_1 \dots t_n \rangle \in T^*$, and a state s
Output: $(-1, -1)$, if (s, tn) is a dead-end, otherwise a (possibly empty) list of tuples $(k, m) \in [n] \times M$

function UNIQUEREFINE($(s, tn = \langle t_1 \dots t_n \rangle)$)
  $uniqueMethods = \emptyset$
  for $i = 1, \dots, n$ **do**
    if $t_i \in A$ **then**
      if $prec(t_i) \not\subseteq s$ **then return** $(-1, -1)$
      $s = (s \setminus del(t_i)) \cup add(t_i)$
    else
      $M' = \{m = (t_i, tn') \in M \mid prec^{\emptyset}(m) \subseteq s\}$
      if $M' = \emptyset$ **then return** $(-1, -1)$
      if $|M'| = 1, M' = \{m'\}$ **then**
        $uniqueMethods = uniqueMethods \cup \{(i, m')\}$
      $s = s \cup \bigcup_{m \in M'} poss\text{-}eff_*^{\emptyset+}(m)$
      $s = s \setminus \bigcap_{m \in M'} eff_*^{\emptyset-}(m)$
  if $g \not\subseteq s$ **then return** $(-1, -1)$
  return $uniqueMethods$

Look-ahead Procedure

We start by presenting pseudo code (Alg. 1), followed by an explanation, an example, and a proof of soundness.

We want to analyze search nodes, so our input is a task network $tn = \langle t_1 \dots t_n \rangle \in T^*$, and a state s. As a result, the procedure either returns that the search node is a dead-end or it returns a (possibly empty) list of tasks, for which always only one method can turn the current task network into a solution, together with the respective method. The idea is to consider the compound tasks like primitive ones with the inferred preconditions and effects of their methods and check whether the sequence is applicable in s and leads to a goal state. Therefore, we go through the tasks in the network from left to right, so we consider one t_i, $i \in \{1 \dots n\}$, after another and do the following:

- If the current task t_i is primitive we just check whether its preconditions are satisfied in s, as usual. If this is not the case, we've detected a dead-end and return $(-1, -1)$. Otherwise, we delete all its delete effects and add its add effects to s, just like applying it.
- If t_i is a compound task, we check which of its methods are applicable in s (denoted by the set M'), i.e. we check their inferred preconditions like for primitive tasks. If none of them are applicable, we have again found a dead-end, stop and return $(-1, -1)$. If one or multiple methods are applicable, we update s by adding all possible positive precondition-relaxed effects and deleting the intersection of all guaranteed negative ones of all methods of t_i that were not proven inapplicable. Additionally, if only one method is applicable, we save the combination of task and method in a list $uniqueMethods$.

At the end – after we have propagated all tasks of tn – we test whether the resulting state is a goal state. If it is not a goal state, we know that it is a dead-end and return $(-1, -1)$, otherwise, we return the list $uniqueMethods$ of compound tasks with inevitable decomposition choices.

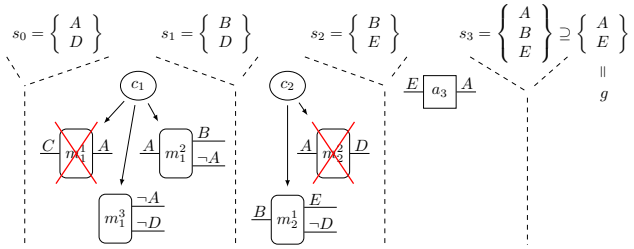

Figure 1: An example illustrating the look-ahead procedure. Tasks c_1 and c_2 are compound, a_3 is primitive. The depicted positive effects of all methods are possible positive precondition-relaxed effects, the negative effects are guaranteed ones.

Example 1 Let's consider the example in Figure 1, where the task network $tn = \langle c_1 c_2 a_3 \rangle$ and state s_0 are given and to be analyzed. We start with c_1. It has three methods m_1^1, m_1^2, and m_1^3 but only m_1^2 and m_1^3 are applicable as C, the precondition of m_1^1, does not hold in s_0. So we add B and delete A to generate $s_1 = \{B, D\}$. We do not delete D because it is not a negative effect of *both* methods, m_1^2 and m_1^3. When we go on to c_2 we see that the precondition of m_2^2 does not hold in s_1 but the ones of m_2^1 do. Thus now there is only a single applicable method left for c_2, so we add $(2, m_2^1)$ to $uniqueMethods$ and we apply m_2^2 according to its effects in s_1, which gives us $s_2 = \{B, E\}$. Now, a_3 is primitive, so we can handle it as usual in planning (and like unique methods): it is applicable in s_2 and turns it into $s_3 = \{A, B, E\}$, which is a goal state. As we have not encountered a dead-end, the algorithm would return $uniqueMethods = \{(2, m_2^1)\}$. We could, thus, decompose c_2 directly using method m_2^1. If in contrast any compound task would not have any method left, or if any primitive task could not be executed (cf. Example 2 later on), then the procedure would return $(-1, -1)$, i.e. the detection of a dead-end, and the search node would be discarded.

Theorem 1. *Algorithm 1 is sound.*

Proof. Given $\Pi = (\mathcal{D}, s_I, tn_I, g)$, task network $tn = \langle t_1 \dots t_n \rangle \in T^*$, and a state s, we have to show that when Algorithm 1 returns $(-1, -1)$ then (s, tn) is a dead-end, and when it returns a (possibly empty) list $uniqueMethods$ of tuples $(k, m) \in [n] \times M$, then all *other* methods (not listed in $uniqueMethods$) for those tasks listed in $uniqueMethods$ lead to a dead-end. Formally, let $(k, m) \in uniqueMethods$, then we must show that for all $m' = (t_k, tn') \in M \setminus \{m\}$ with $tn \to_{t_k, m'} tn''$, tn'' is a dead-end.

W.l.o.g. assume t_0 is compound. If we considered its mandatory preconditions (instead of the executability-relaxed preconditions) it follows immediately from their definitions that tn in combination with s can not be turned into a solution if s does not contain all these preconditions. Moreover, if we considered the state-independent effects of t_0 to generate the next state s', it follows analogously that

$s' \supseteq s''$ for all s'' that result from applying any executable refinement of t_0 in s as we add all possible positive effects and delete only the guaranteed negative ones. So the new state is a (non-necessarily strict) superset of the actual one covering all possible refinements of the compound task. Inductively this holds for all tasks in tn because if a task is not applicable in a superset of the actual state, then it can not be applicable in the actual one either.

Olz, Biundo, and Bercher (2021) showed that $prec^{\emptyset}(c) \subseteq prec(c)$, $poss\text{-}eff_*^{\emptyset+}(c) \supseteq poss\text{-}eff_*^{+}(c)$ and $eff_*^{\emptyset-}(c) \subseteq eff_*^{-}(c)$ $(= post_*^{-}(c))$ for all $c \in C$. Therefore, the above arguments still hold even if we consider the relaxed version of inferred preconditions and effects – which we do.

By relying on the inferred preconditions and effects of *methods* we are more precise since $prec^{\emptyset}(c) = \bigcap_{m=(c,tn)\in M} prec^{\emptyset}(m)$, $poss\text{-}eff_*^{\emptyset+}(c) = \bigcup_{m=(c,tn)\in M} poss\text{-}eff_*^{\emptyset+}(m)$ and $eff_*^{\emptyset-}(c) = \bigcap_{m=(c,tn)\in M} eff_*^{\emptyset-}(m)$ and we exclude some of them in the propagating process, i.e. when their preconditions do not hold in the current superstate. They can be omitted as they can not turn the current task network into a solution because of the same arguments as before.

□

Integration in Search Algorithm

We propose to use our look-ahead-technique in any HTN progression planner (Alford et al. 2012), such as SHOP2 and SHOP3 (Nau et al. 2003; Goldman and Kuter 2019), or in the progression subplanner of PANDA$_\pi$ (Höller et al. 2020b). In principle, however, it can also be used for other search-based techniques, like plan space search.

We integrated our technique (i.e., Alg. 1) into the progression-based PANDA planner. Note that Höller et al. (2020b) propose *three* different versions of the algorithm for partially ordered domains, with increasing level of *systematicity* (the level of redundancy in the search space (Kambhampati, Knoblock, and Yang 1995)). In the simplest version, multiple paths through the search space could lead to the same plan since decisions concerning ordering constraints and decompositions are just made in a different order but lead to the same result. Höller et al. propose two improved algorithms that reduce the search space (without losing solutions) by identifying and eliminating those redundant paths. However, as Höller et al. state, in total-order HTN planning all these versions collapse into one.

Pseudo code of our extended t.o. HTN progression search procedure is povided in Alg. 2. Lines 8-12 and 19-23 show our modifications to the original by Höller et al. (2020b).

The standard progression search algorithm that we extend works as follows: Search nodes consist of a state together with a task network, i.e., a sequence of tasks in t.o. HTN planning. They are organized in a fringe that is sorted according to some search strategy and potentially a heuristic. Search nodes are selected from that fringe and handled as follows. If the first task is primitive, we apply it if possible, else discard the node. Otherwise, i.e. if the first task is compound, we generate a new successor search node for each

Algorithm 2: Modified progression-based HTN planning

Input: A planning problem $\Pi = (\mathcal{D}, s_I, tn_I, g)$
Output: A search node with the solution to Π or *unsolvable*

```
1: fringe = {(s_I, tn_I)}
2: while fringe ≠ ∅ do
3:     n = (s, tn = ⟨t_0 ... t_k⟩) := fringe.pop()
4:     if n.isGoal then return n
5:     if isPrimitive(t_0) then
6:         if n.Applicable then
7:             n' := n.apply(t_0)
8:             if UNIQUEREFINE(s', tn') == (−1, −1) then
9:                 return unsolvable
10:            else
11:                for all (j, m') ∈ UNIQUEREFINE(s', tn') do
12:                    n' = n'.decompose(t_j, m')
13:            n'.calcHvalue()
14:            fringe.add(n')
15:        else continue
16:    else                                   ▷ t_0 is compound
17:        for m = (t_0, tn) ∈ M do
18:            n' = (s', tn') := n.decompose(t_0, m)
19:            if UNIQUEREFINE(s', tn') == (−1, −1) then
20:                continue
21:            else
22:                for all (j, m') ∈ UNIQUEREFINE(s', tn') do
23:                    n' = n'.decompose(t_j, m')
24:            n'.calcHvalue()
25:            fringe.add(n')
26: return unsolvable
```

of its methods by decomposing it accordingly. Before they are inserted into the fringe their heuristic value (if any) gets computed. All successors with a finite heuristic value are added to the fringe and the procedure starts over.

Now, our look-ahead technique can be integrated as follows. After a new node n' is generated – i.e., before one would normally compute its heuristic – we call UNIQUEREFINE to analyze it. If it is identified to be a dead-end (lines 8 and 19), then the node will be discarded, i.e., the heuristic will not get called (saving its computation time) and it will not get added to the fringe. Otherwise, when n' passes the dead-end check, we consider the returned list $uniqueMethods$ containing all identified unique decomposition choices and decompose for all these entries $(j, m') \in uniqueMethods$ the respective tasks t_j with the respective method m' (lines 12 and 23). Note that all these decompositions are done directly for the current search node, no matter how many these are or at which position within the sequence. Once all these inevitable choices have been applied directly, the heuristic gets computed and the resulting node is added to the fringe.

The benefits of our technique are manifold: (1) We can detect dead-ends that were not recognized by the heuristic as both procedures may operate on different problem relaxations as pointed out in the introduction. Since such dead-ends will not get expanded further the search space gets smaller. (2) The identification of unique refinement choices

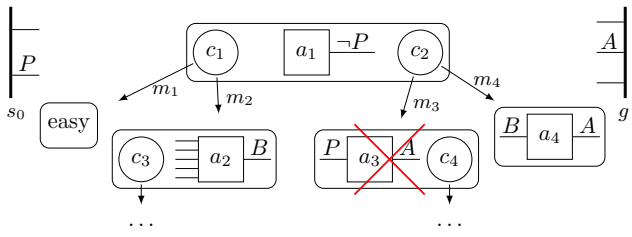

Figure 2: A task network, which shows that the early decomposition can complement a heuristic. Method m_3 is not applicable, so c_2 can only be decomposed by m_4. A heuristic estimate can get more exact given the information that the subtasks of m_3 are not available anymore.

can also reduce the search space as fruitless search nodes will never be generated and added to the fringe. (3) By applying unique decomposition choices in the back of the task network earlier than usual we can improve the heuristic's estimates as it must then evaluate a search node on a smaller (and thus more exact) set of available actions thus tightening informedness. (In an extreme case this could even make a heuristic capable to detect dead-ends.) Least importantly, but still worth mentioning, (4) applying a range of unique decompositions *at once* (cf. lines 11–12 and 22–23) reduces the number of heuristic calls (which may be expensive) and insertions into the search fringe, where each of the latter normally costs $O(log(n))$, n being the size of the fringe.

We showcase some of the advantages with two examples.

Example 2 Consider again Figure 1 and assume that a_3 has another precondition, say D. Then, our technique identifies a dead-end since it notices that a_3 can't be executed in s_2 – independently of which methods are chosen prior to its execution. All currently existing HTN planning heuristics, however, cannot possibly recognize this, this because they all rely on delete-relaxation – so they do not realize that D gets deleted by every executable refinement of c_2. Even if there were heuristics that do not perform delete-relaxation (which is not yet the case), such heuristics might ignore the ordering relations – which indeed is being done by *all* existing HTN planning heuristics. Then they can erroneously elude the dead-end situation by executing a_3 before c_2.

Example 3 As another example consider Figure 2, where a task network $\langle c_1 a_1 c_2 \rangle$ is given. Given as input to Alg. 1 it returns that c_2 can only be refined using m_4 because P is deleted by a_1, which is needed in m_3. The heuristic estimate for this search node becomes more accurate when c_2 is already decomposed into a_4 because of two points. (1) Task c_1 will have to produce B: Assume that the plans become very costly when m_2 is chosen to decompose c_1 as a_2 might have lots of preconditions that need to be fulfilled somehow by the refinement of c_3, whereas m_1 leads to a cheap plan but does not add B. Based on the original task network a delete-relaxing heuristic would therefore build a plan based on m_1 and m_3 as it does not see that P gets deleted, i.e., the heuristic value will be way too small. (2) The early decomposition does also prevent heuristics from using the subtasks of c_4 to find further oversimplified plans. So the heuristic value is more accurate – due to the combination of the look-ahead technique and early decomposition.

The progression algorithm we extended is known to be sound and complete (Höller et al. 2020b). Due to Theorem 1 we know that the integration of our look-ahead procedure only eliminates parts of the search space that provably do not lead to a solution. Performing certain decompositions at once rather than doing it sequentially as it would normally have been done, also neither influences soundness, nor could it lead to the elimination of possible solutions. Thus:

Proposition 1. *Algorithm 2 is sound and complete.*

Evaluation

We implemented our proposed technique in the latest progression-based version of the PANDA$_\pi$ system[3] (Höller and Behnke 2021; Höller et al. 2020b) and conducted an evaluation on the whole total-order IPC 2020 benchmark set[4]. While the code was not integrated into the public repository upon publication, a reference to the code is provided in our repository containing all produced data (Olz and Bercher 2023b). We selected the currently two best-performing configurations (Höller and Behnke 2021), i.e. GBFS in combination with the Relaxed Composition (RC) heuristic (Höller et al. 2020b) together with the classical heuristics Add (Bonet and Geffner 2001) and FF (Hoffmann and Nebel 2001). Our version is denoted PANDA$_\pi^{\text{Add}}$-reach and PANDA$_\pi^{\text{FF}}$-reach. The results were compared to the standard (progression-based) version of this planning algorithm with the same configurations PANDA$_\pi^{\text{Add}}$ and PANDA$_\pi^{\text{FF}}$, HyperTensioN (Magnaguagno, Meneguzzi, and de Silva 2021) and Lilotane (Schreiber 2021) (the winner and runner-up of the IPC 2020 total-order track), three planners that were published after the IPC 2020 TOAD (Höller 2021), PANDA$_\pi$-BDD (Behnke and Speck 2021) and HTN2SAS (Behnke et al. 2022), as well as PANDA$_\pi$-SAT (Behnke, Höller, and Biundo 2018; Behnke 2021), which did not compete at the IPC 2020 but solves more instances than the winner.

Experiments ran on Xeon E5-2660 v3 with 2.60GHz and 40 CPUs using 1 core, 8 GB of memory, and 30 minutes time limit per instance (time/memory limits of the IPC 2020).

Table 1 shows detailed results. Coverage reports on the number of solved instances within the time and memory limits. Normalized coverage ensures that all domains have equal weight thus making sure that domains with large numbers of instances do not contribute more than those with just a few. The IPC score is computed by $min\{1, 1 - log(t)/log(1800)\}$, where t is the time required to solve the problem in seconds. It rewards solving problems quickly.

Regarding previously existing planners, our experiments mainly confirm the findings by Behnke et al. (2022): The two variants of the progression-based HTN planner PANDA$_\pi^{\text{Add}}$ and PANDA$_\pi^{\text{FF}}$, respectively, and HTN2SAS[5] perform best, followed by PANDA$_\pi$-SAT, TOAD, and

[3] http://panda.hierarchical-task.net

[4] https://ipc2020.hierarchical-task.net

[5] We evaluated a newer version we received from the authors in

Domain		PANDA$_\pi^{\text{Add}}$-reach	PANDA$_\pi^{\text{FF}}$-reach	PANDA$_\pi^{\text{Add}}$	HTN2SAS	PANDA$_\pi^{\text{FF}}$	PANDA$_\pi$-SAT	TOAD	HyperTensioN	Lilotane
1)	30	**30**	27	**30**	**30**	25	6	**30**	3	5
2)	20	16	18	16	13	18	18	16	**20**	16
3)	30	**30**	29	**30**	27	**30**	23	22	16	21
4)	30	**30**	26	26	20	23	6	21	**30**	1
5)	30	23	21	23	25	21	23	23	**30**	29
6)	30	22	**27**	22	22	**27**	**27**	24	24	23
7)	147	**147**	**147**	**147**	**147**	**147**	**147**	141	**147**	**147**
8)	12	**12**	**12**	**12**	**12**	**12**	**12**	**12**	8	2
9)	20	**11**	8	8	6	7	8	5	3	4
10)	60	16	**25**	18	0	14	10	0	0	7
11)	30	25	25	25	23	**26**	22	22	25	21
12)	80	**80**	**80**	48	45	47	76	47	22	41
13)	20	4	4	4	3	4	3	1	**5**	1
14)	59	42	42	42	42	42	33	39	**57**	28
15)	20	**20**	**20**	**20**	**20**	**20**	**20**	0	**20**	**20**
16)	20	13	13	11	16	11	19	0	0	**20**
17)	74	**74**	40	**74**	72	27	19	**74**	8	4
18)	20	**20**	**20**	**20**	**20**	**20**	11	**20**	**20**	11
19)	30	27	26	29	16	26	22	8	**30**	21
20)	20	19	15	19	19	16	19	10	**20**	14
21)	20	**20**	**20**	**20**	**20**	**20**	19	14	**20**	17
22)	20	13	13	13	**14**	13	7	9	13	10
23)	40	22	23	25	28	19	38	36	**40**	34
24)	30	28	**30**	27	20	**30**	26	**30**	7	**30**
Cov.	892	**744**	711	709	660	645	614	604	568	527
Norm.		**19.50**	18.83	18.85	17.60	17.76	16.25	14.86	15.57	13.86
IPC Sc.		**15.32**	14.77	14.63	12.40	13.75	12.97	11.57	14.84	9.27

Table 1: A table showing coverage and IPC score. For the names of domains, please consult Table 3.

the winner and runner-up of the IPC (HyperTensioN and Lilotane, respectively).

The overall best-performing planners are the ones that incorporate the proposed technique: PANDA$_\pi^{\text{Add}}$-reach and PANDA$_\pi^{\text{FF}}$-reach. We can thus see that our technique increases the number of solved instances by 35 from 709 to 744 (the IPC score from 14.63 to 15.32) for the Add heuristic and by 66 from 645 to 711 (and the IPC score from 13.75 to 14.77) for the FF heuristic. An interesting observation is that even with the weaker FF heuristic our technique outperforms the better original planner (using the Add heuristic).

When looking at individual domains we see that in almost all domains coverage stays the same or increases. There were also a few domains in which the coverage dropped slightly. We speculate that the preprocessing technique that computes preconditions and effects does not pay off in these domains (too few dead-ends or too few detectable inferred preconditions/effects), though further investigations are required to identify when and why it pays off.

which a bug has been resolved. This results in a slightly weaker performance compared to their paper.

Domain	Dead-ends				Early dec.	
	total (%)		only (%)			
	reach	h(Add)	reach	h(Add)	nodes(%)	T/N
1)	27.15	16.20	18.64	7.70	1.11	1.00
2)	6.69	0.00	6.69	0.00	74.18	1.45
3)	20.92	3.65	17.26	0.00	29.74	5.17
4)	18.65	0.00	18.65	0.00	49.80	1.00
5)	0.00	0.00	0.00	0.00	1.76	1.02
6)	19.06	14.27	4.78	0.00	66.72	1.57
7)	2.39	0.00	2.39	0.00	12.83	16.47
8)	12.86	9.39	6.79	3.33	37.23	1.34
9)	9.03	2.36	6.67	0.00	64.03	1.21
10)	46.13	10.07	37.56	1.51	31.41	1.46
11)	1.61	10.56	0.33	9.28	4.00	1.33
12)	23.95	0.00	23.95	0.00	59.83	1.39
13)	18.09	0.08	18.00	0.00	32.97	1.01
14)	0.00	0.00	0.00	0.00	0.49	1.00
15)	41.21	30.04	23.36	12.19	27.82	1.91
16)	44.60	47.51	10.59	13.50	30.03	1.54
17)	58.99	3.44	55.55	0.00	22.17	1.35
18)	25.76	13.31	12.46	0.00	13.69	1.10
19)	2.76	0.00	2.76	0.00	21.93	1.21
20)	0.00	0.00	0.00	0.00	23.32	1.01
21)	41.07	10.80	30.91	0.65	35.48	1.50
22)	24.92	0.00	24.92	0.00	33.61	1.50
23)	0.00	0.00	0.00	0.00	23.42	1.42
24)	41.62	38.83	6.40	3.61	15.54	1.02

Table 2: PANDA$_\pi^{\text{Add}}$-reach: Number of dead-ends and early decompositions (% of evaluated search nodes) detected in total or only by our technique and the heuristic, resp. Last column shows how many tasks per node were decomposed. For the names of domains, please consult Table 3.

Detailed Analysis To give an insight into how often dead-ends or early decompositions were performed, we report the respective numbers for PANDA$_\pi^{\text{Add}}$-reach in Table 2.

The first two columns show how many of the search nodes are detected dead *in total* (in percentage of the overall checked search nodes[6]) by our look-ahead technique (reach) and the Add heuristic (h(Add)), respectively. In the third and fourth column we report how many search nodes are proven unsolvable only by our technique or only the heuristic. Normally, the heuristic will not get called on search nodes that are detected dead by our technique but for the purpose of this statistic, we evaluated them with both. So consider, e.g., Freecell: Without our technique 10.07% of the evaluated search nodes would have been detected dead by h(add) and thus pruned. With our technique (46.13+1.51)% of the evaluated search nodes were pruned. By looking at the data we can see that pruning increased at most from 3.44% to 58.99% for Multiarm-Blocksworld, which is an increase by a factor of 17. The second last column shows in how many search nodes early decomposition was per-

[6]Note that the used progression search often generates and processes search nodes without evaluating them by the heuristic (in case of unique decisions). Since they are also not analyzed by our technique we excluded them from this statistic.

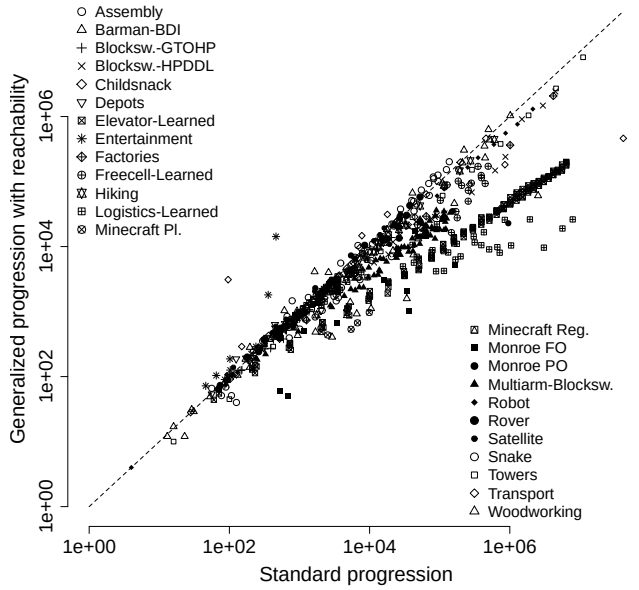

Figure 3: Scatterplot showing generated search nodes of the $\text{PANDA}_\pi^{\text{Add}}$ planner with standard progression against generalized progression with reachability analysis. Be aware of the log scale.

formed. Since sometimes several tasks in one node were decomposed directly, we report how many tasks per node were decomposed in the last column, i.e., a number 2 means that if we perform early decomposition in a search node then on average two tasks were decomposed. For all domains we report the mean of the mentioned metrics over all instances.

If we compare Tables 1 and 2 we see some correlations. For example, in Childsnack, Hiking, and Minecraft Reg. almost no dead-ends or early decompositions were detected and also coverage did not increase. On the other hand, domains that benefited from our technique like Blocksworld, Freecell, and Logistic also show a higher number of detected dead-ends and lookaheads. Since numbers are also high for Snake and Monroe FO it is likely that we could increase the coverage if further instances are generated since already all of their instances are solved by the standard PANDA_π.

However, looking at the number of identified dead-ends could be misleading since the concrete number does not say much because once a search node is pruned, no pruning in its subsearch space can occur. Instead, it is more meaningful to know how much search space is being saved. This is explicitly shown in Figure 3, which reveals that for most problems the search space becomes clearly smaller, and in many cases a considerable number of search nodes can be saved by using our proposed look-ahead technique (please be aware of the log scale). The search space can sometimes increase due to the planners' search strategy, which is greedy best-first search with a non-admissible heuristic.

Optimal Planning Lastly, we evaluated our technique for optimal planning. For that, we ran PANDA_π as A* search combined with the RC(LM-cut) (Helmert and Domshlak 2009) heuristic, which is currently the best admissible one

Domain		PANDA_π -reach	PANDA_π	PANDA_π -BDD
1) Assembly	30	4	4	**5**
2) Barman-BDI	20	**10**	**10**	7
3) Blocksw.-GTOHP	30	**26**	23	5
4) Blocksw.-HPDDL	30	**5**	**5**	3
5) Childsnack	30	0	0	**2**
6) Depots	30	**18**	**18**	15
7) Elevator-Learned	147	92	**112**	97
8) Entertainment	12	5	5	**7**
9) Factories	20	**6**	5	2
10) Freecell-Learned	60	0	0	0
11) Hiking	30	6	6	**7**
12) Logistics-Learned	80	**27**	**27**	22
13) Minecraft Pl.	20	**2**	1	1
14) Minecraft Reg.	59	33	33	**38**
15) Monroe FO	20	**19**	12	1
16) Monroe PO	20	**10**	7	0
17) Multiarm-Blocksw.	74	**12**	**12**	10
18) Robot	20	11	11	11
19) Rover	30	**8**	**8**	6
20) Satellite	20	**6**	**6**	0
21) Snake	20	**20**	**20**	19
22) Towers	20	**13**	12	9
23) Transport	40	10	9	**24**
24) Woodworking	30	**17**	16	14
Coverage	892	360	**362**	305
Normalized Coverage		**10.00**	9.33	7.30
IPC Score		**6.93**	6.51	5.16

Table 3: Coverage and IPC score for optimal planning.

implemented for PANDA_π as inner heuristic for RC. The results are given in Table 3. We can see that overall the planner benefits from our technique since normalized coverage and the IPC score are higher than before. In total, we solve two instances less, which is due to a single domain, Elevator. An analysis revealed that while PANDA_π-reach explores always fewer search nodes than the standard planner in this domain, the time for considering one search node is higher. The overhead in time is only implicitly caused by our technique: the heuristic became slower. In the Elevator domain many early refinements in the middle or back of the search nodes are performed (see also Table 2) such that the task networks get very large. The heuristic then needs more time to evaluate them. The estimates become more accurate because of this as predicted, however, in the end, this does not pay off here.

Conclusion

We have proposed and implemented a look-ahead technique based on inferred preconditions and effects of compound tasks that can identify dead-ends and unique refinement choices in search nodes for t.o. HTN planning. It can be included in different kinds of search-based HTN planners and complements existing heuristics. Empirical results show that exploited in a progression-based planning algorithm it outperforms the currently strongest HTN planners in the default measures of performance coverage and IPC score.

Acknowledgments

We would like to thank Mario Schmautz and Gregor Behnke for their support and assistance for the evaluation.

References

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight Bounds for HTN Planning. In *ICAPS*, 7–15. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. 2012. HTN Problem Spaces: Structure, Algorithms, Termination. In *SoCS*, 2–9. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. 2014. On the Feasibility of Planning Graph Style Heuristics for HTN Planning. In *ICAPS*, 2–10. AAAI Press.

Behnke, G. 2021. Block Compression and Invariant Pruning for SAT-based Totally-Ordered HTN Planning. In *ICAPS*, 25–35. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – Totally-Ordered Hierarchical Planning through SAT. In *AAAI*, 6110–6118. AAAI Press.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On Succinct Groundings of HTN Planning Problems. In *AAAI*, 9775–9784. AAAI Press.

Behnke, G.; Pollitt, F.; Höller, D.; Bercher, P.; and Alford, R. 2022. Making Translations to Classical Planning Competitive With Other HTN Planners. In *AAAI*, 9687–9697. AAAI Press.

Behnke, G.; and Speck, D. 2021. Symbolic Search for Optimal Total-Order HTN Planning. In *AAAI*, 13, 11744–11754. AAAI Press.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations. In *IJCAI*, 6267–6275. IJCAI.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In *IJCAI*, 480–488. 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 *ECAI*, 225–233. IOS Press.

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *AIJ*, 129(1-2): 5–33.

Clement, B. J.; Durfee, E. H.; and Barrett, A. C. 2007. Abstract Reasoning for Planning and Coordination. *JAIR*, 28: 453–515.

de Silva, L.; Meneguzzi, F. R.; and Logan, B. 2020. BDI Agent Architectures: A Survey. In *IJCAI*, 4914–4921. IJCAI.

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

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *IJCAI*, 1955–1961. AAAI Press.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: the case of SHOP3. In *ELS*, 73–80. ACM.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *ICAPS*, 162–169. AAAI Press.

Hoffmann, J.; and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14: 253–302.

Höller, D. 2021. Translating Totally Ordered HTN Planning Problems to Classical Planning Problems Using Regular Approximation of Context-Free Languages. In *ICAPS*, 159–167. AAAI Press.

Höller, D.; and Behnke, G. 2021. Loop Detection in the PANDA Planning System. In *ICAPS*, 168–173. AAAI Press.

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 *AAAI*, 9883–9891. AAAI Press.

Höller, D.; and Bercher, P. 2021. Landmark Generation in HTN Planning. In *AAAI*, 11826–11834. AAAI Press.

Höller, D.; Bercher, P.; and Behnke, G. 2020. Delete- and Ordering-Relaxation Heuristics for HTN Planning. In *IJCAI*, 4076–4083. IJCAI.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020b. HTN Planning as Heuristic Progression Search. *JAIR*, 67: 835–880.

Kambhampati, S.; Knoblock, C. A.; and Yang, Q. 1995. Planning as Refinement Search: A Unified Framework for Evaluating Design Tradeoffs in Partial-Order Planning. *AIJ*, 76(1-2): 167–238.

Magnaguagno, M. C.; Meneguzzi, F.; and de Silva, L. 2021. HyperTensioN: A three-stage compiler for planning. In *10th International Planning Competition: Planner and Domain Abstracts (IPC 2020)*, 5–8.

Magnaguagno, M. C.; Meneguzzi, F.; and de Silva, L. 2022. HyperTensioN and Total-order Forward Decomposition optimizations. arXiv preprint. arXiv:2207.00345.

Marthi, B.; Russell, S. J.; and Wolfe, J. A. 2007. Angelic Semantics for High-Level Actions. In *ICAPS*, 232–239. AAAI Press.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *JAIR*, 20: 379–404.

Olz, C.; and Bercher, P. 2023a. Can They Come Together? A Computational Complexity Analysis of Conjunctive Possible Effects of Compound HTN Planning Tasks. In *ICAPS*. AAAI Press.

Olz, C.; and Bercher, P. 2023b. Experimental Results for the SoCS 2023 Paper "A Look-Ahead Technique for Search-Based HTN Planning: Reducing the Branching Factor by Identifying Inevitable Task Refinements". doi: 10.5281/zenodo.7900414.

Olz, C.; Biundo, S.; and Bercher, P. 2021. Revealing Hidden Preconditions and Effects of Compound HTN Planning Tasks – A Complexity Analysis. In *AAAI*, 11903–11912. AAAI Press.

Schreiber, D. 2021. Lilotane: A Lifted SAT-Based Approach to Hierarchical Planning. *JAIR*, 70: 1117–1181.

Tsuneto, R.; Hendler, J.; and Nau, D. 1998. Analyzing External Conditions to Improve the Efficiency of HTN Planning. In *AAAI*, 913–920. AAAI Press.

Yang, Q. 1990. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6(1): 12–24.