Compiling HTN Plan Verification Problems into HTN Planning Problems

Daniel Höller,[1] Julia Wichlacz,[1] Pascal Bercher,[2] Gregor Behnke[3]
[1]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany,
[2]The Australian National University, Canberra, Australia,
[3]University of Freiburg, Freiburg, Germany,
hoeller@cs.uni-saarland.de, wichlacz@cs.uni-saarland.de, pascal.bercher@anu.edu.au, behnkeg@cs.uni-freiburg.de

Abstract

Plan Verification is the task of deciding whether a sequence of actions is a solution for a given planning problem. In HTN planning, the task is computationally expensive and may be up to NP-hard. However, there are situations where it needs to be solved, e.g. when a solution is post-processed, in systems using approximation, or just to validate whether a planning system works correctly (e.g. for debugging or in a competition). In the literature, there are verification systems based on translations to propositional logic and based on techniques from parsing. Here we present a third approach and translate HTN plan verification problems into HTN planning problems. These can be solved using any HTN planning system. We test our solver on the set of solutions from the 2020 International Planning Competition. Our evaluation is yet preliminary, because it does not include all systems from the literature, but it already shows that our approach performs well compared with the included systems.

1 Introduction

Plan Verification is the task of deciding whether a sequence of actions is a solution for a given planning problem. It is necessary in several situations, e.g. when a plan is post-processed, or to verify whether a planning system works correctly (e.g. for debugging or in a competition).

In classical planning, it can be solved in (lower) polynomial time. In Hierarchical Task Network (HTN) planning (Bercher, Alford, and Höller 2019), the complexity depends on several parameters:

- On whether the decomposition steps (i.e., the chosen methods) leading to a solution are known.
- On the specific problem class (e.g., whether it's partially or totally ordered).

When we look at the formalisms in HTN planning, the decomposition steps are usually not regarded part of a solution. Since only the contained primitive tasks (i.e., actions) need to be executed, there is often no need to do so. However, there are use cases in the literature where they are needed, mainly for communication with a user on different levels of abstraction (Bercher et al. 2021; Behnke et al. 2020a; Köhn et al. 2020; de Silva, Padgham, and Sardina 2019). When they are present (and if full information about task labeling is available so that multiple occurrences of the same task can still be distinguished) plan verification can be solved in polynomial time. Another polytime case is given by Totally Ordered (TO) HTN problems, where all methods and the initial task network are totally ordered. In this case, plan verification is cubic and resembles the problem of parsing in context-free languages. Otherwise, for general partially ordered (PO) HTN problems, it becomes NP-hard (Behnke, Höller, and Biundo 2015).

Both cases, i.e., TO and PO HTN planning problems, were considered in the 2020 International Planning Competition (IPC). Here, the participating systems needed to return the decomposition steps to allow the organizers to verify solutions in polynomial time. However, though it is possible for the solvers to track this information, it causes technical problems – consider e.g. the various compilation steps often performed in preprocessing[1] that need to be undone in post-processing. In other cases it is even not possible, e.g. when postoptimizing solutions or when internally using approximations like e.g. the TOAD system (Höller 2021), which overapproximates the solution set of a problem and needs verification as a regular step of its planning procedure to make sure only to return correct solutions.

In the literature, there are systems to solve the problem via translation to propositional logic (Behnke, Höller, and Biundo 2017) and based on parsing techniques (Barták, Maillard, and Cardoso 2018; Barták et al. 2020).

In this paper, we present an approach to compile HTN verification problems to common HTN planning problems. Our compilation is based on an approach for plan recognition as planning (Höller et al. 2018). It is applicable in both TO and PO HTN planning. We combine our transformation with two planning systems from the PANDA framework (Höller et al. 2021) and evaluate it on a new benchmark set that includes the models from the 2020 IPC and solutions generated by the IPC participants. It yields good results both in PO and in TO HTN planning.

The paper is structured as follows: we first introduce the formal framework used in the paper (Section 2), then we introduce the compilation (Section 3), describe the realization (Section 4) and evaluate the approach (Section 5). We conclude the paper with a short discussion (Section 6).

[1]See e.g. the PANDA grounder (Behnke et al. 2020b) used in the following. Among other compilations, it changes decomposition rules until convergence, which is not a simple task to undo.

2 Formal Framework

In HTN planning there are two types of tasks, *primitive* and *abstract* tasks. Primitive tasks are equivalent to actions in classical planning, i.e., they are directly applicable in the environment and cause state transition. Abstract (or compound) tasks are not directly applicable and need to be decomposed into other tasks in a process similar to the derivation of words from a formal grammar. A solution to an HTN planning problem needs to be derived via this grammar.

Formally, an HTN planning problem is a tuple $p = (F, C, A, M, s_0, tn_I, g, prec, add, del)$. F is a set of propositional state features. A state s is defined by the subset of state features that hold in it, $s \in 2^F$, all other state features are assumed to be *false*. $s_0 \in 2^F$ is the initial state of the problem, and $g \subseteq F$ is the state-based goal description[2]. A state s is called a *goal state* if and only if $g \subseteq s$. A is a set of symbols called *primitive tasks* (also called *actions*). These symbols are mapped to a subset of the state features by the functions $prec$, add, del, which are all defined as $f : A \to 2^F$ and define the actions' preconditions, add-, and delete-effects, respectively. An action a is applicable in a state s if and only if $prec(a) \subseteq s$. When an applicable action a is applied in a state s, the state $s' = \gamma(s, a)$ resulting from the application is defined as $s' = (s \setminus del(a)) \cup add(a)$. A sequence of actions $a_1, a_2, \ldots, a_n$ is applicable in some state s_0 if and only if a_i is applicable in the state s_{i-1}, where s_i for $1 \leq i \leq n$ is defined as $s_i = \gamma(s_{i-1}, a_i)$. We call the state s_n the state *resulting* from the application.

Tasks in HTN planning are maintained in *task networks*. A task network is a partially ordered multiset of tasks. Formally, it is a triple $tn = (T, \prec, \alpha)$. T is a set of identifiers (ids) that are mapped to the actual tasks by the function $\alpha : T \to N$, where $N = A \cup C$ is the union of the primitive tasks A and the abstract (compound) tasks C. $\prec$ is a partial order on the task ids. tn_I is the initial task network, i.e., the task network the decomposition process starts with. Legal decompositions are defined by the set of *(decomposition) methods* M. A method is a pair (c, tn), where $c \in C$ defines the task that can be decomposed using the method, and the task network tn defines into which tasks it is decomposed. When a task t from a task network tn is decomposed using a method (c, tn'), it is replaced by the tasks in tn'. When t has been ordered with respect to other tasks in tn, the new tasks inherit these ordering constraints. Formally, a method $m = (c, tn)$ decomposes a task network $tn_1 = (T_1, \prec_1, \alpha_1)$ that contains a task id $t \in T_1$ with $\alpha_1(t) = c$ into a task network tn_2, which is defined as follows. Let $tn' = (T', \prec', \alpha')$ be a copy of tn that uses ids not contained in T_1. Then tn_2 is defined as:

$$tn_2 = ((T_1 \setminus \{t\}) \cup T', \prec' \cup \prec_1 \cup \prec_D, (\alpha_1 \setminus \{t \mapsto c\}) \cup \alpha')$$

$$\prec_D = \{(t_1, t_2) \mid (t_1, t) \in \prec_1, t_2 \in T'\} \cup \{(t_1, t_2) \mid (t, t_2) \in \prec_1, t_1 \in T'\} \cup \{(t_1, t_2) \mid (t_1, t_2) \in \prec_1, t_1 \neq t \wedge t_2 \neq t\}$$

[2]In HTN planning, there is usually no state-based goal given, because it can be compiled away. However, it makes our definition in the next sections more natural.

When a task network tn can be decomposed into a task network tn' by applying (a finite sequence of) 0 or more methods, we write $tn \to^* tn'$.

A task network $tn_S = (T_S, \prec_S, \alpha_S)$ is a solution to a given HTN planning problem if and only if the following conditions hold:

1. $tn_I \to^* tn_S$, i.e., it can be derived from the initial task network,
2. $\forall t \in T_S : \alpha_S(t) \in A$, i.e., all tasks are primitive, and
3. there is a sequence $(i_1 i_2 \ldots i_n)$ of the task ids in T_S in line with the ordering constraints $\prec_S$ such that $(\alpha_S(i_1)\alpha_S(i_2) \ldots \alpha_S(i_n))$ is applicable in s_0 and results in a goal state.

We call an HTN method *totally ordered* when the tasks in the contained task network are totally ordered. We call an HTN planning problem totally ordered when all contained methods and the initial task network are totally ordered.

Definition 1 (Plan Verification)**.** *Given an HTN planning problem p and a sequence of actions $(a_1 a_2 \ldots a_n)$, plan verification is the problem to decide whether there is a task network $(T_S, \prec_S, \alpha_S)$ that is a solution for p and for an ordering $(i_1 i_2 \ldots i_n)$ of the task identifiers T_S fulfilling solution criterion 3 as given above, it holds that $(\alpha_S(i_1)\alpha_S(i_2) \ldots \alpha_S(i_n)) = (a_1 a_2 \ldots a_n)$.*

3 Compiling Verification Problems to Planning Problems

Let $p = (F, C, A, M, s_0, tn_I, g, prec, add, del)$ be an HTN planning problem, $\pi = (a_1 a_2 \ldots a_n)$ a sequence of actions out of A, and $v = (p, \pi)$ a plan verification problem. We compile v into a new HTN planning problem $p' = (F', C', A', M', s'_0, tn_I, g', prec', add', del')$ that has a solution if and only if v is solvable.

The encoding is widely identical with the one introduced by Höller et al. (2018) for plan and goal recognition (PGR) as planning. It has also been shown that it can be used for plan repair (Höller et al. 2020b). We first change the state and the actions of the original problem such that the only applicable sequence of actions exactly resembles π. Let $f_0, f_1, f_2, \ldots, f_n$ be new state features. We use them to encode which actions out of π have already been executed. In addition to that, we need a state feature $\bot$, which will never be reachable. The set of state features of p' is defined as $F' = F \cup \{f_0, f_1, f_2, \ldots, f_n\} \cup \{\bot\}$. In the beginning, no action out of π has been executed, i.e., $s'_0 = s_0 \cup \{f_0\}$. We want solutions to exactly equal π, i.e., all actions need to be included. This is enforced by including f_n in the state-based goal definition g, i.e., $g' = g \cup \{f_n\}$.

For $a_i \in \pi$ with $1 \leq i \leq n$, we introduce a new action a'_i. The preconditions of the new actions enforce the correct position in the generated solution

$$prec'(a'_i) = prec(a_i) \cup \{f_{i-1}\},$$

each action deletes its own precondition and adds the one of the next action in the solution

$$add'(a'_i) = add(a_i) \cup \{f_i\},$$
$$del'(a'_i) = del(a_i) \cup \{f_{i-1}\}.$$

Please be aware that an action out of A may appear more than once in the solution. In such cases, there will be multiple copies of the action in A'. The novel actions mimic the state transition of the original ones, but additionally ensure their respective position in the solution. All other actions shall never appear in any solution, so we add the new state feature $\bot$ that is never reachable to their preconditions.

$$\begin{aligned} \forall a \in A : prec'(a) &= prec(a) \cup \{\bot\}, \\ add'(a) &= add(a), \\ del'(a) &= del(a) \end{aligned}$$

The new set of actions is defined as $A' = A \cup \{a'_i \mid a_i \in \pi\}$.

Due to the new state features, preconditions and effects, there is only one sequence of actions that is applicable and leads to a state-based goal. However, none of the new actions can ever be reached by decomposing the initial task network. To make this possible, we need to modify the decomposition hierarchy. It shall be possible for a newly introduced action a' to be placed at exactly those positions where the action a might have been in the original model. We thereby need to keep in mind that there might be multiple copies of some action a, so we cannot just replace them in the methods. We need to introduce a new choice point to choose which copy $a', a'', \dots$ of a shall be at which position in the action sequence. We do this by introducing one novel abstract task c_a for each action a. Let $C^A = \{c_a \mid a \in A\}$. We further introduce new methods to decompose this new task into one of the copies of a.

$$M^A = \{(c_a, (\{i\}, \emptyset, \{i \mapsto a'\})) \mid a' \in \pi\}$$

We define $C' = C \cup C^A$ and $M' = M \cup M^A$ and have fully specified our compiled problem p'.

The resulting encoding is nearly identical with the one used in the fully observable case of plan and goal recognition as planning (Höller et al. 2018). The only difference is the additional precondition of the actions not included in the solution. While the PGR encoding forces these actions to be placed after a given plan prefix of observed actions, the encoding here makes them entirely unreachable.

3.1 Some Technicalities

The models in our benchmark are those from the 2020 IPC, which are modeled in the description language HDDL (Höller et al. 2020a). In HDDL, models may include state-based preconditions for methods. These are preconditions as known from actions, which have to hold for the method to be applicable. The semantics of such preconditions is a bit problematic (see Höller et al. (2020a, p. 5) for a discussion). In HDDL it is defined as follows: a new action is introduced that is inserted in the method and placed before every other task of the method's subtasks. This new action holds the precondition of the method. We will call such actions *technical actions*.

In TO HTN planning, this fully specifies the position where the precondition needs to hold. However, consider the case of PO HTN planning. Here, the task decomposed by the method might be partially ordered with respect to other tasks and the subtasks might be interweaved. As a result, we cannot exactly determine the position the precondition needs to hold. When the state-features contained in the method's precondition are not static (i.e., might change over time), the position where the precondition is checked might change applicability.

Since technical actions are not actually part of the solution, planners will not return them. From a verification perspective, this is problematic, a verifier needs to check whether there *exists* a position where a technical action is applicable. This is a main problem with the SAT-based approach from related work (Behnke, Höller, and Biundo 2017), which needs to get the technical actions as input.

In our approach, handling this issue is not a problem: we leave the preconditions and effects of technical actions unchanged, i.e., they will not be affected by the encoding. As a result, they can be inserted into plans of the compiled problems at arbitrary positions in line with the definition of HTN decomposition. That is, if the plan π to verify has a length of n, then our encoding makes sure that – provided plan π is indeed a valid solution to the original problem – a solution of size at least n will be found (corresponding to π), but additional actions which encode the method preconditions may be included as well at appropriate positions (one for each applied method with a precondition). Note that this means that there is no clear limit on the length of solutions (other than its minimum). Since methods might replace an abstract task by no other task, i.e., delete it, it is not clear how many such empty methods might have been applied in the worst case leading to a plan of a certain length.

3.2 Properties

Let $v = (p, \pi)$ be a Plan Verification Problem and p' the encoding as given above.

Our encoding serves the purpose of deciding whether π is a solution for p. This is being achieved provided that π is a solution for p if and only if p' is solvable, which we capture in the following two theorems. Note that this result (restricted to methods without preconditions) follows as a special case from Thm. 1 by Höller et al. (2018). That theorem from the context of plan recognition states that the encoding – the same one we deploy for plan verification – ensures that the solutions of the encoded problem are *exactly* those of the original problem that start with the enforced actions. In the context of this earlier work, there might have been additional actions after the prefix of enforced actions, namely the remaining plan that should be recognized. In our case, this part remains empty (modulo technical actions encoding method preconditions).

Theorem 1. *When π is a solution for p, then the compiled problem p' is solvable.*

Proof Sketch. Since π is a solution to p, we know that there is a sequence of method applications that transforms the initial task network tn_I into a primitive task network tn, which in turn allows π as executable linearization. Note that we can assume that the solution was achieved by a progression planner, which applies methods and actions in a forward-fashion, since such a progression-based solution exists if and only if

any solution exists at all (Alford et al. 2012, Thm. 3). Thus, we can assume that there is a sequence of method and action applications $\overline{ma}$ that transforms tn_I into π. That sequence can be transformed into a corresponding (and potentially longer) sequence in p'. For each action a_i at position i in π its corresponding encoding a'_i will be executable in the solution π' to p', though the respective sequence of method and action applications will be preceded by the method decomposing c_a thus introducing that encoding of a'_i. Furthermore, every method m in $\overline{ma}$ will be applicable in the corresponding method and action sequence $\overline{ma}'$ leading to π' in p' as well, though immediately preceded by the technical action encoding the precondition of m. □

Theorem 2. *When π is no solution for p, then the compiled problem p' is unsolvable.*

Proof Sketch. This direction is a bit easier to see than the previous one, since the model of p' is an extension of the original one, i.e., it follows the exact same structure, but each action has additional preconditions and thus makes the problem more constrained. So if there is no solution in the original model, there cannot be a solution in the encoded one. □

It was also shown that the transformation maintains most structural properties of the problem (Höller et al. 2020b, Sec. 6.1), i.e., tail-recursive, acyclic, and totally ordered problems remain tail-recursive, acyclic, or totally ordered, respectively. Since we deploy the same encoding we essentially get the same property, though the restriction to a specific solution might lead to even more restrictive cases. E.g., the restriction to the model required to obtain the plan π to verify might turn a problem without any restriction even into a totally ordered acyclic problem. We still can directly conclude the following properties:

Corollary 1. *If p is tail-recursive, p' is tail-recursive. If p is acyclic, p' is acyclic. If p is totally ordered, p' is totally ordered.*

4 Pruning the Model

The encoding makes wide parts of the original actions inapplicable. When realizing it on the lifted definition, this would be detected by the grounding procedure (see e.g. (Ramoul et al. 2017; Behnke et al. 2020b)), which would not generate unreachable parts.

However, we realized our encoding on a fully grounded model. For us, this had two main advantages: First, it simplifies the implementation. Second, since our planning system grounds the model before planning, we have the ground model and do not need to ground twice. However, in order to result in a small model, we need to prune unreachable parts. We use the following two pruning methods, which are similar to what the PANDA grounder described by Behnke et al. (2020b) would do in its grounding process.

4.1 Bottom-up Reachability

By construction, all actions not included in the enforced solution contain the unreachable precondition $\bot$, i.e., the only actions that are (potentially) reachable are those in the enforced solution as well as the technical actions. Let A_T be the set of technical actions. We initialize our reachability analysis with these actions: $N_r = \{a'_i \mid a'_i \in \pi\} \cup A_T$.

We now want to determine the set of reachable methods and abstract tasks. Since we know that certain actions are not reachable, we know that any method which includes such actions will never be part of any solution. Or, the other way around, we know that only methods not including such actions will be part of a solution. For an abstract task c, we know that it can only be part of a solution when there is at least one method that decomposes c that might be part of a solution. Based on these observations, we calculate the sets of methods and abstract tasks that might be part of a solution.

Let TN^N be the set of all task networks over the tasks N. We define the relation $R : TN^N \times 2^N$ with

$$R = \{((T, \prec, \alpha), N') \mid \forall t \in T : \alpha(t) \in N'\}$$

Let $N_r \subseteq N$ be a set of reachable tasks. As discussed above, the set of reachable methods based on this set is defined as

$$M_r = \{(c, tn) \in M \mid (tn, N_r) \in R\}$$

The overall reachability is then defined as follows:

function bottom-up(N_r)
$M_r = \emptyset$
while N_r *changes* **do**
  $M_r = \{(c, tn) \in M \mid (tn, N_r) \in R\}$
  $N_r = N_r \cup \{c \mid (c, tn) \in M_r\}$
return (N_r, M_r)

As given above, this algorithm is started with $N_r = \{a'_i \mid a'_i \in \pi\} \cup A_T$, i.e., the set containing the actions in the enforced solution and all technical actions.

4.2 Top-down Reachability

The sets returned by the analysis given above might include tasks and methods not reachable from the initial task network. We therefore perform a second (top-down) analysis. Let $tn_I = (T_I, \prec_I, \alpha_I)$ be the initial task network and N_r and M_r the sets returned by the bottom-up analysis. We determine the tasks and methods reachable top-down using the following function.

function top-down(T_I, α_I, N_r, M_r)
$M'_r = \emptyset$
$N'_r = \{n \mid \exists i \in T_I : \alpha_I(i) = n\}$
while N'_r *changes* **do**
  $N'_r = N'_r \cup \{n \mid \exists c \in N'_r, (c, (T, \prec, \alpha)) \in M_r), i \in T : \alpha(i) = n\}$
  $M'_r = \{(c, tn) \in M_r \mid c \in N'_r\}$
return (N'_r, M'_r)

We perform a single pass of these two methods and output the resulting (reduced) problem afterwards.

5 Evaluation

We next describe the benchmark set and the systems included in the evaluation. Then we discuss the results.

The experiments ran on Xeon Gold 6242 CPUs using one core, a memory limit of 8 GB, and a time limit of 10 minutes.

5.1 Benchmark Set

We use a new set of benchmark problems that is based on the models from the 2020 IPC. It contains 892 planning problems from 24 domains in TO planning and 224 instances from 9 domains in PO planning. The solutions have been created by 7 different planning systems for TO and by 4 systems for PO; namely by the final versions of the participants of the IPC as well as by planners from the PANDA framework (Höller et al. 2021). In total, this results in 10963 plan verification instances in TO and 1077 in PO HTN planning.

Since plans and domains stem from a recent competition, we consider it an interesting benchmark set with respect to the included plans and the difficulty of the instances. However, there is one weakness that we want to address in future work: Since the current set includes only instances from the *final* planner versions (after the debugging process), it includes only very few instances that are incorrect plans, only 2 instances for TO and 1 for PO.

In related work, this problem was solved by using random walks (Behnke, Höller, and Biundo 2017). However, it is hard to create such instances with appropriate difficulty, and we do not consider the resulting instances as interesting as the positive ones given above. In future work, we want to include instances from early planner versions from the IPC (before debugging) to obtain a more realistic benchmark set. However, this work is still in progress and here we present the results from the benchmark set as described above.

5.2 Systems

We ground the models using the PANDA grounder (Behnke et al. 2020b). Grounding time is included in the runtimes. After transformation and pruning as given in Section 3 and 4, we output the same format as the PANDA grounder.

We use two planner configurations from the PANDA framework (Höller et al. 2021) to solve the resulting HTN planning problems:

- The progression search with the Relaxed Composition (RC) heuristic (Höller et al. 2018, 2020c) and loop detection (Höller and Behnke 2021).
- The SAT-based solver for TO (Behnke, Höller, and Biundo 2018; Behnke 2021) and for PO (Behnke, Höller, and Biundo 2019) HTN planning.

We compare our system against two systems from the literature, a SAT-based and a parsing-based approach.

SAT-based verification. The first system is based on a compilation to propositional logic (Behnke, Höller, and Biundo 2017). It supports both totally ordered and partially ordered problems. However, it relies on an input plan that contains all actions – including the technical actions given above. This makes a comparison difficult. We addressed the issue by removing all method preconditions from the input model provided to this particular system. As a result, no technical actions are introduced and the approach can be applied. Since we remove constraints from the model, all solutions to the original problem are also solutions to the new model. However, the new model allows for more solutions. Thus, the results obtained by this workflow might be incorrect. However, we argue that this does not make the solving process harder and that we can fairly compare our runtimes against this approach.

Parsing-based verification. The second approach from the literature is based on parsing (Barták, Maillard, and Cardoso 2018; Barták et al. 2020). In principle, it supports both TO and PO models. However, while the TO version works fine on our benchmark set, we where not able obtain a stable run with the PO version and thus do not include the results here. We know that this makes our evaluation preliminary and will address the issue in future work.

5.3 Results

In the TO setting, our compilation approach reaches a coverage of 99.4% with the progression search and 89.2% with the SAT-based PANDA. The SAT-based verifier has a coverage of 8.3%, the parsing-based system one of 23.5%. When comparing our two configurations, the SAT-based planner solves 7 instances that the progression search does not solve. For our compilation combined with the progression search planner, only 67 plans of the corpus cannot be verified. In 43 of these cases, we already fail to ground the planning problem (within the given memory/time limits). We currently do grounding without considering the plan to be verified. As such, the grounded model usually contains a significant number of actions, tasks, and methods which cannot be part of the compiled model. From the instances where the grounding was successful, our compilation combined with the progression search solves 99.8% of the instances.

Figure 1 visualizes the runtimes for the TO setting. The curve on the left is the SAT-based approach from related work, which has the worst performance, followed by the parsing-based approach. Our translation combined with the progression search performs best, followed by our translation combined with the SAT-based planner.

For the PO setting, our compilation reaches a coverage of 98.9% with the SAT-based planner and a coverage of 90.3% with the progression search. The SAT-based verifier has a coverage of 72.0%. We assume that the higher coverage is caused by the fact that the plans in the PO setting are significantly shorter. For the PO instances, grounding never fails.

Table 1 shows some characteristic numbers in the TO setting for the different domains. Our approach combined with the progression search has a coverage of 100% in 18 of 24 domains. The parsing-based system has a higher coverage in the Childsnack domain, where we have a coverage of 98.1% and the parsing-based system has 99.6%. Though the parsing-based system also works on a grounded model, it does not use an external grounder and can incorporate reachability information into the grounding process. This seems

Domain	#Plans	Verified				Shortest unverified plan Comp_{pro}	Plan Length			Runtime for Verified			Pearson Correlation
		SAT	Parsing	Comp SAT	Comp pro		Min–Max	Avg	Median	Min–Max	Avg	Median	
AssemblyHierarchical	193	24	102	**193**	**193**	–	4 – 256	31.1	14	0.07 – 0.76	0.2	0.11	0.812
Barman-BDI	423	79	33	396	**423**	–	10 – 1198	128.4	69	0.07 – 6.57	0.3	0.14	0.890
Blocksworld-GTOHP	160	2	5	142	**160**	–	21 – 6661	479.8	209.5	0.07 – 534.39	10.3	0.15	0.906
Blocksworld-HPDDL	172	0	5	143	170	4853	20 – 5732	461.1	163	0.07 – 542.21	15.8	0.19	0.914
Childsnack	529	92	527	516	519	750	50 – 2500	119.8	80	0.12 – 56.20	1.2	0.28	0.864
Depots	455	60	210	436	**455**	–	15 – 971	129.1	92	0.07 – 4.32	0.3	0.13	0.930
Elevator-Learned	2812	2	213	2700	**2812**	–	10 – 2165	225.1	200	0.06 – 6.71	0.3	0.21	0.940
Entertainment	159	111	**159**	**159**	**159**	–	24 – 128	71.7	64	0.08 – 4.33	1.6	0.58	0.199
Factories-simple	123	9	9	96	**123**	–	15 – 2968	623.7	251	0.07 – 17.38	1.8	0.14	0.928
Freecell-Learned	204	0	26	152	**204**	–	57 – 489	162.7	138.5	2.86 – 13.06	4.9	5.2	0.882
Hiking	565	0	156	**565**	**565**	–	26 – 174	70.8	72	0.17 – 45.97	2.4	0.97	0.641
Logistics-Learned	1108	0	9	683	**1108**	–	27 – 2813	413.1	370	0.07 – 14.55	0.6	0.37	0.919
Minecraft-Player	75	0	0	73	74	278	35 – 278	51.9	44	10.64 – 120.90	73.5	93.61	0.923
Minecraft-Regular	766	0	0	616	734	107	35 – 9947	253.8	135	0.12 – 207.75	11.6	1.455	0.326
Monroe-FO	248	0	176	**248**	**248**	–	3 – 96	41.5	39	3.48 – 3.94	3.7	3.66	0.334
Monroe-PO	217	0	63	**217**	**217**	–	6 – 91	45.1	45	3.43 – 3.91	3.7	3.67	0.390
Multiarm-Blocksworld	443	9	22	419	**443**	–	20 – 543	182.1	124	0.07 – 6.25	0.8	0.20	0.903
Robot	117	21	27	85	**117**	–	2 – 1725	272.4	37	0.06 – 59.25	4.2	0.08	0.914
Rover-GTOHP	509	22	172	397	**509**	–	16 – 2640	320.7	212	0.06 – 86.33	5.3	1.49	0.827
Satellite-GTOHP	296	9	84	199	**296**	–	12 – 1584	379.1	270	0.06 – 58.23	6.9	2.67	0.846
Snake	183	153	77	182	**183**	–	2 – 162	20.6	16	0.09 – 8.99	1.1	0.57	0.230
Towers	17	3	5	8	12	8191	1 – 131071	15419.1	511	0.07 – 141.21	14.6	0.195	0.684
Transport	695	65	239	664	678	382	8 – 5077	188.9	76	0.06 – 406.08	2.4	0.1	0.719
Woodworking	494	251	261	**494**	**494**	–	3 – 219	57.5	25	0.08 – 22.49	5.0	1.01	0.994
	10963	912	2580	9783	10896	107	1 – 131071	239.2	119	0.06 – 542.21	3.2	0.29	0.273

Table 1: Characteristic numbers for the TO setting. From left to right: Name of the domain, followed by the number of verification instances and coverage for all systems per domain. Length of the shortest plan that has not been solved by the progression search, statistics regarding plan length and runtime. The last column gives the correlation between plan length and runtime.

Domain	#Plans	Verified			Shortest unverified plan Comp_{SAT}	Plan Length			Runtime for Verified			Pearson Correlation
		SAT	Comp pro	Comp SAT		Min–Max	Avg	Median	Min–Max	Avg	Median	
Barman-BDI	56	37	46	44	90	10 – 1198	108.4	32	0.01 – 495.17	48.1	6.65	0.621
Monroe-Fully-Observable	129	7	**129**	**129**	–	9 – 61	24.9	24	6.85 – 105.65	29.3	14.82	0.656
Monroe-Partially-Observable	104	9	103	**104**	–	9 – 47	23.3	24	3.84 – 88.76	21.9	14.375	0.234
PCP	26	6	**26**	**26**	–	10 – 90	28.0	26	0.01 – 99.15	4.6	0.19	0.772
Rover	144	131	138	**144**	–	8 – 115	31.2	25	0.01 – 113.22	4.3	0.57	0.579
Satellite	246	**246**	**246**	**246**	–	5 – 28	13.5	13	0.00 – 0.11	0.0	0.02	0.677
Transport	183	150	96	**183**	–	8 – 69	27.2	28	0.01 – 139.83	2.1	0.23	0.321
UM-Translog	52	**52**	**52**	**52**	–	7 – 37	16.8	13	0.01 – 0.05	0.0	0.02	0.800
Woodworking	137	**137**	**137**	**137**	–	3 – 20	11.9	12	0.00 – 0.36	0.2	0.14	0.863
	1077	775	973	1065	90	3 – 1198	25.7	18	0.00 – 495.17	8.8	0.19	0.667

Table 2: Characteristic numbers for the PO setting. From left to right: Name of the domain, followed by the number of verification instances and coverage for all systems per domain. Length of the shortest plan that has not been solved by the progression search, statistics regarding plan length and runtime. The last column gives the correlation between plan length and runtime.

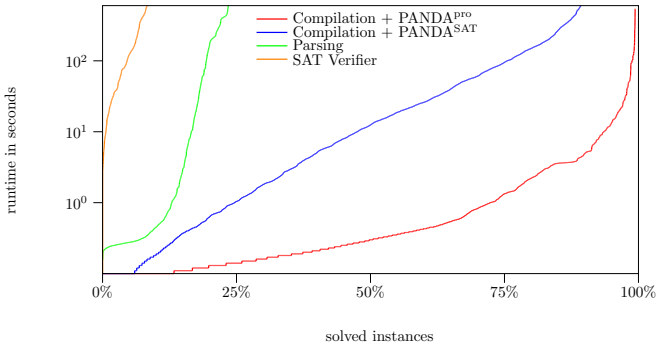

Figure 1: Solved instances in percent (on the y axis) relative to the runtime (on the x axis) for the TO setting.

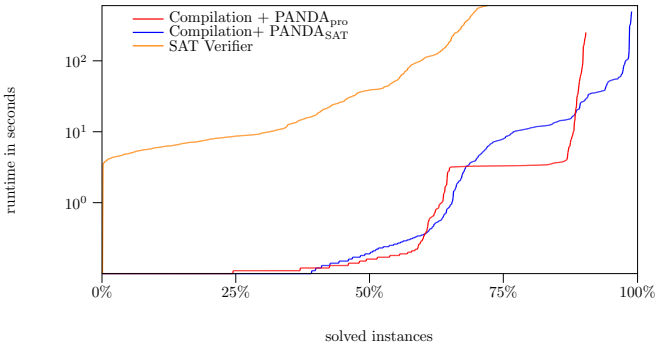

Figure 2: Solved instances in percent (on the y axis) relative to the runtime (on the x axis) for the PO setting.

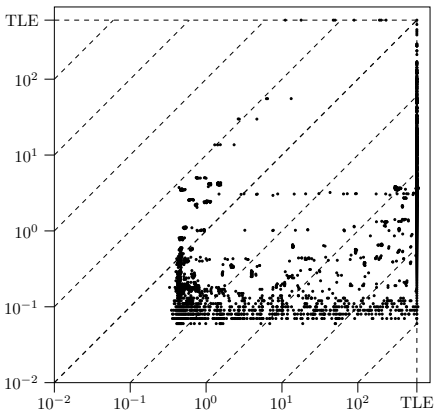

Figure 3: Runtime of our system (y axis) on the TO set compared to the parsing-based approach on the x axis (log scale).

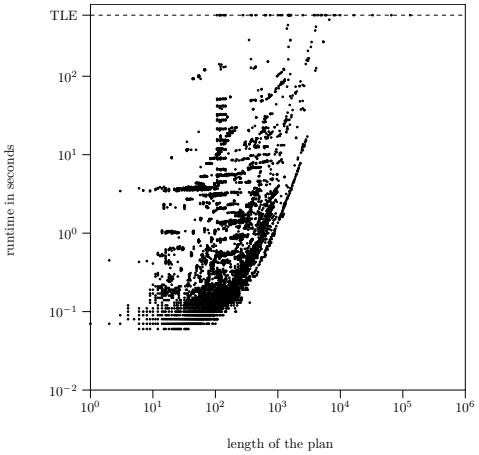

Figure 4: Runtime against length of the verified solution (be aware of the log scale).

to be an advantage in the Childsnack domain. In all other domains, our system has the highest coverage. For those domains where *not* all instances have been solved, we included the length of the shortest plan that could not be verified.

The next table (Table 1) gives statistics on plan length. Regarding the medians, AssemblyHierarchical is the domain with the shortest plans (14), and Towers the one with the longest (511). The median over all domains is 119.

Next, the table gives information about the runtime needed by our approach (combined with the progression search). The longest median runtime is needed for Minecraft-Player. In 16 domains, the median is one second or less. The last column gives the correlation between the plan length and the runtime needed for verification.

Table 2 shows the same characteristics for the PO setting. Notably the plans are much shorter (the average length is smaller by nearly one order of magnitude). Our compilation together with the SAT-based planner can verify all plans for 8 of the 9 domains with only Barman-BDI to have some plans that could not be verified. Notably, these plans are longer than almost all plans.

Figure 3 shows the runtime of our approach with the progression search in the TO setting compared to the parsing-based system. It can be seen that in most instances solved by both systems, our system is faster than the one from the literature. If it is not, the difference is about one second or less. Figure 4 shows a comparison of runtime and plan length. It can be seen that the runtime of equally long plans can be very different, but also that longer plans are harder to verify.

Figure 2 visualizes the runtime in the PO setting. It includes the compilation in combination with SAT-based PANDA and progression search and the SAT-based verifier. Like in the TO setting, our approach outperforms the SAT-based verifier. The plateau in the curves of our approaches are caused by the Monroe domain, where (especially using progression search) nearly the entire time is needed for grounding and not for solving the ground problem.

6 Discussion & Conclusion

We have presented an approach to compile HTN plan verification problems to HTN planning problems and have shown that recent planning systems can solve the resulting problems for plans with reasonable length (i.e., for plan lengths resulting from the recent IPC benchmarks/planners).

A possible criticism of a compilation-based approach might be that one has to rely on the correctness of the applied HTN planning system. So the question is why we rely more on these systems than on the planning system that has generated the plan in the first place. Since HTN planning systems are complex systems, we agree that these systems might also be incorrect (though this might also be the case for verifiers, of course). However the HTN planning systems used in our evaluation return the decomposition steps performed to find a plan as specified for the IPC. Therefore they provide a witness for the validity of their result (at least for cases where they find a solution) that can be checked with the much simpler systems based on these steps. So we can e.g. use the verifier developed for the IPC to check our results.

For cases where the HTN planning system does not find a solution, we cannot provide a meaningful explanation why planning failed. However, please note that what we would like to have here is a certificate of unsolvability of a planning problem, which is at least in classical planning an active field of research (see e.g. (Eriksson, Röger, and Helmert 2017; Eriksson and Helmert 2020)), though we are not aware of similar work in HTN planning.

As most important steps in future work we consider the collection of unsolvable instances from early runs of the IPC planners and the comparison to the parsing-based approach in the setting of PO planning.

The performance of the progression-based system on the PO benchmark set points to other lines of research. One is to help the planners by propagating the implications of the total order of the plan though the partial ordered HTN model. Since this is another compilation, it would preserve the property of needing no specialized solver. A second promising direction is to actually adapt the planner and, e.g., come up with specialized heuristics that take the additional information about the problem into account. Natural candidates would be the landmark heuristic by Höller and

Bercher (2021), which might benefit from the ordering constraints implied by the solutions, or the IP-based heuristic introduced by Höller, Bercher, and Behnke (2020), where it is straightforward to integrate the additional knowledge. However, such systems would, of course, lack the elegance of using standard planning systems.

Acknowledgments

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

References

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2012. HTN Problem Spaces: Structure, Algorithms, Termination. In *Proc. of the 5th Annual Symposium on Combinatorial Search (SoCS)*, 2–9. AAAI Press.

Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of Hierarchical Plans via Parsing of Attribute Grammars. In *Proc. of the 28th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 11–19. AAAI Press.

Barták, R.; Ondrcková, S.; Maillard, A.; Behnke, G.; and Bercher, P. 2020. A Novel Parsing-based Approach for Verification of Hierarchical Plans. In *Proc. of the 32nd IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, 118–125. IEEE Press.

Behnke, G. 2021. Block Compression and Invariant Pruning for SAT-based Totally-Ordered HTN Planning. In *Proc. of the 31st Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 25–35. AAAI Press.

Behnke, G.; Bercher, P.; Kraus, M.; Schiller, M.; Mickeleit, K.; Häge, T.; Dorna, M.; Dambier, M.; Minker, W.; Glimm, B.; and Biundo, S. 2020a. New Developments for Robert – Assisting Novice Users Even Better in DIY Projects. In *Proc. of the 30th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 343–347. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the Complexity of HTN Plan Verification and Its Implications for Plan Recognition. In *Proc. of the 25th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 25–33. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2017. This is a solution! (. . . but is it though?) – Verifying solutions of hierarchical planning problems. In *Proc. of the 27th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 20–28. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – Totally-Ordered Hierarchical Planning through SAT. In *Proc. of the 32nd AAAI Conf. on Artificial Intelligence (AAAI)*, 6110–6118. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019. Bringing Order to Chaos – A Compact Representation of Partial Order in SAT-based HTN Planning. In *Proc. of the 33rd AAAI Conf. on Artificial Intelligence (AAAI)*, 7520–7529. AAAI Press.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020b. On Succinct Groundings of HTN Planning Problems. In *Proc. of the 34th AAAI Conf. on Artificial Intelligence (AAAI)*, 9775–9784. AAAI Press.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations. In *Proc. of the 28th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 6267–6275. IJCAI organization.

Bercher, P.; Behnke, G.; Kraus, M.; Schiller, M.; Manstetten, D.; Dambier, M.; Dorna, M.; Minker, W.; Glimm, B.; and Biundo, S. 2021. Do It Yourself, but Not Alone: *Companion*-Technology for Home Improvement – Bringing a Planning-Based Interactive DIY Assistant to Life. *Künstliche Intelligenz – Special Issue on NLP and Semantics* .

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.

Eriksson, S.; and Helmert, M. 2020. Certified Unsolvability for SAT Planning with Property Directed Reachability. In *Proc. of the 30th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 90–100. AAAI Press.

Eriksson, S.; Röger, G.; and Helmert, M. 2017. Unsolvability Certificates for Classical Planning. In *Proc. of the 27th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 88–97. AAAI Press.

Höller, D. 2021. Translating Totally Ordered HTN Planning Problems to Classical Planning Problems Using Regular Approximation of Context-Free Languages. In *Proc. of the 31st Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 159–167. AAAI Press.

Höller, D.; and Behnke, G. 2021. Loop Detection in the PANDA Planning System. In *Proc. of the 31st Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 168–173. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2018. Plan and Goal Recognition as HTN Planning. In *Proc. of the 30th IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, 466–473. IEEE Computer Society.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2021. The PANDA Framework for Hierarchical Planning. *Künstliche Intelligenz* doi:10.1007/s13218-020-00699-y.

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 *Proc. of the 34th AAAI Conf. on Artificial Intelligence (AAAI)*, 9883–9891. AAAI Press.

Höller, D.; and Bercher, P. 2021. Landmark Generation in HTN Planning. In *Proc. of the 35th AAAI Conf. on Artificial Intelligence (AAAI)*, 11826–11834. AAAI Press.

Höller, D.; Bercher, P.; and Behnke, G. 2020. Delete- and Ordering-Relaxation Heuristics for HTN Planning. In *Proc. of the 29th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 4076–4083. IJCAI organization.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A Generic Method to Guide HTN Progression Search with Classical Heuristics. In *Proc. of the 28th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 114–122. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020b. HTN Plan Repair via Model Transformation. In *Proc. of the 43rd German Conference on AI (KI)*, 88–101. Springer.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020c. HTN Planning as Heuristic Progression Search. *Journal of Artificial Intelligence Research* 67: 835–880.

Köhn, A.; Wichlacz, J.; Torralba, Á.; Höller, D.; Hoffmann, J.; and Koller, A. 2020. Generating Instructions at Different Levels of Abstraction. In *Proc. of the 28th Int. Conf. on Computational Linguistics (COLING)*, 2802–2813. International Committee on Computational Linguistics.

Ramoul, A.; Pellier, D.; Fiorino, H.; and Pesty, S. 2017. Grounding of HTN Planning Domain. *International Journal on Artificial Intelligence Tools* 26(5): 1760021:1–1760021:24.