Compiling HTN Plan Verification Problems into HTN Planning Problems

Daniel Höller,[1] Julia Wichlacz,[1] Pascal Bercher,[2] Gregor Behnke[3,4]

[1]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany,
[2]The Australian National University, Canberra, Australia,
[3]University of Freiburg, Freiburg, Germany,
[4]University of Amsterdam, ILLC, The Netherlands
hoeller@cs.uni-saarland.de, wichlacz@cs.uni-saarland.de, pascal.bercher@anu.edu.au, g.behnke@uva.nl

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). There are verification systems based on translations to propositional logic and 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 collected a new benchmark set based on models and results of the 2020 International Planning Competition. Our evaluation shows that our compilation outperforms the approaches from the literature.

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 (see Bercher, Alford, and Höller (2019) for an overview), the complexity depends on several parameters, e.g. on whether the decomposition steps (i.e. the chosen methods) leading to a solution are known or on the specific problem class (e.g. whether the tasks are partially or totally ordered). In 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.

However, there are use cases in the literature where they are needed, e.g. for communication with a user (Bercher et al. 2021; Behnke et al. 2020; Köhn et al. 2020; de Silva, Padgham, and Sardina 2019). When they are present, (HTN) plan verification can be solved in polynomial time. Another polytime case is given in Totally Ordered (TO) HTN problems, where all methods and the initial task network are totally ordered. Here, the decomposition rules resemble rules of a context-free grammar, making plan verification equivalent to parsing such a grammar (which can be done in cubic time). For general partially ordered (PO) HTN problems, it becomes NP-hard (Behnke, Höller, and Biundo 2015).

Both TO and PO HTN planning were tracks in the 2020 International Planning Competition (IPC). The participants needed to return the decomposition steps to allow the organizers a verification in polynomial time. However, though it is possible to track this information, it causes technical problems – consider e.g. the various compilation steps often performed in preprocessing that need to be undone. In other cases it is even not possible to output the decomposition steps, e.g. when postoptimizing solutions or when using approximations like e.g. the TOAD system (see 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 standard HTN planning problems. Our compilation is based on previous work on plan recognition as planning (Höller et al. 2018). It is applicable in both TO and PO HTN planning. We solve the resulting problems with planning systems that return the decomposition steps, which provide a witness of correctness for the verified solution. As second contribution, we collected a novel benchmark set based models and results of the 2020 IPC and solutions generated by the IPC participants. To also include instances where the provided plans are *not valid*, we collected instances from non-final submissions of the IPC planning systems that lead to incorrect plans. On this benchmark set, our approach outperforms both the SAT-based and the parsing-based approach from related work.

Formal Framework

In HTN planning there are two types of tasks, *primitive* and *compound* tasks. Primitive tasks are equivalent to actions in classical planning, i.e., they are directly applicable and cause state transitions. 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 needs to be derived via this grammar.

We use the HTN formalization by Höller et al. (2016), which is based on the one by Geier and Bercher (2011). A 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, which can be compiled away in HTN planning and is therefore often omitted. However, it makes our definitions more natural. A state s is a *goal state* if and only if $g \subseteq s$. A is a set of symbols called *primitive tasks* (also *actions*). These are mapped to a subset of the state features by the functions $prec$, add, del, all defined as $f : A \to 2^F$. They define the actions' preconditions, add-, and delete-effects. 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 $a_1 a_2 \dots a_n$ is applicable in a 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 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:

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

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 an HTN problem if and only if

1. $tn_I \to^* tn_S$, i.e., it can be derived from the initial task network via decomposition,
2. $\forall t \in T_S : \alpha_S(t) \in A$, i.e., all tasks are primitive, and
3. there is a sequence $(i_1 i_2 \dots i_n)$ of the task ids in T_S in line with the ordering constraints $\prec_S$ such that $(\alpha_S(i_1)\alpha_S(i_2) \dots \alpha_S(i_n))$ 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 $(a_1 a_2 \dots a_n)$, 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 $(i_1 i_2 \dots i_n)$ of the task identifiers T_S fulfilling solution criterion 3 as given above, it holds that $(\alpha_S(i_1)\alpha_S(i_2) \dots \alpha_S(i_n)) = (a_1 a_2 \dots a_n)$.*

Compilation to Planning

The presented 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).

Let $p = (F, C, A, M, s_0, tn_I, g, prec, add, del)$ be an HTN planning problem, $\pi = (a_1 a_2 \dots a_n)$ 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.

We first change the state and the actions of the original problem such that the only applicable sequence of actions exactly resembles π. Let $f_0, f_1, \dots, f_n$ 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 cannot be made true in any state. The set of state features of p' is defined as $F' = F \cup \{f_0, f_1, \dots, f_n\} \cup \{\bot\}$. 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 goal definition, 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

$$
\begin{aligned}
add'(a_i') &= add(a_i) \cup \{f_i\}, \\
del'(a_i') &= del(a_i) \cup \{f_{i-1}\}.
\end{aligned}
$$

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 state feature $\bot$ that cannot be made true 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 goal state. 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 compound task c_a for each action a. Let $C^A = \{c_a \mid a \in A\}$. We first replace actions in the original methods by these new tasks.

$$M^O = \{m = (T, \prec, \alpha') \mid m \in M\} \text{ where}$$
$$\forall t \in T \text{ with } \alpha(t) = n \begin{cases} \alpha'(t) = c_n & \text{for } n \in A \\ \alpha'(t) = c & \text{else} \end{cases}$$

Then we introduce new methods to decompose the new tasks into 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 $M' = M^O \cup M^A$ and have fully specified our compiled problem p'.

The resulting encoding is nearly identical with the one used in the fully observable case of PGR 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.

Next we discuss theoretical properties of the compilation. 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 is a special case of Thm. 1 by Höller et al. (2018), which states that the encoding 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 PGR, there might be additional actions after the prefix of enforced actions, namely the remaining plan that should be recognized. In our case, this part remains empty due to the state feature $\bot$ added to the 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 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. □

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 leading to the particular plan, there cannot be a solution in the encoded one. □

It was also shown that the compilation maintains most structural properties of the original 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.*

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

A Novel Benchmark Set for Plan Verification

We collected a new benchmark set that is based on the models from the 2020 IPC. These are 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 participants of the IPC as well as by planners from the PANDA framework (Höller et al. 2021). 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.

To include instances of *invalid* solutions, we collected invalid plans from early (non-final) versions of the IPC participants (before the debugging process). Naturally, these are by far less than the *valid* ones. However, since these are also examples from real systems, we deem them much more interesting than artificially generated instances as e.g. used by Behnke, Höller, and Biundo (2017). The number of instances are given in the upper four rows of Table 1, column "Inst.". We come to the other rows in the next paragraphs.

		Inst.	Compilation		Parsing	SAT
			Progression	SAT		
TO	Valid	10961	**10881** (**99.27**)	9757 (89.02)	9158 (83.55)	not supported
	Invalid	1406	**1364** (**97.01**)	727 (51.71)	1301 (92.53)	not supported
PO	Valid	1211	1088 (89.84)	**1198** (**98.93**)	not supported	not supported
	Invalid	138	**129 (93.48)**	64 (46.38)	not supported	not supported
TO No M.P.	Valid	11304	**9679** (**85.62**)	8986 (79.49)	7889 (69.79)	1036 (9.16)
	Invalid	1063	898 (84.48)	406 (38.19)	**915** (**86.08**)	684 (64.35)
PO No M.P.	Valid	1243	1103 (88.74)	**1212** (**97.51**)	973 (78.28)	897 (72.16)
	Invalid	106	98 (92.45)	57 (53.77)	**106** (**100.00**)	103 (97.17)

Table 1: Coverage table. The first column gives the HTN sub-class (partially ordered or totally ordered), followed by the verification setting (valid or invalid plans), followed by the results. Highest coverage per setting is given bold. The first four rows give the results on the original data set. The evaluation of models without method preconditions are given thereafter.

The models from the 2020 IPC are modeled in the description language HDDL (Höller et al. 2020a). In HDDL, models may include state-based preconditions for *methods*. These are similar to preconditions of actions, but specify when *a method* is applicable. The semantics of such preconditions is a bit problematic (see Höller et al. (2020a) for a discussion). In HDDL it is defined via a compilation: a new action holding the precondition is inserted in the method and placed before all other subtasks. 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 decomposed task might be partially ordered with respect to other tasks and the subtasks might be interweaved. Thus we cannot exactly determine the position the precondition is checked/needs to hold. This definition was chosen for HDDL due to practical reasons, since it is the most simple way for the *planning systems*. However, this does not hold for verification, as discussed in the next paragraph.

Since the new actions are not actually part of the solution, planners will not return them. For verifiers, this means that they need to check whether there *exists* a position where the precondition holds (in a certain range of the plan). Two verifiers from related work do not support this, which makes a comparison difficult. To include these systems in our evaluation, we created a second benchmark set by removing all method preconditions[1]. This relaxes the constraints induced by the preconditions. As a result, some instances that have been "invalid" before are now "valid", which shows that these preconditions have to be checked in a real verification.

We ran the evaluation on both benchmark sets, including only systems supporting method preconditions in the first evaluation, and all systems in the second one. This results in the last four lines in Tab. 1.

Evaluation

The experiments ran on Xeon Gold 6242 CPUs using one core, a memory limit of 8 GB, and a time limit of 10 minutes[1]. Next we discuss the systems used in the evaluation, before we come to the results.

[1]The source code of the translation and both benchmark sets are available at *panda.hierarchical-task.net*

Systems

Compilation-based Verification We combined our compilation with two planners from the PANDA framework (Höller et al. 2021) to solve the resulting problems: the progression search with the Relaxed Composition (RC) heuristic (Höller et al. 2018, 2020c) and graph search (Höller and Behnke 2021) and the SAT-based solvers for TO (Behnke, Höller, and Biundo 2018; Behnke 2021) and for PO (Behnke, Höller, and Biundo 2019) planning.

For our compilation, there are two ways to handle method preconditions:

1. When performing the compilation on the lifted model, method preconditions can simply be ignored during compilation and written to the output problem. One just has to ensure the planner used afterwards supports them.
2. When grounding first, the new actions introduced by the grounder have to be ignored during compilation and written (unchanged) to the output problem.

In both cases, the additional actions are integrated in the plan when solving the compiled problem, enforcing the method preconditions. An approach similar to (2) is not easily possible for the following systems from related work, since the additional actions would then need to be contained in the plan to be verified that is provided to these systems.

SAT-based Verification The first system from related work we compare against is based on a compilation to propositional logic (Behnke, Höller, and Biundo 2017). It supports both TO and PO models. However, it does not support method preconditions.

Parsing-based Verification The second approach is based on techniques from parsing. We included one version tailored to TO planning (Barták et al. 2021), which shows better results on these models but is not applicable to PO models, and a general one used on the PO models (Barták, Maillard, and Cardoso 2018; Barták et al. 2020).

For the PO setting, the system does also not support a verification of method preconditions as specified by HDDL.

Results

On the *valid* instances in the TO setting, our compilation approach reaches a coverage of 99.27% with the progression search and 89.02% with the SAT-based PANDA, and

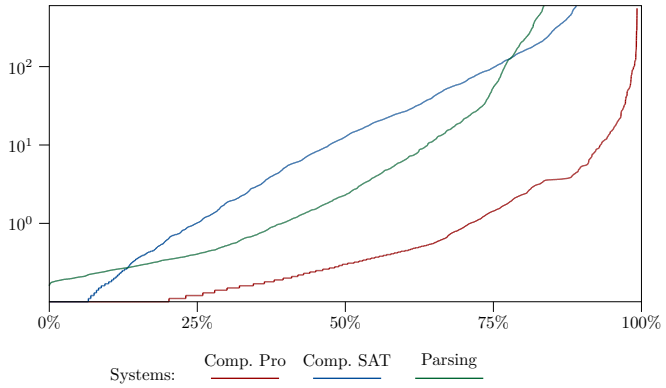

Figure 1: Solved "valid" instances relative to runtime in seconds (TO setting, with method preconditions).

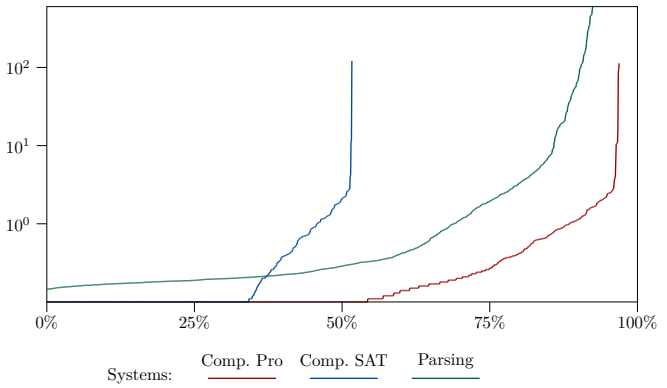

Figure 2: Solved "invalid" instances relative to runtime in seconds (TO setting, with method preconditions).

97.01% and 51.71% for the *invalid* instances. The parsing-based approach reaches 83.55% and 92.53%. For the PO setting, the compilation solves 89.84% and 98.93% of the valid instances (progression/SAT) and 93.48% and 46.38% of the invalid instances. No other system supports (the IPC semantic of) method preconditions.

The SAT-based planner is less successful in showing unsolvability of a planning problem than the progression-based system. This is because common benchmarks do not contain unsolvable instances, so that it is not optimized towards showing unsolvability. Currently, it can only show unsolvability easily if the planning problem is acyclic. If it is cyclic, it needs to exhaust a bound that is exponential in the number of state features (Behnke, Höller, and Biundo 2018). Interestingly, the progression-based system, which is not tailored towards totally ordered problems has a higher coverage than the parsing-based system, which is a specialized TO system.

On the data set without method preconditions (Tab. 1, rows marked *No M. P.*). Relaxing these constraints makes the resulting problems harder to solve. Coverage of the progression-based system drops in all categories. The same holds for the parsing-based verifier on the TO benchmarks. Like before, the compilation with the SAT-based HTN planning system is worse in showing unsolvability. Interestingly, when showing the unsolvability of a PO instance, the parsing-based and SAT-based systems have a higher coverage than our compilations. It seems that the planning systems are not tailored towards showing unsolvability.

Fig. 1 and 2 show the accumulated coverage in % over the runtime in seconds of the systems for the TO setting including method preconditions. Fig. 1 includes data on *valid* instances, Fig. 2 *invalid* instances. Be aware the log scale on the time axis. In the "valid" setting, the compilation with progression search solves its instances very fast. It reaches the highest coverage, followed by the SAT-based planner, which is slower than both the progression and the parsing-based approach. While the parsing-based approach is faster than the SAT-based planner, it has the lowest coverage. The behavior in the "invalid" setting is similar regarding the progression- and the parsing-based systems, but the SAT-based planner has a much smaller coverage.

Discussion & Conclusion

In this paper we introduced a compilation from HTN plan verification to HTN planning. Our second contribution is a novel benchmark set based on the models of the 2020 IPC.

A compilation-based approach has the advantage that no specialized verification systems are needed, and that performance will improve automatically with further progress in HTN planning (which, in turn, makes the problems harder that need to be verified). Our system is currently the only verifier supporting the entire feature set used in the IPC. The empirical evaluation shows that our approach outperforms the systems from related work.

A possible criticism of a compilation-based approach might be that one has to rely on the correctness of the applied planning system. The question is why we rely more on these systems than on the system that generated the plan to be verified. Since HTN planners are complex systems, these might also be incorrect (though this is also the case for specialized verification systems, of course). However, the planning systems used in our evaluation return the decomposition steps performed to find a plan. 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 much simpler verification systems like the one used in the IPC.

In cases where verification fails, we simply return that no solution (of the compiled problem) was found. To provide a meaningful explanation on why verification failed, one could incorporate explanations for the unsolvability of the generated planning problem. Providing certificates for unsolvability of planning problems is an active field of research, at least in classical planning (Eriksson, Röger, and Helmert 2017; Eriksson and Helmert 2020). I.e., like for the solvers, we benefit from established research directions in planning and their future progress. Further, no verification system from the literature can provide such explanations.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 232722074 – SFB 1102 and Project-ID 452150823.

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.; Behnke, G.; and Bercher, P. 2021. On the Verification of Totally-Ordered HTN Plans. In *Proc. of the 3rd ICAPS Workshop on Hierarchical Planning (HPLAN)*, 44–48.

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 Computer Society.

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. 2020. 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.

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*, 35(3): 367–375.

de Silva, L.; Padgham, L.; and Sardina, S. 2019. HTN-Like Solutions for Classical Planning Problems: An Application to BDI Agent Systems. *Theor. Comput. Sci.*, 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.

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *Proc. of the 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 1955–1961. IJCAI/AAAI.

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. 2016. Assessing the Expressivity of Planning Formalisms through the Comparison to Formal Languages. In *Proc. of the 26th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 158–165. 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*, 35(3): 391–396.

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.; 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.