On Total-Order HTN Plan Verification with Method Preconditions – An Extension of the CYK Parsing Algorithm

Songtuan Lin[1], Gregor Behnke[2], Simona Ondrčková[3], Roman Barták[3], Pascal Bercher[1]

[1] School of Computing, The Australian National University, Canberra, Australia
[2] ILLC, University of Amsterdam, Amsterdam, The Netherlands
[3] Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
{songtuan.lin, pascal.bercher}@anu.edu.au, g.behnke@uva.ni, {ondrckova, bartak}@ktiml.mff.cuni.cz

Abstract

In this paper, we consider the plan verification problem for totally ordered (TO) HTN planning. The problem is proved to be solvable in polynomial time by recognizing its connection to the membership decision problem for context-free grammars. Currently, most HTN plan verification approaches do not have special treatments for the TO configuration, and the only one features such an optimization still relies on an exhaustive search. Hence, we will develop a new TOHTN plan verification approach in this paper by extending the standard CYK parsing algorithm which acts as the best decision procedure in general.

Introduction

The problem of plan verification is to decide, given a plan, whether it is a solution to a planning problem. The study of this problem has drawn increasing attentions in the last decade for its potential usages in benefiting the research on planning. For instance, an independent plan verifier is vital in International Planning Competition (IPC) for the purpose of verifying whether a plan produced by a participated planner is correct or not. Recently, several works have explored the possibility of employing plan verification technique in Human-AI interaction. For example, Behnke, Höller, and Biundo (2017) pointed out the connection between the plan verification problem and mixed-initial planning (Myers et al. 2003) where a planner shall iteratively adjust its output plan according to a user's change requests, and plan verification might also be seen as an approach for planning domain validation, i.e., deciding whether a planning domain is correctly engineered by a domain engineer, where a plan is given as a test case that is supposed to be a solution to a planning problem, and a failed verification indicates that there are some flaws in the domain.

In this paper, we consider the plan verification problem in Hierarchical Task Network (HTN) planning (Erol, Hendler, and Nau 1996; Geier and Bercher 2011; Bercher, Alford, and Höller 2019). We particularly focus on a special class of HTN planning problems called totally ordered (TO) HTN planning problems which plays a prominent role in HTN planning, as evidenced by the fact that TO planning problem benchmarks significantly outnumber partially ordered (PO) ones in the IPC 2020 on HTN Planning. In spite of the significance, most existing HTN plan verifiers (Behnke, Höller, and Biundo 2017; Barták et al. 2020; Höller et al. 2022) have no special treatments for TO problems, and the only one having such an optimization is by Barták et al. (2021b).

The core of our TO plan verification approach is the CYK parsing algorithm (Sakai 1961), which can be employed here because a TOHTN planning problem is semantically equivalent to a context-free grammar (CFG) (Höller et al. 2014, Thm. 6), and hence, the TOHTN plan verification problem is essentially the parsing problem for CFG. However, that result by Höller et al. (2014) does not take into account so-called *method preconditions* which occur quite often in practice in many TOHTN planning benchmarks and thus also become an obstacle to directly applying the CYK algorithm to plan verification. Consequently, we will extend the standard CYK algorithm to adapt method preconditions.

The idea of viewing an HTN plan verification problem as a parsing problem is widely used. For instance, Barták, Maillard, and Cardoso (2018) and Barták et al. (2020) exploited the connection between HTN planning problems and attributed grammars and proposed a parsing-based plan verification approach for general HTN planning problems, which can also be used to correct flawed HTN plans (Barták et al. 2021a) and is then extended to have the special treatments for the TO setting (Barták et al. 2021b). Notably, their treatments for TOHTN planning problems still rely on an exhaust search and thus have several overheads, which is mandatory because the approach takes into account some additional state constraints. However, those state constraints are rare in many TOHTN planning benchmarks. For this reason, we are not concerned with such constraints, which allows us to fully exploit the connection between TOHTN planning problems and CFGs and thus develop a more efficient TO plan verification approach.

HTN Formalism

In order to explain how our TO plan verification approach works, we first introduce the HTN formalism employed in the paper. Since we only consider TOHTN plan verification in this paper, the formalism presented here is targeted specifically at the TO configuration, and it is an adaption of the one by Geier and Bercher (2011), by Behnke, Höller, and Biundo (2018), and by Bercher, Alford, and Höller (2019). We start by presenting the definition of TOHTN planning problems and explain in detail each component in the definition later.

Definition 1. A totally ordered HTN planning problem $\mathcal{P}$ is a tuple $(\mathcal{D}, tn_I, s_I)$ where $\mathcal{D} = (F, N_c, N_p, M, \delta)$ is called the domain of $\mathcal{P}$. F is a finite set of propositions, N_c is a finite set of *compound* task names, N_p is a finite set of *primitive* task names, M is a finite set of methods m with $m \in 2^F \times N_c \times (N_c \cup N_p)^*$, and $\delta : N_p \to 2^F \times 2^F \times 2^F$ is a function. $s_I \in 2^F$ and $tn_I \in (N_c \cup N_p)^*$ are called the initial state and the initial task network (or the goal task network) of $\mathcal{P}$, respectively.

We also define a $tn \in (N_c \cup N_p)^*$ as a task network, which is a sequence of task names.

In the definition presented above, task names are categorized as being primitive and compound. A primitive task name p, also called an action, is mapped to the respective precondition, add list, and delete list by the function δ, written $\delta(p) = (prec(p), add(p), del(p))$, where $prec(p)$, $add(p)$, and $del(p)$ respectively refer to the preconditions, add list, and delete list of p, each of which is a set of propositions. A primitive task name p is applicable in a state $s \in 2^F$, *iff* $prec(p) \subseteq s$, and we say that a state s' is a consequence of applying a primitive task p in a state s, written $s \to_p s'$, *iff* p is applicable in s, and $s' = (s \backslash del(p)) \cup add(p)$. Similarly, a state trajectory $\langle s_0 \cdots s_n \rangle$ is a consequence of applying a sequence of primitive task names $tn = \langle p_1 \cdots p_n \rangle$ with $n \in \mathbb{N}_0$, i.e., a primitive task network, in a state s *iff* $s_0 = s$, and for each $1 \leq i \leq n$, $s_{i-1} \to_{p_i} s_i$, and we say that the state s_n is obtained by applying tn in s, written $s \to^*_{tn} s_n$.

On the other hand, a compound task c in a task network could be rewritten as another task network tn by a method $m = (prec(m), c, tn)$ where $prec(m)$ refers to the precondition of m. We call this process the decomposition of c, written $c \to_m tn$. We will also omit the subscript m in the notation, i.e., $c \to tn$, to indicate that there *exists* some method which decomposes c into tn. m can be applied to decomposing c *if and only if* its precondition is satisfied. We will elaborate how to determine whether the precondition of a method is satisfied (i.e., the semantics of method preconditions) later on. The concept of decompositions can also be extended to task networks:

Definition 2. Let tn and tn' be two task networks where tn is of the form $tn = \langle tn_1 \; c \; tn_2 \rangle$ with c being a compound task and tn_1 and tn_2 being two sequences of task names, each of which might be empty, and $m = (prec(m), c, \hat{tn})$ be a method. We say that tn is decomposed into tn' by m, written $tn \to_m tn'$, if $tn' = \langle tn_1 \; \hat{tn} \; tn_2 \rangle$. Similarly, we write $tn \to tn'$ to indicate that there *exists* some method which decomposes tn into tn'. We also write $tn \to^*_{\overline{m}} tn'$ if tn is decomposed into tn' by a *sequence* $\overline{m}$ of methods and $tn \to^* tn'$ if there exists such a method sequence.

For any two task networks tn and tn' with $tn \to^* tn'$, a compound task c in tn is eventually decomposed into a continuous subsequence $\hat{tn}$ in tn' (Barták et al. 2021b). Hence, we abuse the notation to let $c \to^* \hat{tn}$ denote that the compound task c in some task network is decomposed into the continuous subsequence $\hat{tn}$ of another task network by a sequence of methods, and we write $c \to^*_{\overline{m}} \hat{tn}$ if such a method sequence $\overline{m}$ is understood in the context.

Although a method sequence could capture the decomposition of a task network (or a compound task), it is ambiguous because it does not specify the correspondence between the methods and the compound tasks occurring in the decomposition process. In order to address this, we introduce the notion of *decomposition trees* based upon the one by Geier and Bercher (2011) which characterizes a decomposition process unambiguously.

Definition 3. Given a TOHTN planning problem $\mathcal{P}$, a decomposition tree $g = (\mathcal{V}, \mathcal{E}, \prec_g, \alpha_g, \beta_g)$ with respect to $\mathcal{P}$ is a labeled directed tree where $\mathcal{V}$ and $\mathcal{E}$ are the sets of vertices and edges, respectively, $\prec_g$ is a *total order* defined over $\mathcal{V}$, $\alpha_g : \mathcal{V} \to N_p \cup N_c$ labels a vertex with a task name, and β_g maps a vertex $v \in \mathcal{V}$ to a method $m \in M$. Particularly, g is *valid* if it is rooted at a vertex r with $\alpha_g(r) = c_I$, and for every inner vertex v whose children in the total order $\prec_g$ forms the sequence $\langle v_1 \cdots v_n \rangle$ ($n \in \mathbb{N}$), if $\alpha(v) = c$ for some $c \in N_c$, then $\beta_g(v) = m$ for some $m \in M$ with $m = (prec(m), c, tn)$ and $tn = \langle \alpha_g(v_1) \cdots \alpha_g(v_n) \rangle$.

Let $\langle l_1 \cdots l_n \rangle$ ($n \in \mathbb{N}$) be the leafs of g ordered in $\prec_g$. We define the yield of g, written $yield(g)$, as the task network $\langle \alpha_g(l_1) \cdots \alpha_g(l_n) \rangle$. For convenience, we will simply use $\mathcal{L}(g)$ to refer to the leafs of g ordered in $\prec_g$.

Having the definition of decomposition trees in hand, we can now define the semantics of method preconditions.

Definition 4. Let $\mathcal{P}$ be a TOHTN planning problem, g a valid decomposition tree g with respect to $\mathcal{P}$ where $\mathcal{L}(g) = \langle l_1 \cdots l_n \rangle$ and $yield(g) = \langle \alpha_g(l_1) \cdots \alpha_g(l_n) \rangle$ ($n \in \mathbb{N}$) consists solely of primitive tasks, and $m = (prec(m), c, tn)$ a method with $\beta_g(v) = m$ for some inner vertex $v \in \mathcal{V}$. The precondition of m is satisfied if and only if for the first vertex l_i ($1 \leq i \leq n$) in $\mathcal{L}(g)$ that is a descendant of v, it holds that $prec(m) \subseteq s_{i-1}$ with $s_I \to^*_{\overline{tn}'} s_{i-1}$ and $\overline{tn}' = \langle \alpha_g(l_1) \cdots \alpha_g(l_{i-1}) \rangle$. For $i = 1$, we define $s_0 = s_I$.

Lastly, we define the solution criteria for TOHTN planning problems.

Definition 5. Given a TOHTN planning problem $\mathcal{P}$, a solution to $\mathcal{P}$ is a task network tn consisting solely of primitive tasks such that tn is executable in s_I, i.e., $s_I \to^*_{tn} s$ for some $s \in 2^F$, and there exists a valid decomposition tree g with respect to $\mathcal{P}$ such that $yield(g) = tn$ and for every inner vertex v of g with $\beta_g(v) = m$ for some $m \in M$, the precondition of m is satisfied.

TOHTN Plan Verification

Having presented the TOHTN planning formalism, we now move on to introduce our CYK-based TOHTN plan verification approach. The basis for using the standard CYK parsing algorithm in TOHTN plan verification is that primitive tasks, compound tasks, and methods in TOHTN planning problems are respectively analogy to terminal symbols, nonterminal symbols, and production rules in CFGs. Consequently, the TOHTN plan verification problem is analogy to the membership decision problem for CFGs, which is what the CYK algorithm targeted at.

The CYK algorithm demands that an input CFG (*resp.* a TOHTN planning problem) should be in Chomsky Normal Form (Chomsky 1959) where every production rule (*resp.* a

Algorithm 1: The CYK-based plan verification approach. The lines without being numbered are the standard CYK algorithm, and those being numbered are the modifications for adapting method preconditions and 2RF.

Input: A plan $\pi = \langle p_1 \cdots p_n \rangle$
    A planning problem $\mathcal{P}$ in 2RF
Output: True or false depending on whether π is a solution to $\mathcal{P}$
▷ Let $\langle s_0 \cdots s_n \rangle$ be the state sequence *s.t.* $s_0 = s_I$, and $s_{i-1} \rightarrow_{p_i} s_i$ for each $i \in \{1 \cdots n\}$
for $i \leftarrow n$ to 1
  $A[i,i] = \{c \mid c \rightarrow \langle p_i \rangle\} \cup \{p_i\}$
  for $j \leftarrow i$ to n
    for $k \leftarrow i$ to $j-1$
      for $m \in \left\{ m \;\middle|\; \begin{array}{l} m = (prec(m), c, tn), \\ tn = \langle c'_1\, c'_2 \rangle, c'_1 \in A[i,k], \\ c'_2 \in A[k+1,j] \end{array} \right\}$
        ▷ Checking the method precondition
8:       **if** $prec(m) \subseteq s_{i-1}$
          $A[i,j] \leftarrow A[i,j] \cup \{c\}$
    ▷ Finding the unit productions
11:    **for** $\overline{m} \in \left\{ \overline{m} \;\middle|\; \begin{array}{l} c' \rightarrow^*_{\overline{m}} \langle c \rangle, c' \in N_c, \\ c \in A[i,j] \end{array} \right\}$
12:     **if** $prec(m) \subseteq s_{i-1}$ for each m in $\overline{m}$
13:       $A[i,j] \leftarrow A[i,j] \cup \{c'\}$
if $c_I \in A[1,n]$ **return** true
else return false

method) decomposes a non-terminal symbol (*resp.* a compound task) into two non-terminal symbols or into a terminal symbol (*resp.* a primitive task). It then determines whether a string is in the language of the CFG (*resp.* whether a plan is a solution to the planning problem) by constructing parse trees (*resp.* decomposition trees) in a bottom-up manner.

More concretely, given a string (*resp.* a plan) $\langle p_1 \cdots p_n \rangle$ ($n \in \mathbb{N}$), the ultimate goal of the CYK algorithm is to find, for each subsequence $\pi^i_j = \langle p_i \cdots p_j \rangle$ ($1 \leq i \leq j \leq n$), the set $A[i,j]$ of all possible non-terminal symbols c such that $c \rightarrow^* \pi^i_j$, i.e., c can be decomposed into π^i_j by a sequence of production rules (methods). Mathematically, this goal can be accomplished via the following recursion formula:

$$A[i,j] = \begin{cases} \{c \mid c \rightarrow \langle p_i \rangle\} & \text{if } i = j \\ \left\{ c \;\middle|\; \begin{array}{l} c \rightarrow \langle c'_1\, c'_2 \rangle, i \leq k < j \\ c'_1 \in A[i,k], c'_2 \in A[k+1,j] \end{array} \right\} & \text{if } i < j \end{cases}$$

The interpretation of the formula is that, for each $1 \leq i \leq n$, a non-terminal symbol c is in the set $A[i,i]$ if it can be decomposed into the terminal symbol p_i by some production rule, and for each i, j with $1 \leq i < j \leq n$, $A[i,j]$ has a non-terminal symbol c if c can be decomposed into two other non-terminal symbols c'_1 and c'_2 by some production rule such that there exists a k with $i \leq k < j$, $c'_1 \in A[i,k]$, and $c'_2 \in A[k+1,j]$, i.e., $c'_1 \rightarrow^* \pi^i_k$ and $c'_2 \rightarrow^* \pi^{k+1}_j$. Notably, the recursion holds because we make the restriction that the input CFG must be in CNF.

In the CYK algorithm, the recursion is implemented via dynamic programming where a two dimension table is constructed to memorize each entry $A[i,j]$ ($1 \leq i \leq j \leq n$), and the table is filled in a right-left, bottom-up order. The implementation is shown by Alg. 1 in which the lines without being numbered are the code for the standard CYK algorithm, and we substitute every component in CFGs (i.e., terminal/non-terminal symbols, production rules, etc.) with its counterpart in TOHTN planning problems.

In order to adapt the CYK algorithm in TOHTN plan verification, we have to deal with method preconditions whose counterpart does not exist in CFGs. This is however trivial because we can simply check whether a method's precondition is satisfied when filling the table, see Alg. 1, line 8.

Though the procedure presented above can already serve as a mature TOHTN plan verification approach, it relies on the strict constraint that an input planning problem must be in CNF. It is unfortunately not trivial to transform an arbitrary TOHTN planning problem into CNF. Similar to how such transformation is done for CFGs (Hopcroft, Motwani, and Ullman 2007; Lange and Leiß 2009), transforming a TOHTN planning problem into CNF usually requires four steps ordered as follows:

1) *binarization*: splitting every method such that it contains *at most* two subtasks,
2) *deletion*: deleting all methods and tasks which will result in the empty task network,
3) *elimination*: eliminating all unit productions, and
4) *termination*: enforcing that for any method, if it contains only one subtask, then the task is a primitive one.

As pointed out by Lange and Leiß (2009), the four steps (the third one in particular) for transforming a CFG into CNF will lead to a quadratic explosion of the size of the grammar, which is also the case for a TOHTN planning problem. This is a significant overhead for TOHTN plan verification because usually a planning problem already contains an enormous number of methods. Further, due to the existence of method preconditions, the four-step transformation on TOHTN planning problems need to modify method preconditions accordingly.

In ordered to avoid these overheads, we only apply the first step *binarization* to an input TOHTN planning problem and result in the planning problem being in so-called 2-regularation Form (2RF) (Behnke and Speck 2021), i.e., in which every method contains *at most* two subtasks (could be either primitive or compound). Note that 2RF also has its counterpart in CFGs called 2-normal Form (2NF) (Lange and Leiß 2009). The advantage of adapting 2RF is that we could avoid the quadratic explosion of the size of a problem and the additional modifications to method preconditions.

The price for adapting 2RF instead of CNF is that we have to merge the remaining three transformation steps into the plan verification procedure. That is, after computing an entry $A[i,j]$ in the standard CYK algorithm, we shall also search for all compound tasks $c' \in N_c$ such that $c' \rightarrow^* \langle c \rangle$ for some $c \in A[i,j]$, and the precondition of every method occurring in the decomposition process is satisfied. This is equivalent to finding *all* method sequences $\overline{m}$ such that the precondition of each method in it is satisfied, and $c' \rightarrow^*_{\overline{m}} \langle c \rangle$ for some $c' \in N_c$ and $c \in A[i, j]$, and such compound tasks c' should then also be included in $A[i, j]$ (Alg. 1, lines 11 to 13).

For this purpose, we can do the following two things. First, we want to find *all* compound tasks c and *all* method sequences $\overline{m}$ such that $c \rightarrow^*_{\overline{m}} \varepsilon$ where ε refers to the empty task network. We call such a c a nullable task which is analogy to a nullable symbol in CFGs. We can directly apply the procedure for finding all nullable symbols in a CFG to finding all nullable tasks in a TOHTN planning problem together with all method sequences that decompose them into the empty task network. For the details about how this procedure works, we refer to the work by Hopcroft, Motwani, and Ullman (2007). Second, we will construct a graph $G = (V, E)$ with $V = M$, i.e., the vertices are the methods of the planning problem, and an edge $(m', m) \in E$ with $m' = (prec(m'), c', tn')$ and $m = (prec(m), c, tn)$ *iff* either $tn' = \langle c \rangle$ or $tn' = \langle t_0\, t_1 \rangle$ such that there exists an $i \in \{0, 1\}$ with $t_i = c$ and t_{1-i} being a nullable task. We name such a graph as a *unit production graph*.

Having identified all nullable tasks in a planning problem, the unit production graph can be constructed by simply iterating through all methods in the planning problem. As an example about how to construct the graph, consider a method $m = (prec(m), c, \langle c'_0\, c'_1 \rangle)$ where c'_0 is a nullable task and $c'_1 \in N_c$. For each method m' that can decompose c'_1, we add the edge (m, m') to the graph (see the appendix for more details about the construction).

For finding all method sequences $\overline{m}$ such that $c' \rightarrow^*_{\overline{m}} \langle c \rangle$ for some $c', c \in N_c$. We observe one key property of the unit production graph G in which, for any two compound tasks c' and c, $c' \rightarrow^* \langle c \rangle$ *iff* there exists a path in G from m' to m such that m' and m respectively decompose c' and c. Consequently, we only need to conduct several depth-first search in the *reverse* graph of G (i.e., reversing the direction of each edge in G), each of which starts from a vertex m which can decompose c and ends at a vertex m' which decomposes c'. For each found method sequence $\overline{m} = \langle m_1 \cdots m_k \rangle$, we shall also check whether the precondition of each m_i ($1 \leq i \leq k$) in it is satisfied. Notably, if m_i contains a subtask t which is nullable, then we must also check whether there exists a method sequence $\overline{m}'$ such that $t \rightarrow^*_{\overline{m}'} \varepsilon$ and the precondition of every method in it is satisfied. This is trivial because for each nullable task, we have already found all method decomposing it into the empty task network.

Taking together, Alg. 1 summarizes the procedure of our TOHTN plan verification approach, given a planning problem in 2RF. We first implement the standard CYK algorithm for computing each table entry $A[i, h]$, and then for each such entry $A[i, j]$, we find all method sequences $\overline{m}$ such that $c' \rightarrow^*_{\overline{m}} \langle c \rangle$ for some $c' \in N_c$ and $c \in A[i, j]$ and check whether all method preconditions in the sequence are satisfied. If so, we then add c' to $A[i, j]$ (see the appendix for more implementation details).

Empirical Evaluation

We ran the experiments on a Xeon Gold 6242 CPU. For each instance, each verifier was given 10 minutes of runtime and 8 GB of RAM. The experiments were done both on the TO

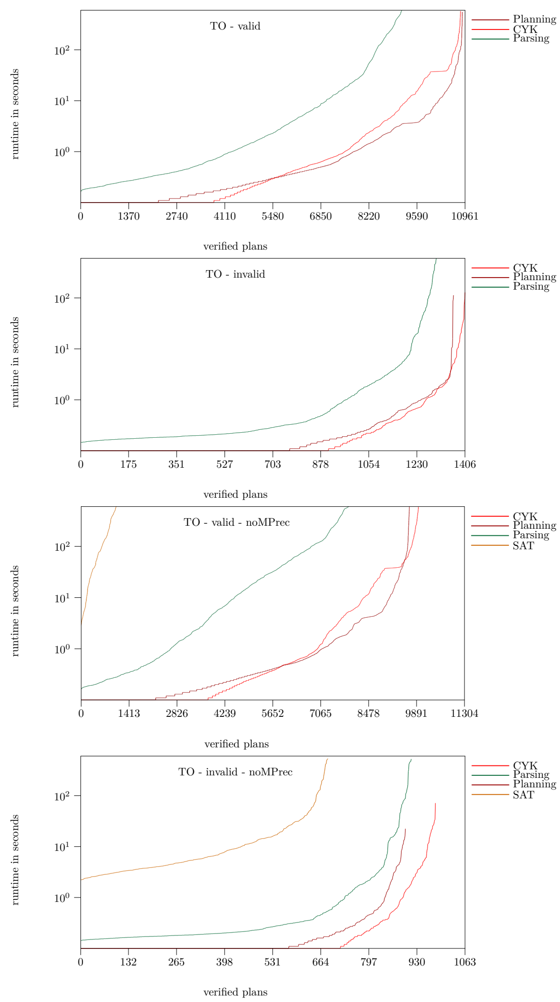

Figure 1: The number of solved instances against runtimes.

benchmark set which have method preconditions and on the one which does not. The benchmark set with method preconditions are from the IPC 2020 on HTN Planning which contain 12367 plan instances from 24 domains where 10961 instances are valid, i.e., those are solutions to some planning problems, and the remaining 1406 instances are invalid. The benchmark set without method preconditions is again from the IPC 2020 on HTN Planning, and it is obtained by discarding method preconditions in original planning problems. This set again contains 12367 instances where 11264 are valid, and 1103 are invalid (note the increasing number of valid instances after removing method preconditions).

Experiment Results

We compared our CYK-based approach with the parsing-based one by Barták et al. (2021b), which is the current state-of-the-art TO plan verifier, and with two general (i.e., PO) plan verifiers which can also be employed in verifying TO plans, i.e., the SAT-based one by Behnke, Höller, and Biundo (2017) and the planning-based one by Höller et al. (2022), which respectively transform a verification problem

Benchmark	Instances	Parsing-based		Planning-based		SAT-based		CYK-based (Ours)	
to-val	10961	9158	(83.55)	**10881**	(**99.27**)	*no support*		10832	(98.82)
to-inval	1406	1301	(92.53)	1364	(97.01)	*no support*		**1406**	(**100.00**)
to-val-no-mprec	11304	7889	(69.79)	9679	(85.62)	1036	(9.16)	**9946**	(**87.99**)
to-inval-no-mprec	1063	915	(86.08)	898	(84.48)	684	(64.35)	**981**	(**92.29**)

Table 1: Table comparing runs of multiple approaches for plan verification. For each verifier, the number in each row indicates the number of solved instances in the corresponding benchmark set, and the respective percentage indicates the coverage rate.

into a SAT problem and an HTN planning problem.

In the experiments on the benchmark set with method preconditions, we did not consider the SAT-based verifier because it does not support method preconditions. For the valid instances, the planning-based verifier achieve the best performance by solving 10881 instances (99.27%). Our approach slightly underperforms it by solving 10832 instances (98.82%) and beats the parsing-based one which solves 9158 instances (83.55%). For the invalid instances, our approach solved all 1406 instances (100%) compared with the planning-based one and the parsing-based one which solve 1364 instances (97.01%) and 1301 instances (92.53%), respectively. The results are summarized in Tab. 1 where the rows to-val and to-inval respectively indicate the valid and invalid instances.

In the experiments on the benchmark set without method preconditions, we included the SAT-based verifier. Our approach outperforms the others in solving both valid and invalid instances. Specifically, our verifier solved 9946 valid instances (87.99%) and 981 invalid instances (92.29%). The planning-based one solved 9679 valid instances (85.62%) and 898 invalid instances (84.48%), and the parsing-based one solved 7889 valid instances (69.79%) and 915 invalid instances (86.08%). The SAT-based verifier has the worst performance, which only solved 1036 valid instances (9.16%) and 684 invalid ones (64.35%), see the last two rows in Tab. 1 for the summary.

Further, Fig. 1 depicts the number of solved instances (the x-axis) against the runtimes (the y-axis), i.e., how many instances can be solved in a specific runtime, for the evaluations of both valid and invalid instances on the two benchmark sets. One might observe that in solving the instances with method preconditions, our approach has the similar performance compared with the planning based one and outperforms the parsing based one. For those without method preconditions, our approach clearly beats the others.

Discussion

We now give some discussion over our CYK-based plan verification approach compared with others, i.e., the parsing-based, the SAT-based, and the planning-based approach.

According to the experiment results, our approach outperforms the parsing-based one (Barták et al. 2020, 2021b) which is the only one by now having the special treatments for the TO configuration. We believe that the major reason for the underperformance of the parsing-based approach is that the approach does not restrict the number of subtasks in each method. As a consequence, the parsing-based approach, which, like our CYK-based approach, try to find all possible compound tasks that can be decomposed into a subsequence of a given plan, relies on an exhaustive search for that purpose.

For example, in our approach, in order to decide whether a compound task c can be decomposed into a subsequence $\pi^i_j = \langle p_i \cdots p_j \rangle$ via a method $m = (prec(m), c, \langle c'_1\, c'_2 \rangle)$, we only have to check whether $c'_1 \in A[i,k]$ and $c'_2 \in A[k+1,j]$ for some $i \leq k < j$. In contrast, in the parsing-based approach, checking whether c can be decomposed into π^i_j via a method which has k ($k \in \mathbb{N}$) subtasks $\langle c'_1 \cdots c'_k \rangle$ requires deciding whether π^i_j can be divided into k subsequences $\pi^i_j = \langle \pi'_1 \cdots \pi'_k \rangle$ such that $c'_r \to^* \pi'_r$ for each $1 \leq r \leq k$. The latter one is clearly more computationally expensive.

Notably, the parsing-based approach does not restrict the number of subtasks in a method for the purpose of supporting an additional state constraint imposed by the method called the *between-constraint* which must hold *between* the start and the end of the subsequence of the plan obtained from the method. Although it is possible to transform a TO-HTN planning problem into 2RF (or CNF) while maintaining these additional constraints, it might cause an unavoidable quadratic explosion of the problem size, which is another significant overhead. Further, despite that the parsing-based approach support such an additional constraint, the benchmark set on which we did the empirical evaluation does not feature it, and hence, this extra functionality will not incur overheads to the approach in the experiments.

For the planning-based approach (Höller et al. 2022), it outperforms our approach in the experiment of verifying valid plan instances with method preconditions and underperforms ours in the remaining three experiments. We hypothesize that the outstanding performance of the planning-based approach in verifying valid plans is due to the heuristics employed by the TOHTN planner which solves the planning problem transformed from a plan verification problem. Particularly, the heuristics might rule out some methods in advance whose preconditions are not satisfiable and henceforth significantly reduce the search space, as evidenced by its underperformance in solving instances without method preconditions. On the other hand, the heuristics might be less powerful when confronting an unsolvable planning problem (i.e., verifying an invalid plan), which might be the reason for why it underperforms the CYK-based approach in verifying invalid plans (with or without method preconditions). Generally speaking, we argue that our CYK-based approach as a decision is still better than the planning-based approach.

Lastly, the SAT-based approach has the worst performance compared with others. We hypothesize that this is because phrasing a plan verification problem as a SAT formula is already computationally expensive, and solving a SAT problem is $\mathbb{NP}$-hard as well.

Future Works

The TOHTN plan verification approach developed in the paper is based on the CYK algorithm, which is in the family of the so-called *chart parsers*. Some parsing algorithms in the family, e.g., the LR parsing algorithm, have been proved to be more efficient than the CYK algorithm when an input CFG (*resp.* a planning problem) has certain properties, and some, e.g., the Earley parsing algorithm (Earley 1970), do not require any special format of input CFGs while still maintaining reasonable time complexity. Thus, in future works, we would like to explore the possibility of employing other chart parsers in TOHTN plan verification and henceforth make the connection between TOHTN planning and formal languages more strong.

Conclusion

In this paper, we developed a totally ordered HTN plan verification approach that are tailored to method preconditions by extending the standard CYK parsing algorithm. The empirical evaluation results show that our approach significantly outperforms another parsing-based plan verification approach by Barták et al. (2020; 2021b) which is also the only approach by now features the special treatments for the TO configuration. Further, though the approach slightly underperforms the state-of-the-art plan verifier by Höller et al. (2022) when input plans are indeed solutions, it has better performance when an input plan is invalid. Additionally, our approach always has better performance when method preconditions are not considered independent of whether an input plan is valid or not. We thus still regard our approach as a better decision procedure.

Acknowledgment

Simona Ondrčková is (partially) supported by SVV project number 260 575 and by the Charles University project GA UK number 280122.

References

Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of Hierarchical Plans via Parsing of Attribute Grammars. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling, ICAPS 2018*, 11–19. AAAI.

Barták, R.; Ondrčková, S.; Behnke, G.; and Bercher, P. 2021a. Correcting Hierarchical Plans by Action Deletion. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021*, 99–109. IJCAI.

Barták, R.; Ondrčková, S.; Behnke, G.; and Bercher, P. 2021b. On the Verification of Totally-Ordered HTN Plans. In *Proceedings of the 33rd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2021*, 263–267. IEEE.

Barták, R.; Ondrčková, S.; Maillard, A.; Behnke, G.; and Bercher, P. 2020. A Novel Parsing-based Approach for Verification of Hierarchical Plans. In *Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2020*, 118–125. IEEE.

Behnke, G.; Höller, D.; and Biundo, S. 2017. This Is a Solution! (... But Is It Though?) - Verifying Solutions of Hierarchical Planning Problems. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling, ICAPS 2017*, 20–28. AAAI.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT - Totally-Ordered Hierarchical Planning Through SAT. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, 6110–6118. AAAI.

Behnke, G.; and Speck, D. 2021. Symbolic Search for Optimal Total-Order HTN Planning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence, AAAI 2021*, 11744–11754. AAAI.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning - One Abstract Idea, Many Concrete Realizations. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019*, 6267–6275. IJCAI.

Chomsky, N. 1959. On Certain Formal Properties of Grammars. *Information and Control*, 2(2): 137–167.

Earley, J. 1970. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2): 94–102.

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

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011*, 1955–1961. IJCAI.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In *Proceedings of the 21st European Conference on Artificial Intelligence, ECAI 2014*, 447–452. IOS.

Höller, D.; Wichlacz, J.; Bercher, P.; and Behnke, G. 2022. Compiling HTN Plan Verification Problems into HTN Planning Problems. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling, ICAPS 2022*. AAAI.

Hopcroft, J. E.; Motwani, R.; and Ullman, J. D. 2007. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Lange, M.; and Leiß, H. 2009. To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm. *Informatica Didactica*, 8: 1–21.

Myers, K. L.; Jarvis, P.; Tyson, M.; and Wolverton, M. 2003. A Mixed-initiative Framework for Robust Plan Sketching. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling, ICAPS 2003*, 256–266. AAAI.

Sakai, I. 1961. Syntax in Universal Translation. In *Proceedings of the International Conference on Machine Translation and Applied Language Analysis*, 593 – 608.

Appendix

In the appendix, we would like to elaborate more technical details about our CYK-based TOHTN plan verification approach. In our approach (as well as the CYK algorithm), when computing an entry $A[i, j]$, we have to find all methods m (*resp.* production rules) such that $c \to_m \langle c'_1\, c'_2\rangle$ for some $c \in N_c$, $k \in \{i \cdots j-1\}$, $c'_1 \in A[i, k]$, and $c'_2 \in A[k+1, j]$. Most literature about the CYK algorithm in the context of formal languages accomplish this step via iterating through *all* production rules. This is however *not* efficient in the context of plan verification. The reason for this is that in a CFG, the number of production rules is considered to be relatively smaller than the length of a string, whereas this is not the case in plan verification. For instance, some TOHTN planning problem could have more than 10 thousands methods compared with the length of an input plan which is normally below one thousand.

Thus, in order to eliminate this overhead, we maintain two mappings $\varphi_1 : N_p \to M$ and $\varphi_2 : N \times N \to M$ where $N = N_p \cup N_c$. Specifically, given a $p \in N_p$, $\varphi_1(p) = m$ for some $m \in M$ *iff* m decomposes some compound task into $\langle p\rangle$, and similarly, given $t_1, t_2 \in N$, $\varphi_2(t_1, t_2) = m$ *iff* m decomposes a compound task into $\langle t_1\, t_2\rangle$. Consequently, given two entries $A[i, k]$ and $A[k + 1, j]$ (or one single entry $A[i, i]$), we can quickly find all methods which decompose a compound task into two (or one) subtask(s) that are (is) in the respective entries (entry) by visiting the mapping(s).

Next, we would like to give a recursive procedure for finding all nullable tasks together with the respective method sequences that decompose them into the empty task networks. The procedure is not given in the main paper. We instead refer to the work by Hopcroft, Motwani, and Ullman (2007) for the procedure. The procedure works as follows:

Basis: If $c \to_m \varepsilon$ for some $m \in M$, then c is a nullable task, and we mark $\langle m\rangle$ as a method sequence that decomposes c into the empty task network.

Induction: If $c \to_m \langle t_1\, t_2\rangle$ (or $c \to_m \langle t\rangle$) for some $m \in M$ and t_1, t_2 (or t) are (is) nullable, then c is also nullable, and for *any* two method sequences $\overline{m}_1$ and $\overline{m}_2$ that respectively decompose t_1 and t_2 into the empty task network (or any $\overline{m}$ with $t \to^*_{\overline{m}} \varepsilon$), $\langle m\ \overline{m}_1\ \overline{m}_2\rangle$ (or $\langle m\ \overline{m}\rangle$) together with any permutation of it is marked as a method sequence that decomposes c into ε.

Having identified all nullable tasks in a planning problem, we can then construct the *unit production graph*. The procedure for constructing the graph is described as follows: For each method $m \in M$ with $m = (prec(m), c, tn)$,

- if $tn = \langle t_0\ t_1\rangle$ for some $t_0, t_1 \in N$ ($N = N_c \cup N_p$), and there exists an $i \in \{0, 1\}$ such that t_{1-i} is nullable, then for *each* m' that can decompose t_i, we add the edge (m, m') to the graph, or
- if $tn = \langle t\rangle$ for some $t \in N_c$, then for each method m' that decomposes t, we add the edge (m, m') to the graph.

Now that we have clarified how to construct a unit production graph, we move on to prove the most important property of such a graph, that is, for any two compound tasks c and c', c can eventually be decomposed into c' *iff* there exists a path connecting two methods m and m' in the graph which respectively decomposes c and c'.

Theorem 1. *Let $c, c' \in N_c$, $c \to^* \langle c'\rangle$ if and only if there is a path in the unit production graph $G = (V, E)$ from m to m' such that m decomposes c and m' decomposes c'.*

Proof. ($\Longrightarrow$): We prove this by induction on the number of steps in decomposing c into c'. The base case is $c \to \langle c'\rangle$. In this case, a path (m', m) with c' being decomposed by m' exists by the construction of the graph G.

Now suppose that $c \to^* \langle c'\rangle$ in k steps ($k > 1$), it follows that there must exist a method m which decomposes c into a task network tn such that either tn containing only one subtask task $\hat{c}$ that is in N_c or tn consisting two subtasks where one is nullable, and the other $\hat{c}$ is decomposed into c', because otherwise, c cannot be decomposed into c'. For both cases, we have that $\hat{c} \to^* \langle c'\rangle$ in $k - 1$ steps. By the induction hypothesis, there exists a path from $\hat{m}$ to m' in the graph such that m' decomposes c' and $\hat{m}$ decomposes $\hat{c}$. Further, by the construction of the graph, $(m, \hat{m}) \in E$, and hence, there is a path in G from m to m'.

($\Longleftarrow$): We prove this by induction on the length of the path from m to m'. The base case is that $(m, m') \in E$. By construction, m decomposes a compound task c into a task network tn such that either $tn = \langle c'\rangle$ or $tn = \langle t_0\ t_1\rangle$ in which there exists an $i \in \{0, 1\}$ with $t_i = c'$ and t_{1-i} is nullable. For the former, $c \to c'$ holds naturally, and for the latter, since $t_{1-i} \to^* \varepsilon$ (because t_{1-i} is nullable), it follows immediately that $c \to^* \langle c'\rangle$.

For the case where a path from m to m' has length k ($k > 1$), the path can be divided as two parts: a path from $\hat{m}$ to m' of length $k - 1$ and an edge $(m, \hat{m}) \in E$. By the induction hypothesis, there exist $\hat{c} \in N_c$ with $\hat{c}$ being decomposed by $\hat{m}$ such that $\hat{c} \to^* \langle c'\rangle$. Further, by the construction of the graph G, the presence of the edge $(m, \hat{m})$ implies that m decomposes a compound task c into a task network tn in which either $\hat{c}$ is the only subtask, or tn contains two subtasks where one is $\hat{c}$ and the other is nullable. For both cases, we have $c \to^* \langle \hat{c}\rangle$ and henceforth $c \to^* \langle c'\rangle$. ☐

The correctness of Alg. 1 thus follows immediately.

Theorem 2. *A plan $\pi = \langle p_1 \cdots p_n\rangle$ is a solution to a planning problem $\mathcal{P}$ if and only if Alg. 1 returns true.*

Lastly, we would like to discuss the time complexity of our CYK-based plan verification approach. For an input plan $\langle p_1 \cdots p_n\rangle$, one can easily recognize that the time required for visiting all entries $A[i, j]$ ($1 \le i \le j \le n$) is $\mathcal{O}(n^3)$. Further, when computing each entry $A[i, j]$, we need to visit at most all $|M|$ methods for finding all $c \in N_c$ with $c \to^* \langle c'\rangle$ and $c' \in A[i, j]$. Therefore, the time complexity of the CYK-based plan verification approach is $\mathcal{O}(|M| \times n^3)$.

Theorem 3. *Alg. 1 has the time complexity $\mathcal{O}(|M| \times n^3)$.*

Note that the time complexity of the CYK-based approach also emphasize the importance of maintaining the mappings φ_1 and φ_2 because, as said, $|M|$ is normally larger than $|n|$ in plan verification, and hence, if we visit all methods in each iteration like what is done in most literature, the actual time complexity in practice would be $\mathcal{O}(n^4)$.