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

Songtuan Lin1, Gregor Behnke2, Simona Ondrčková3, Roman Barták3, Pascal Bercher1
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
{singtuan.lin, pascal.bercher}@anu.edu.au, {g.behnke@uva.nl, {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 whether a given plan 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 verifying whether participating planner produces a correct plan. Recently, several works have explored the possibility of deploying plan verification techniques in Human-AI interaction. For example, Behnke, Höller, and Biundo (2017) pointed out the connection between plan verification 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 (Lin and Bercher 2021, 2023; Lin, Grastien, and Bercher 2023), i.e., deciding whether a planning domain is correctly modeled, 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 total-order (TO) HTN planning problems which plays a prominent role in HTN planning, as evidenced by the fact that TO planning problem benchmarks significantly outnumber partial-order (PO) ones in the IPC 2020 on HTN Planning. In spite of the significance, most existing HTN plan verification approaches (e.g., see the work by Behnke, Höller, and Biundo (2017), by Barták et al. (2020), and by 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; Lin and Bercher 2022), 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 a special treatment for the TO setting (Barták et al. 2021b). Notably, their treatment for TOHTN planning problems did not leverage the fact that any TOHTN planning problem can be transformed into a simplified form (e.g., Chomsky Normal Form (Chomsky 1959)) which can make the search happening in verifying a plan more systematic, and hence, their approach has several overheads. Unfortunately, those overheads are 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 adaptation 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 \( P \) is a tuple \( (D, \text{tn}_I, s_I) \) where \( D = (F, N_c, N_p, M, \delta) \) is called the domain of \( P \). \( F \) 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 with \( m \in 2^F \times N_c \times (N_c \cup N_p)^* \), and \( \delta : N_p \rightarrow 2^F \times 2^F \times 2^F \) is a function. \( s_I \in 2^F \) and \( \text{tn}_I \in (N_c \cup N_p)^* \) are called the initial state and the initial task network (or the goal task network) of \( P \), respectively.

We also define a \( \text{tn} \in (N_c \cup N_p)^* \) 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 \( \delta \). The written \( \text{tn} \) is decomposed into \( m \) subtasks, \( \delta(\text{tn}) \) respectively refer to the preconditions, add list, and delete list of \( m \), each of which is a set of propositions. A primitive task name \( p \) is applicable in a state \( s \in 2^F \), iff \( \text{prec}(p) \subseteq s \), and we say that a state \( s \) is a consequence of applying a primitive task \( p \) in a state \( s \), written \( s \models p \), iff \( p \) is applicable in \( s \), and \( s' = (s \setminus \text{del}(p)) \cup \text{add}(p) \). Similarly, a state trajectory \( (s_0 \cdots s_n) \) is a consequence of applying a sequence of primitive task names \( \text{tn} = \langle p_1 \cdots p_n \rangle \) with \( n \in \mathbb{N}_0 \) i.e., a primitive task network, in a state \( s \) iff \( s_0 = s \), and for each \( 1 \leq i \leq n \), \( s_{i-1} \models p_i \), \( s_i \), and we say that the state \( s_n \) is obtained by applying \( \text{tn} \) in \( s \), written \( s \models^*_\text{tn} s_n \).

On the other hand, a compound task \( c \) in a task network could be rewritten as another task network \( \text{tn} \) by a method \( m = (\text{prec}(m), c, \text{tn}) \) where \( \text{prec}(m) \) refers to the precondition of \( m \). We call this process the decomposition of \( c \), written \( c \rightarrow_m \text{tn} \). We will also omit the subscript \( m \) in the notation, i.e., \( c \rightarrow \text{tn} \), to indicate that there exists some method which decomposes \( c \) into \( \text{tn} \). \( m \) 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 \( \text{tn} \) and \( \text{tn}' \) be two task networks where \( \text{tn} \) is of the form \( \text{tn} = \langle \text{tn}_1, c, \text{tn}_2 \rangle \) with \( c \) being a compound task and \( \text{tn}_1 \) and \( \text{tn}_2 \) being two sequences of task names, each of which might be empty, and \( m = (\text{prec}(m), c, \text{tn}) \) be a method. We say that \( \text{tn} \) is decomposed into \( \text{tn}' \) by \( m \), written \( \text{tn} \rightarrow_m \text{tn}' \), if \( \text{tn}' = \langle \text{tn}_1, c, \text{tn}_2 \rangle \). Similarly, we write \( \text{tn} \rightarrow \text{tn}' \) to indicate that there exists some method which decomposes \( \text{tn} \) into \( \text{tn}' \). We also write \( \text{tn} \rightarrow^*_m \text{tn} \) if \( \text{tn} \) is decomposed into \( \text{tn}' \) by a sequence \( m \) of methods and \( \text{tn} \rightarrow^*_m \text{tn} \) if there exists such a method sequence.

For any two task networks \( \text{tn} \) and \( \text{tn}' \) with \( \text{tn} \rightarrow^* \text{tn}' \), a compound task \( c \) in \( \text{tn} \) is eventually decomposed into a continuous subsequence \( \text{tn} \) in \( \text{tn}' \) (Barták et al. 2021b). Hence, we abuse the notation to let \( c \rightarrow^* \text{tn} \) denote that the compound task \( c \) in some task network is decomposed into the continuous subsequence \( \text{tn} \) of another task network by a sequence of methods, and we write \( c \rightarrow^* \text{tn} \) if such a method sequence \( \text{tn} \) 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 \( P \), a decomposition tree \( g = (V, E, \prec_g, \alpha_g, \beta_g) \) with respect to \( P \) is a labeled directed tree where \( V \) and \( E \) are the sets of vertices and edges, respectively, \( \prec_g \) is a total order defined over \( V \), \( \alpha_g : V \rightarrow N_p \cup N_c \) labels a vertex with a task name, and \( \beta_g \) maps a vertex \( v \in V \) to a method \( m \in M \). Particularly, \( g \) is valid if it is rooted at a vertex \( r \) with \( \alpha_g(r) = c \), 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 \( m = (\text{prec}(m), c, \text{tn}) \) and \( \text{tn} = (\alpha_g(v_1) \cdots \alpha_g(v_n)) \).

Let \( \langle l_1 \cdots l_n \rangle \) (\( n \in \mathbb{N} \)) be the leaves of \( g \) ordered in \( \prec_g \). We define the yield of \( g \), written \( \text{yield}(g) \), as the task network \( (\alpha_g(l_1) \cdots \alpha_g(l_n)) \). For convenience, we will simply use \( L(g) \) to refer to the leaves 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 \( P \) be a TOHTN planning problem, \( g \) a valid decomposition tree \( g \) with respect to \( P \) where \( L(g) = \langle l_1 \cdots l_n \rangle \) and \( \text{yield}(g) = (\alpha_g(l_1) \cdots \alpha_g(l_n)) \) (\( n \in \mathbb{N} \)) consists solely of primitive tasks, and \( m = (\text{prec}(m), c, \text{tn}) \) a method with \( \beta_g(v) = m \) for some inner vertex \( v \in V \). The precondition of \( m \) is satisfied if and only if for the first vertex \( l_i \) (\( 1 \leq i \leq n \)) in \( L(g) \) that is a descendant of \( v \), it holds that \( \text{prec}(m) \subseteq s_{i-1} \) with \( s_i \rightarrow^*_m s_{i-1} \) and \( \text{tn} = (\alpha_g(l_1) \cdots \alpha_g(l_{i-1})) \). 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 \( P \), a solution to \( P \) is a task network \( \text{tn} \) consisting solely of primitive tasks such that \( \text{tn} \) is executable in \( s_I \), i.e., \( s_I \rightarrow^*_m \text{tn} \) for some \( s \in 2^F \), and there exists a valid decomposition tree \( g \) with respect to \( P \) such that \( \text{yield}(g) = \text{tn} \) 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 analogous to terminal symbols,
non-terminal symbols, and production rules in CFGs. Consequently, the TOHTN plan verification problem is analogous 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 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 \( \pi_i = \langle p_i \cdots p_j \rangle \) (1 \( \leq i \leq j \leq n \)), the set \( A[i, j] \) of all possible non-terminal symbols \( c \) such that \( c \rightarrow^* \pi_i \), i.e., \( c \) can be decomposed into \( \pi_i \) by a sequence of production rules (methods). Mathematically, this goal can be accomplished via the following recursion formula:

\[
A[i, j] = \left\{ \begin{array}{ll}
\{ c | c \rightarrow \langle p_i \rangle \} & \text{if } i = j \\
\{ c | c \rightarrow (c'_1 \cdot c'_2), i \leq k < j \} & \text{if } i < j \\
\{ c | c \rightarrow (c'_1 \cdot c'_2), i \leq k < j \} & \text{if } i > j
\end{array} \right.
\]

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 \( i, j \) such that 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 \( i \leq k < j \), \( c'_1 \rightarrow \pi_k \), and \( c'_2 \rightarrow \pi_{k+1} \). 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 memorise 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 (where line 8 and line 11 – 13 are the extra steps for TOHTN plan verification for which we will have more detailed discussions later on). In Alg. 1, each component in CFGs (i.e., terminal/non-terminal symbols, production rules, etc.) is substituted with its counterpart in TOHTN planning problems.

In order to adapt the CYK algorithm to 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.

Notably, 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 \( c \rightarrow^m \pi_i \) for some \( c \in N_c, i = \{i \cdots i\} \), \( c' \in A[i, k], \) and \( c'' \in A[k + 1, j] \). 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 \( \varphi_1 : N_p \rightarrow M \) and \( \varphi_2 : N \times N \rightarrow M \) 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 \( p \), and similarly, given \( t_1, t_2 \in N, \varphi_2(t_1, t_2) = m \) iff \( m \) decomposes a compound task into \( t_1 \cdot t_2 \). 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).

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. Similar to how the transformation from a CFG to CNF is done (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. E.g., consider a sequence of unit productions $c_1 \to \cdots \to c_n$, where $c_1, \ldots, c_n$ are compound tasks. Further, there exist $k$ methods $m_1, \ldots, m_k$, with $m_i = (c_{a_i}, \langle a_i, a_i' \rangle)$ for each $1 \leq i \leq k$ where $a_i$ and $a_i'$ are two actions. For the purpose of eliminating this sequence of unit productions, for each $c_i$ ($1 \leq i \leq n$), we have to construct additional $k$ methods $m^*_1, \ldots, m^*_k$ with $m^*_i = (c_j, \langle a_i, a_i' \rangle)$ for each $1 \leq i \leq k$. It thus results in a quadratic explosion. Such an explosion is a significant overhead for TOHTN plan verification because usually a planning problem already contains an enormous number of methods.

In order to avoid such an explosion, we only apply the first step binarization to an input TOHTN planning problem and result in the planning problem being in so-called 2-Normal Form (2NF) (Behnke and Speck 2021; Lange and Leiß 2009), i.e., in which every method contains at most two subtasks (could be either primitive or compound). The binarization step works as follows. For a method $(c, \langle t_1 \cdot \cdots \cdot t_n \rangle)$ with $n > 2$ and $t_i \in N_0 \cup N_c$, we first construct $n - 1$ compound tasks $c_1, \ldots, c_{n-1}$ for each $1 \leq i \leq n$. Afterwards, we construct the methods $(c, \langle t_1 c_1 \rangle), (c_{n-1}, \langle t_n \rangle)$, and $(c_1, \langle t_1 c_{i+1} \rangle)$ for each $1 \leq i < n - 1$. Clearly, the size of the input problem only increases linearly after this step.

The price for adapting 2NF 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 $c' \rightarrow \ast (c)$ for some $c \in A[i, j]$, and the preconditions of every method occurring in the decomposition process is satisfied. This is equivalent to finding all method sequences $m$ such that the preconditions of each method in it is satisfied, and $c' \rightarrow m (c)$ 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 first want to find all compound tasks $c$ and all method sequences $m$ such that $c \rightarrow m (c)$ where $c$ refers to the empty task network. We call such a $c$ a nullable task which is analogous to a nullable symbol in CFGs. This can be done by adapting the recursive procedure for finding all nullable symbols in a CFG (Hopcroft, Motwani, and Ullman 2007), as shown below:

**Basis:** If $c \rightarrow m (c)$ 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 $c \rightarrow \langle t_1 t_2 \rangle$ (or $c \rightarrow \langle t \rangle$) 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 $m_1$ and $m_2$ that respectively decompose $t_1$ and $t_2$ into the empty task network (or any $\langle m \rangle$ with $t \rightarrow m (c)$), $(m_1 m_2) (c)$ (or $(m m)$) together with any permutation of it is marked as a method sequence that decomposes $c$ into $c$.

Having identified all nullable tasks in a planning problem, we could now find all method sequences $m$ such that $c' \rightarrow m (c)$ for some $c', c \in N_c$. We do so by constructing a graph $G = (V, E)$ such that $V = M$, i.e., the vertices are the methods of the planning problem, and an edge $(m', m) \in E$ with $m' = (\langle m' \rangle, c', t' n')$ and $m = (\langle m \rangle, c, t n)$ iff either $t' = \langle c \rangle$ or $t n' = \langle t_0 t_1 \rangle$ such that there exists an $i \in \{0, 1\}$ with $t_i = c$ and $t_{i-1}$ being a nullable task. We name such a graph as a unit production graph. The concrete procedure for constructing such a graph is as follows: For each method $m \in M$ with $m = (\langle m \rangle, c, t n)$,

- if $t n = \langle t_0 t_1 \rangle$ for some $t_0, t_1 \in N$ ($N = N_0 \cup N_p$), and there exists an $i \in \{0, 1\}$ such that $t_{i-1}$ is nullable, then for each $m'$ that can decompose $t_i$, we add the edge $(m, m')$ to the graph, or
- if $t n = \langle t \rangle$ for some $t \in N_c$, then for each method $m'$ that decomposes $t$, we add the edge $(m, m')$ to the graph.

The core of exploiting a unit production graph to find all method sequences $m$ such that $c' \rightarrow m (c)$ for some $c', c \in N_c$ is the fact that for any two compound tasks $c', c \in N_c$, $c' \rightarrow c$ iff there exists a path in $G$ from $m'$ to $m$ such that $m'$ and $m$ respectively decompose $c'$ and $c$.

**Theorem 1.** Let $c, c' \in N_c, c \rightarrow \ast (c')$. 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.** ($\Rightarrow$): We prove this by induction on the number of steps in decomposing $c$ into $c'$. The base case is $c \rightarrow (c')$. In this case, a path $(m', m)$ with $c'$ being decomposed by $m'$ exists by the construction of the graph $G$.

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

($\Leftarrow$): 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 $t n$ such that either $t n = \langle c' \rangle$ or $t n = \langle t_0 t_1 \rangle$ in which there exists an $i \in \{0, 1\}$ with $t_{i+1} = c'$ and $t_{i-1}$ is nullable. For the former, $c \rightarrow c'$ holds naturally, and for the latter, since $t_{i-1} \rightarrow \ast \langle c \rangle$ (because $t_{i-1}$ is nullable), it follows immediately that $c \rightarrow \ast (c')$.

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 $m$ to $m'$ of length $k - 1$ and an edge $(m, m) \in E$. By the induction hypothesis, there exist $c \in N_c$ with $c$ being decomposed by $m$ such that $c \rightarrow \ast (c')$. Further, by the construction of the graph $G$, the presence of the edge $(m, m)$ implies that $m$ decomposes a compound task $c$ into a task network $t n$ in which either $c$ is the only subtask, or $t n$ contains two subtasks where one is $c$ and the other is nullable. For both cases, we have $c \rightarrow \ast (c')$ and henceforth $c \rightarrow \ast (c')$. 

Consequently, we can find all method sequences $m$ with $c' \rightarrow m (c)$ for some $c', c \in N_c$ by doing several depth-first search in the reverse graph of the unit production graph $G$. 

12044
(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 $\pi = (m_1 \cdots m_k)$, 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 $\pi'$
such that $t \rightarrow^* \epsilon$ 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 prob-
lem in 2RF. We first implement the standard CYK algorithm
for computing each table entry $A[i, j]$, and then for each
such entry $A[i, j]$, we find all method sequences $\pi$ such that
$c' \rightarrow^*_{\pi} (c)$ for some $c' \in N_c$ and $c \in A[i, j]$ and check
whether all method preconditions in the sequence are satis-
fied. If so, we then add $c' \rightarrow A[i, j]$.

Lastly, we would like to discuss the time complexity of
our plan verification approach. For an input plan $(p_1 \cdots p_m)$,
one can easily recognize that the time required for visiting
all entries $A[i, j]$ ($1 \leq i \leq j \leq n$) is $O(n^3)$. 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 $c \rightarrow^* (c')$ and
$c' \in A[i, j]$. Therefore, the time complexity of the CYK-
based plan verification approach is $O(|M| \times n^3)$.

Theorem 2. Alg. 1 has the time complexity $O(|M| \times n^3)$
with $n$ being the length of the input plan.

Note that the time complexity of the CYK-based approach
also emphasizes the importance of maintaining the mappings
$\varphi_1$ and $\varphi_2$ mentioned earlier because $|M|$ 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 $O(n^4)$.

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
benchmark set which have method preconditions and on the
one which does not. The benchmark set with method pre-
conditions 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 plan-
ning 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 prob-
lems. This set again contains 12367 instances where 11304
are valid, and 1063 are invalid (note the increasing number
of valid plans 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
into a SAT problem and an HTN planning problem. All the
experimental results produced by our approach are publicly
available (Lin et al. 2023).

In the experiments run on the benchmark sets 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 achieved the best
performance and solved 10925 instances (99.67%). Our
approach slightly underperformed it and solved 10917 in-
stances (99.60%). Our approach beat the parsing-based one
which solved 9158 instances (83.55%). For the invalid
instances, our approach solved all 1406 instances (100%) com-
pared with the planning-based one and the parsing-
based one which solved 1364 instances (97.01%) and 1301
instances (92.53%), respectively. The results are summa-
rized in Tab. 1 where the rows to-val and to-inval respec-
tively indicate the valid and invalid instances.

Notably, although our approach slightly underperformed
the planning-based one in general in verifying valid plans
with method preconditions, it was significantly faster than
the planning-based one in many domains. The runtime in
those domains is depicted in Fig. 1. The plots show the
time (y-axis) against the percentage of instances solved
(x-axis) by our approach and the planning-based approach,
respectively. For the remaining domains, the runtime for
these two approaches is similar. The runtime against the
total number of solved instance for our approach and the
planning-based approach in the various experimental con-
figurations is shown in Fig. 2.

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 989 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.

Fig. 2 depicts the number of solved instances against the
runtime for the evaluations on all the benchmark sets. One
might observe that in solving the instances with method pre-
conditions, our approach has the similar performance com-
pared with the planning based one and outperforms the pars-
based one. For those without method preconditions, our
approach clearly beats the others.

Discussion

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

According to the experiment results, our approach out-
performs the parsing-based one (Barták et al. 2020, 2021b)
which is the only one by now having a special treatment
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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Instances</th>
<th>Parsing-based</th>
<th>Planning-based</th>
<th>SAT-based</th>
<th>CYK-based (Ours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>to-val</td>
<td>10961</td>
<td>9158 (83.55)</td>
<td>10925 (99.67)</td>
<td>no support</td>
<td>10917 (99.60)</td>
</tr>
<tr>
<td>to-inval</td>
<td>1406</td>
<td>1301 (92.53)</td>
<td>1364 (97.01)</td>
<td>no support</td>
<td>1406 (100.00)</td>
</tr>
<tr>
<td>to-val-no-mprec</td>
<td>11304</td>
<td>7889 (69.79)</td>
<td>9679 (85.62)</td>
<td>1036 (9.16)</td>
<td>9946 (87.99)</td>
</tr>
<tr>
<td>to-inval-no-mprec</td>
<td>1063</td>
<td>915 (86.08)</td>
<td>898 (84.48)</td>
<td>684 (64.35)</td>
<td>981 (92.29)</td>
</tr>
</tbody>
</table>

Figure 1: The runtime for the domains where the CYK-based approach significantly outperforms the planning-based approach.

Future Work

The TO plan verification approach presented in the paper is based on the CYK parsing algorithm, which belongs to the family of so-called chart parsing algorithms (see the work by Jurafsky and Martin (2000) for more details about chart parsing algorithms). It is thus natural to think of adapting from the method. Despite that the parsing-based approach supports such an additional constraint, the benchmark sets on which we ran the empirical evaluation do not feature it, and hence, this extra functionality will not incur overheads to the approach in the experiments.

For the planning-based approach (Hölker et al. 2022), it outperforms our approach in verifying valid plan instances with method preconditions and underperforms ours in the remaining three cases. We hypothesize that this is because heuristics used in a TOHTN planner can significantly benefit searching for a solution when an input problem is solvable (which is the case for verifying valid plans), but they are less powerful when the input problem is unsolvable (which is the case for verifying invalid plans). Generally speaking, we argue that our CYK-based approach as a decision procedure is still better than the planning-based one.

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 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. Thus, 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 unsystematic search. For example, in our CYK-based approach, in order to decide whether a compound task \( c \) can be decomposed into a subsequence \( \pi_{i}^{1} = (p_{i} \cdots p_{j}) \) via a method \( m = (\text{prec}(m), c, (c'_{1} c'_{2})) \), 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 \( \pi_{i}^{1} \) via a method which has \( k \) \((k \in \mathbb{N})\) subtasks \( (c'_{1} \cdots c'_{k}) \) requires deciding whether \( \pi_{j}^{1} \) can be divided into \( k \) subsequences \( \pi_{r}^{1} = (\pi'_{r} \cdots \pi'_{k}) \) such that \( c'_{r} \rightarrow^{*} \pi'_{r} \) for each \( 1 \leq r \leq k \). The latter one is clearly more computationally expensive.
some more sophisticated charting parsing algorithms like the Earley parsing algorithm (Earley 1970) to develop a more efficient plan verification approach for TOHTN planning.

Further, although our paper focus solely on TOHTN plan verification, some of our ideas might be exploited and combined with some other parsing based plan verification approaches, for example, the one by Bartáková, Maillard, and Cardoso (2018) and by Bartáková et al. (2020), which work for general HTN planning or even an HTN planning formalism supporting advanced features, e.g., prevail conditions. Specifically, when using those approaches, we could consider some preprocessing for an input plan verification problem to turn it into a more digestible form to make the parsing more systematic, just like how we transform a TOHTN planning problem into 2NF in our paper.

**Conclusion**

In this paper, we developed a totally ordered HTN plan verification approach that is 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 Bartáková 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.

**Acknowledgments**

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

Roman Barták is supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215.

Further, we would like to thank the anonymous reviewers from the 2022 International Symposium on Combinatorial Search (to which this paper was submitted before) who provided many helpful comments that helped to improve the paper significantly.

**References**


