Accelerating SAT-Based HTN Plan Verification by Exploiting Data Structures from HTN Planning

Songtuan Lin[1], **Gregor Behnke**[2] **and Pascal Bercher**[1;*]

[1]School of Computing, The Australian National University
[2]Institute for Logic, Language and Computation, University of Amsterdam

Abstract. Plan verification is the task of deciding whether a given plan is a solution to a planning problem. In this paper, we study the plan verification problem in the context of Hierarchical Task Network (HTN) planning, which has been proved to be $\mathbb{NP}$-complete when *partial order* (PO) is involved. We will develop a novel SAT-based approach exploiting the data structures *solution order graphs* and *path decomposition trees* which encodes an HTN plan verification problem as a SAT one. We show in our experiments that this new approach outperforms the current state-of-the-art (SOTA) planning-based approach for verifying plans for POHTN problems.

1 Introduction

The plan verification problem is the task of deciding whether a given plan is a solution to a planning problem. Research on this problem has drawn increasing attention in the last few years due to its potential usages in numerous applications, e.g., in mixed initiative planning [6, 22], in validating planning domains [19, 21, 20] where failed plan verification indicates flaws in planning domains, and in International Planning Competition. Recently, there were also attempts to exploit plan verification to solve planning problems [14]. Plan verification is an easy task in classical planning, whereas it is computationally expensive in the hierarchical setting.

In this paper, we consider the plan verification problem in the context of Hierarchical Task Network (HTN) planning [12, 10], which is a hierarchical approach to planning where so-called compound tasks are kept being decomposed until primitive actions are obtained. Plan verification is poly-time decidable for a restricted case of HTN planning called *total order* (TO) HTN planning where every compound task is decomposed into a *sequence* of subtasks and is $\mathbb{NP}$-complete [5, 11] in the general case where *partial order* (PO) is involved, i.e., a compound task is decomposed into a partial order set of subtasks. Approaches targeted specifically at verifying plans for TOHTN problems are well-developed [3, 18]. To enhance the efficiency of plan verification in the *partial order* setting, we propose a novel approach based on SAT that surpasses the state-of-the-art (SOTA) planning-based technique [17] and hence outperforms all other existing approaches, i.e., the parsing-based one [4, 2] and the existing SAT-based one [6], in verifying plans for POHTN problems.

Our approach exploits two data structures, *solution order graphs* and *path decomposition trees*, from the SAT-based HTN planner [7, 8], and our approach supports method preconditions, which are not supported by the existing SAT-based verifier.

* Corresponding Author. Email: pascal.bercher@anu.edu.au

2 Preliminaries

To provide some context, we first present the notion of SAT and HTN planning in this section.

SAT Generally speaking, the *boolean satisfiability problem* (SAT) is to decide whether a boolean formula φ defined over a set of propositional variable $X = \{x_1, \cdots, x_n\}$ is satisfiable. More concretely, a boolean formula over a variable set X is defined inductively as follows. 1) For every $x \in X$, x is a boolean formula. 2) If φ is a boolean formula, then $\neg\varphi$ is also a boolean formula. 3) If φ_1 and φ_2 are two boolean formulae, then $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, and $\varphi_1 \rightarrow \varphi_2$ are all boolean formulae. In particular, the symbols $\neg$, $\wedge$, $\vee$, and $\rightarrow$ are called logical connectors.

A boolean formula φ over a set X of variables is satisfiable if and only if there exists a *truth assignment* over X such that φ is evaluated to *true* under this assignment. A truth assignment on X is such that for every $x \in X$, we assign either *true* or *false* to it. The evaluation for a boolean formula φ over X can again be defined inductively as follows: 1) For every variable $x \in X$, the formula $\varphi = x$ is evaluated to the value assigned to x. 2) If φ is a boolean formula, then $\neg\varphi$ is evaluated to *true* if φ is evaluated to *false*, otherwise, it is evaluated to *false*. 3) If φ_1 and φ_2 are two formulae, then $\varphi_1 \wedge \varphi_2$ is evaluated to *true* if *both* φ_1 and φ_2 are evaluated to *true*, otherwise, it is evaluated to *false*. Contrastively, $\varphi_1 \vee \varphi_2$ is evaluated to *true* if *one* of φ_1 and φ_2 is evaluated to *true*, otherwise, it is evaluated to *false*. 4) For any two formulae φ_1 and φ_2, we have that $\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$. In other words, we could interpret $\varphi_1 \rightarrow \varphi_2$ as if φ_1 is evaluated to *true*, then φ_2 must also be evaluated to *true*.

HTN Planning We then present the HTN planning formalism [10] on which our paper is based. An HTN planning problem Π is a tuple $(\mathcal{D}, s_I, c_I)$ where $\mathcal{D} = (\mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{M}, \alpha)$ is called the domain of Π. $\mathcal{P}$ is a set of *propositions*, $\mathcal{A}$ a set of *primitive* tasks (also called *actions*), $\mathcal{C}$ a set of *compound* tasks, $\mathcal{M}$ a set of *methods*, and $\alpha : \mathcal{A} \rightarrow 2^{\mathcal{P}} \times 2^{\mathcal{P}} \times 2^{\mathcal{P}}$ a function. $s_I \in 2^{\mathcal{P}}$ and $c_I \in \mathcal{C}$ are called the initial *state* and the initial task of Π, respectively.

More concretely, an action $a \in \mathcal{A}$ is characterized by its *precondition*, *positive effects*, and *negative effects*, which are defined by the function α, written $\alpha(a) = (\texttt{prec}(a), \texttt{eff}^+(a), \texttt{eff}^-(a))$. An action a is *applicable* in a *state* $s \in 2^{\mathcal{P}}$ if $\texttt{prec}(a) \subseteq s$, and applying a in s will lead to a new state s' with $s' = (s \setminus \texttt{eff}^-(a)) \cup \texttt{eff}^+(a)$, written $s \rightarrow_a s'$. Given an action sequence $\pi = \langle a_1 \cdots a_n \rangle$ and two states s and s', we use $s \rightarrow_\pi^* s'$ to indicate that s' is obtained by

applying π in s, that is, there exists a state sequence $\langle s_0 \cdots s_n \rangle$ with $s_0 = s$, $s_n = s'$, and for each $1 \leq i \leq n$, a_i is applicable in s_{i-1} and $s_{i-1} \rightarrow_{a_i} s_i$.

In HTN planning, primitive tasks can only be obtained by decomposing compound tasks by using methods. Formally speaking, a compound task $c \in \mathcal{C}$ is decomposed into a *task network* tn by a method $m = (\texttt{prec}(m), c, tn) \in \mathcal{M}$ where $\texttt{prec}(m) \in 2^{\mathcal{P}}$ is the precondition of m. A method can only be applied to decompose a compound task if its precondition is satisfied. We will discuss more details about the semantics of method preconditions later on. A task network tn, which is essentially a partial order *multiset* of compound and primitive tasks, is a tuple $(\mathcal{T}, \prec, \gamma)$ with $\mathcal{T}$ being a set of identifiers, $\prec \subseteq \mathcal{T} \times \mathcal{T}$ a partial order defined over $\mathcal{T}$, and $\gamma : \mathcal{T} \rightarrow \mathcal{A} \cup \mathcal{C}$ a function mapping each identifier to a task. Furthermore, two task networks $tn = (\mathcal{T}, \prec, \alpha)$ and $tn' = (\mathcal{T}', \prec', \alpha')$ are isomorphic, written $tn \cong tn'$, *iff* there exists a *bijective* mapping $\varphi : \mathcal{T} \rightarrow \mathcal{T}'$ such that for all $t \in \mathcal{T}$, $\gamma(t) = \gamma'(\varphi(t))$, and for all $t_1, t_2 \in \mathcal{T}$, $(t_1, t_2) \in \prec$ *iff* $(\varphi(t_1), \varphi(t_2)) \in \prec'$. A linearization of a task network $tn = (\mathcal{T}, \prec, \gamma)$ is a total order of $\mathcal{T}$ that respects the partial order $\prec$. For convenience, we define the following operation.

Definition 1 *Let D and V be two arbitrary sets, $R \subseteq D \times D$ be a relation, $f : D \rightarrow V$ be a function and $tn = (\mathcal{T}, \prec, \gamma)$ be a task network. The restrictions of R and f to some set X are defined by*

- $R|_X = R \cap (X \times X)$
- $f|_X = f \cap (X \times V)$
- $tn|_X = (\mathcal{T} \cap X, \prec|_X, \gamma|_X)$

One can easily extend the notion of decomposing a compound task to decomposing a task network tn. Let $tn = (\mathcal{T}, \prec, \gamma)$ be a task network, $t \in \mathcal{T}$ an identifier, c a compound task with $\gamma(t) = c$, and $m = (c, tn_1^*)$ a method. We say m decomposes tn into another task network $tn' = (\mathcal{T}', \prec', \gamma')$, written $tn \rightarrow_m tn'$, *iff* there exists a task network $tn_2^* = (\mathcal{T}^*, \prec^*, \gamma^*)$ with $tn_1^* \cong tn_2^*$ such that

- $\mathcal{T}' = (\mathcal{T} \setminus \{t\}) \cup \mathcal{T}^*$.
- $\prec' = (\prec \cup \prec^* \cup \prec_X)|_{\mathcal{T}'}$ with $\prec_X = \{(t_1, t_2) \,|\, (t_1, t) \in \prec, t_2 \in \mathcal{T}^*\} \cup \{(t_2, t_1) \,|\, (t, t_1) \in \prec, t_2 \in \mathcal{T}^*\}$.
- $\gamma' = (\gamma \setminus \{(t, c)\}) \cup \gamma^*$.

Additionally, a task network tn is decomposed into another one tn' by a sequence of methods $\overline{m} = \langle m_1 \cdots m_n \rangle$ with $n \in \mathbb{N}_0$ (where $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$), written $tn \rightarrow_{\overline{m}}^* tn'$, *iff* there exists a sequence of task networks $\langle tn_0 \cdots tn_n \rangle$ such that $tn_0 = tn$, $tn_n = tn'$, and for each $1 \leq i \leq n$, $tn_{i-1} \rightarrow_{m_i} tn_i$. Similarly, we use the same notation $c \rightarrow_{\overline{m}}^* tn$ to denote that a compound task c is decomposed into a task network tn by a sequence of methods $\overline{m}$.

The process of decomposing a task network (or a compound task) by a sequence of methods can be captured more precisely by a *decomposition tree* [13]. A decomposition tree $g = (\mathcal{V}, \mathcal{E}, \prec_g, \alpha_g, \beta_g)$ with respect to an HTN planning problem Π is a labeled directed tree where $\mathcal{V}$ and $\mathcal{E}$ are the sets of vertices and edges, respectively, $\prec_g$ is a partial order defined over $\mathcal{V}$, $\alpha_g : \mathcal{V} \rightarrow \mathcal{A} \cup \mathcal{C}$ labels a vertex with a task, and β_g maps a vertex $v \in \mathcal{V}$ to a method $m \in \mathcal{M}$.

A decomposition tree with respect to Π is *valid iff* there exists a root vertex $r \in \mathcal{V}$ that is labeled with c_I, and for each $v \in \mathcal{V}$ with $\beta_g(v) = m$, $m = (\texttt{prec}(m), c, tn)$, and $c \in \mathcal{C}$, the following holds: 1) $\alpha_g(v) = c$, 2) tn is isomorphic to the task network induced by the children of v denoted as $\mathbb{C}(v)$, i.e., $tn \cong (\mathbb{C}(v), \prec_g^v, \alpha_g^v)$ where $\prec_g^v = \prec_g|_{\mathbb{C}(v)}$ and $\alpha_g^v = \alpha_g|_{\mathbb{C}(v)}$, 3) for any child v_c of v and any $v' \in \mathcal{V}$, if $(v', v) \in \prec_g$, $(v', v_c) \in \prec_g$, and $(v, v') \in \prec_g$, then $(v_c, v') \in \prec_g$, and 4) there are no other ordering constraints in $\prec_g$ except those demanded by 2) and 3).

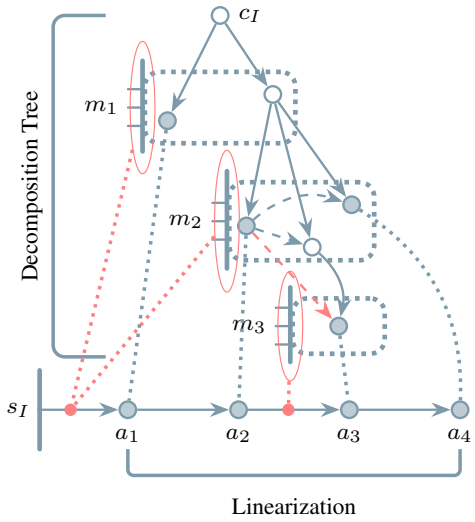

Figure 1. An example of a decomposition tree and method preconditions.

The *yield* of a decomposition tree g, written $yield(g)$, is the task network $(\mathcal{L}(g), \prec_g|_{\mathcal{L}(g)}, \alpha_g|_{\mathcal{L}(g)})$ where $\mathcal{L}(g)$ denotes the set of all *leafs* of the decomposition tree (a leaf is a vertex which has no children). Intuitively speaking, the yield of a tree captures the task network obtained by decomposing the initial task. Further, for the purpose of simplicity, for every vertex $v \in \mathcal{V}$, we define $\mathcal{L}(v)$ as the set of the leafs of g which are also the descendants of v. As a special case, $\mathcal{L}(v) = \emptyset$ if $v \in \mathcal{L}(g)$.

Having defined decomposition trees, we now formalize the semantics of method preconditions, which is defined in terms of a decomposition tree whose yield consists solely of primitive actions and a linearization of its yield. Intuitively, a method's precondition in a decomposition tree is satisfied if it is satisfied *somewhere* before where it is applied, i.e., it is satisfied somewhere before the *first* action in the linearization of the yield that is obtained by using the method.

Definition 2 *Let Π be a planning problem, $g = (\mathcal{V}, \mathcal{E}, \prec_g, \alpha_g, \beta_g)$ a* valid *decomposition tree with respect to Π where $yield(g) = tn$ with $tn = (\mathcal{T}, \prec, \gamma)$ consisting solely of primitive tasks, $\overline{tn} = \langle t_1 \cdots t_n \rangle$ ($n \in \mathbb{N}$ and $n = |\mathcal{T}|$) a linearization of $yield(g)$, and $v \in \mathcal{V} \setminus \mathcal{L}(g)$ an inner vertex of g with $\beta_g(v) = m$ for some method $m \in \mathcal{M}$. The precondition $\texttt{prec}(m)$ of the method m is satisfied if and only if for the first task t_j ($1 \leq j \leq n$) that occurs in $\overline{tn}$ with $t_j \in \mathcal{L}(v)$ (i.e., for all $1 \leq k < j$, $t_k \notin \mathcal{L}(v)$), there exists an i with $1 \leq i \leq j$ such that for all k with $i \leq k < j$, $(t_k, t_j) \notin \prec$, and $\texttt{prec}(m) \subseteq s_{i-1}$ where $s_I \rightarrow_{\pi_{i-1}}^* s_{i-1}$ and $\pi_{i-1} = \langle \gamma(t_1) \cdots \gamma(t_{i-1}) \rangle$. In particular, for the case where $i = 1$, we define $s_0 = s_I$.*

Example for HTN Planning Fig. 1 depicts an example of HTN planning together with a decomposition tree and method preconditions. Each white circle represents a compound task, and a blue one represents a primitive task. These primitive and compound tasks constitute the vertices of the decomposition tree, and the solid arrows are the edges of the tree. The dotted boxes depict methods. The precondition of each method is highlighted by a red circle. The dashed arrows are the partial ordered defined over the vertices among which those blue ones are the ordering constraints defined initially by the method, and the red one is the consequence of the decomposition (i.e., it is in-

herited from the constraint in m_2). The decomposition starts with the initial task c_I, which is decomposed by m_1 into a primitive task and a compound one. The resulting compound task is further decomposed by m_2 into two primitive ones and one compound one which is again decomposed into another primitive task by m_3.

The dotted lines in Fig. 1 (including those colored in red) depict how the leafs and method preconditions are arranged to form the linearization. We want to particularly emphasize the arrangement of the method preconditions. Note that every method precondition must be satisfied *before* the first primitive action in the linearization that is derived from the respective method. Further, $\texttt{prec}(m_2)$ is placed before a_1, i.e., *somewhere before* a_2, and $\texttt{prec}(m_3)$ however *cannot* be placed anywhere before a_2, because a_2 is ordered before a_3 in the yield of the decomposition tree.

Definition 3 *A solution to an HTN planning problem is an action sequence $\pi = \langle a_1 \cdots a_n \rangle$ satisfying the following criteria: 1) π is executable in s_I, i.e., $s_I \rightarrow_\pi^* s$ for some state s, 2) there exists a valid decomposition tree g with respect to Π with $yield(g) = (\mathcal{T}, \prec, \gamma)$ such that $yield(g)$ possesses a linearization $\overline{tn} = \langle t_1 \cdots t_n \rangle$ ($n = |\mathcal{T}|$ and for each $1 \leq i \leq n$, $t_i \in \mathcal{T}$) such that $\pi = \langle \gamma(t_1) \cdots \gamma(t_n) \rangle$, and 3) the precondition of every method in g is satisfied with respect to $\overline{tn}$.*

Notably, the presented definition of solutions is different from the one used in standard HTN literatures [12, 10]. The latter one requires a solution to be a primitive task network, but here, we demand that a solution should be an action sequence. We argue that our definition is more practically coherent. For instance, solutions submitted to IPC are action sequences.

A method's precondition is usually compiled as an action which has the same precondition as the method and which is placed *before* all subtasks of the method. Given an HTN planning problem Π, for every method $m = (\texttt{prec}(m), c, tn)$ with $tn = (\mathcal{T}, \prec, \gamma)$, we transform it into a new one $m^* = (\mathcal{T}^*, \prec^*, \gamma^*)$ where $\mathcal{T}^* = \mathcal{T} \cup \{t^*\}$ with t^* being a new task identifier, $\prec^* = \prec \cup \{(t^*, t) \mid t \in \mathcal{T}\}$, and $\gamma^* = \gamma \cup \{(t^*, a^*)\}$ with $\texttt{prec}(a^*) = \texttt{prec}(m)$ and $\texttt{eff}^+(a^*) = \texttt{eff}^-(a^*) = \emptyset$. Such a compilation is employed by most HTN planners [7, 8, 16] as it allows method preconditions to be processed more easily. We use $\mathbf{C}(\Pi)$ to denote the problem obtained from an HTN problem Π by compiling its method preconditions away. One can easily observe the connection between solutions to Π and $\mathbf{C}(\Pi)$.

Proposition 1 *Given a plan π and an HTN planning problem Π, π is a solution to Π* iff *there exists a valid decomposition tree g with respect to $\mathbf{C}(\Pi)$ such that π is a* subsequence *of an executable linearization of $yield(g)$, and all the remaining actions in the linearization are a compiled method precondition.*

This compilation also benefits our SAT-based HTN plan verifier. Thus, for convenience, we will assume that the method preconditions of every HTN planning problem discussed in the remainder of this paper have been compiled away, unless otherwise specified. Further, for any HTN planning problem, we use $\mathcal{A}^* \subseteq \mathcal{A}$ to refer to the set of all artificial primitive tasks representing a method precondition.

3 Plan Verification

Having introduced the HTN planning formalism, we now move on to present our SAT-based approach for verifying plans for HTN problems. The entire procedure is summarized in Alg. 1, which is to enumerate all possible valid decomposition trees to check whether there exists one that results in the given plan. This is done in an iterative way. On each iteration, we craft a SAT formula which is satisfiable *iff* there exists a valid decomposition tree of height up to a number K (K is set to one initially) which results in the given plan, and all decomposition trees of height up to K are stored in a *path decomposition tree* [7, 8] (line 4 and 5), which we will discuss in more details later on. If the SAT formula is *not* satisfiable, we increase K by one. The procedure stops either when we obtain a satisfiable SAT formula (in which case the plan is a solution) or when K exceeds the maximal height of decomposition trees (line 2) that can lead to the given plan (in which case the plan is not a solution). For the latter case, we will briefly introduce some existing approaches for calculating the bound after we have presented our SAT encoding.

Algorithm 1 HTN Plan Verification by SAT.

Input: An HTN problem Π
A plan π
Output: True if π is a solution to Π, otherwise, false

1: $K \leftarrow 1$ ▷ Height of the PDT
2: $K^* \leftarrow \textsc{MaxHeight}(\Pi, \pi)$
3: **for** $K \leq K^*$ **do**
4: $\Phi_K \leftarrow$ The PDT for Π of height K
5: $\varphi \leftarrow$ SAT formula for Φ_K
6: **if** φ is satisfiable **then**
7: **return** True
8: $K \leftarrow K + 1$
9: **return** False

PDTs and SOGs For constructing the SAT formula given a bound K, we leverage the data structures *path decomposition trees* (PDTs) and *solution order graphs* (SOGs) used in the SAT-based HTN planner [7, 8]. Those two together provide a way to store compactly *all* possible valid decomposition trees up to a certain height.

More concretely, given an HTN planning problem Π and a number K, a PDT of height K, denoted as Φ_K, is a tuple (V, E, ϱ) in which (V, E) is a tree of height K with a root vertex r, and $\varrho : V \rightarrow 2^{\mathcal{A} \cup \mathcal{C}}$ is a function mapping each vertex to a set of tasks with $\varrho(r) = \{c_I\}$. Additionally, for every *inner* vertex v in V and every *compound* task c in $\varrho(v)$, there exists a subset $S_m^v = \{v_1, \cdots, v_n\}$ of v's children for *every* method $m = (\texttt{prec}(m), c, tn)$ with $tn = (\mathcal{T}, \prec, \gamma)$ and $|\mathcal{T}| = n$ such that there exists a bijective mapping $\varphi_m^v : S_m^v \rightarrow \mathcal{T}$ with $\gamma(\varphi_m^v(v_i)) \in \varrho(v_i)$ for all $v_i \in S_m^v$. In particular, φ_m^v is called a child arrangement function of v.

Intuitively speaking, a PDT Φ_K captures all possible valid decomposition tree up to the height K, disregarding all partial order in those trees. Fig. 2 shows an example of a PDT, which contains two decomposition trees, colored in red and green, respectively. The left-most branch of the PDT is shared by both two decomposition trees. Further, note that for the vertex v in the example, $\varrho(v)$ is mapped to two tasks each of which corresponds to a decomposition tree.

The ordering constraints on the decomposition trees contained by a PDT are imposed separately by arranging the vertices of a PDT in an order-consistent way. Intuitively speaking, we want to arrange the vertices of a PDT such that for any two vertices v_1, v_2, there is an ordering constraint (v_1, v_2) *iff* v_1 and v_2 serve as two vertices in a decomposition tree in the PDT where v_1 is ordered before v_2.

Formally, we first define the binary relation $\precsim_v$ over the children of a vertex v induced by v's child arrangement functions. That is, for any $v_1, v_2 \in \mathbb{C}(v)$, $(v_1, v_2) \in \precsim_v$ *iff* there exist a compound task $c \in \varrho(v)$ and a method $m = (\texttt{prec}(m), c, tn)$ such that

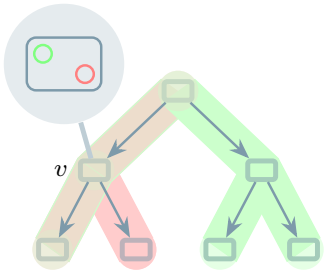

Figure 2. An example of a path decomposition tree.

$(\varphi_m^v(v_1), \varphi_m^v(v_2)) \in \prec$ where $\prec$ is the partial order defined in tn.

The relation $\precsim_v$ for some vertex v might be cyclic, i.e., there might exist two children v_1 and v_2 of v with $(v_1, v_2) \in \precsim_v$ and $(v_2, v_1) \in \precsim_v$. We say that the child arrangement of v is *order-consistent iff* $\precsim_v$ is *acyclic*, that is, $\precsim_v$ forms a partial order.

A SOG is then defined over a PDT arranged in an order-consistent way. Given such a PDT $\Phi_K = (V, E, \varrho)$ with some height bound K, the SOG defined over it, written $\mathcal{S}(\Phi_K)$, is the graph $(\mathcal{L}(\Phi_K), \prec_S)$ where $\mathcal{L}(\Phi_K)$ is the set of vertices, and $\prec_S$ is the set of edges such that for any $l_1, l_2 \in \mathcal{L}(\Phi_K)$, $(l_1, l_2) \in \prec_S$ if and only if there exist $v_1, v_2 \in V$ which are the ancestors of l_1 and l_2, respectively, and v_1 and v_2 have the same parent vertex v with $v_1 \precsim_v v_2$.

One can notice that the SOG of a PDT encapsulates the yield of *every* valid decomposition tree stored by the PDT. Formally, it is proved [8] that for the SOG $(\mathcal{L}(\Phi_K), \prec_S)$ of some Φ_K, if $(l_1, l_2) \in \prec_S$ for some $l_1, l_2 \in \mathcal{L}(\Phi_K)$, then there exists a valid decomposition tree g in Φ_K with $yield(g) = (\mathcal{T}, \prec, \gamma)$ such that $(l_1, l_2) \in \prec$.

The construction of a PDT with an order-consistent child arrangement is well-studied [8]. We will thus skip the details of that here and focus on how to exploit such a PDT (and SOG) to verify a plan.

Provided a PDT Φ_K together with its SOG $\mathcal{S}(\Phi_K)$, we then want to craft a SAT formula that is satisfiable *iff* we can activate a valid decomposition tree in the PDT whose yield (which is a subgraph of the SOG) results in the plan to be verified. To this end, the formula should have clauses that are able to 1) simulate the construction of the PDT, 2) ensure that all activated vertices indeed constitute a valid decomposition tree, and 3) certify that the yield of the activated decomposition tree meets the criterion mentioned in Prop. 1.

Encoding for PDTs The encoding for the first two tasks already exists [7, 8]. We reproduce it for completeness. Let $\Phi_K = (V, E, \varrho)$ be a PDT of height K that is constructed in an order-consistent way, for each vertex $v \in V$, we define a SAT variable x_t^v for *every* task $t \in \varrho(v)$. Such a variable determines whether the task t in v is activated, i.e., whether v is activated as a vertex in a decomposition tree labeled with t. The variables x_t^v's must meet the criterion that *at most one* of them can be true, because only one decomposition tree in the PDT can be activated. Throughout the paper, we will use $\texttt{AMO}(X)$ with X be an arbitrary set of variables to denote the formula which enforces that at most one variable in X can be true. An efficient way to construct such a formula has been studied in the previous work [23]. We thus omit the details here. Let $\mathbb{T}(v) = \{x_t^v \mid t \in \varrho(v)\}$, $\texttt{AMO}(\mathbb{T}(v))$ thus ensures that at most one task can be activated in v.

Next we encode the constraints over an inner vertex v together with its children that if v is activated as a part of a decomposition tree and is labeled with a *compound* task c, some of its children must also be activated as a consequence of decomposing c. For each inner node v, we define a variable x_m^v for *every* method $m \in \mathcal{M}$ that indicates whether the method m is applied to the vertex v (to decompose the task in v that is activated). Let $\mathbb{M}(v) = \{x_m^v \mid m \in \mathcal{M}\}$, we again need to enforce that at most one method can be applied to the vertex, i.e., $\texttt{AMO}(\mathbb{M}(v))$. If x_m^v is set to true, then the task c decomposed by m must be activated in v, and if $c \notin \varrho(v)$, x_m^v must be set to false. Conversely, if the task c is activated in v, then at least one method must be applied to v that can decompose c. Formally, let $\mathbb{M}(c)$ be the set of methods that decompose c, we then have

$$\left(x_c^v \rightarrow \bigvee_{m \in \mathbb{M}(c)} x_m^v\right) \wedge \texttt{AMO}(\mathbb{M}(v))$$

Further, if a method m is applied to a vertex v, then v's children must also be activated that contain m's subtasks. We use the notation $\mathbb{C}(v, m)$ to refer to the subset of v's children on which m's subtasks are distributed. Let $m = (c, (\mathcal{T}, \prec, \gamma))$, we thus have

$$x_m^v \rightarrow \bigwedge_{v' \in \mathbb{C}(v,m)} x^{v'}_{\gamma(\varphi_m^v(v'))}$$

Recall that φ_m^v (a child arrangement function of v) maps a child of v to the respective $t \in \mathcal{T}$. Hence, $\gamma(\varphi_m^v(v'))$ returns the task in v' that is a subtask in m. Another important constraint is that if a method m is applied to a vertex v, all v's children on which *no* m's subtasks are distributed *cannot* be activated. That is, we have

$$x_m^v \rightarrow \bigwedge_{v' \in \mathbb{C}(v) \setminus \mathbb{C}(v,m)} \left(\bigwedge_{t \in \varrho(v')} \neg x_t^{v'}\right)$$

Lastly, we consider the case in which a *primitive* task a is activated in a vertex v. More specifically, such a primitive task must be *inherited* down to one of its children while the rest must be deactivated. This is formulated as

$$x_m^v \rightarrow x_a^{v^*} \wedge \left(\bigwedge_{v' \in \mathbb{C}(v) \setminus \{v^*\}} \left(\bigwedge_{t \in \varrho(v')} \neg x_t^{v'}\right)\right)$$

where $v^* \in \mathbb{C}(v)$ is an arbitrary child of v as it does not matter which child of v is activated. Such an inheritance ensures that the activated primitive task can be passed to some leaf of the PDT. For those vertices that are deactivated, *none* of their children can be activated, that is, for any (inner) vertex v, the following must hold:

$$\left(\bigwedge_{t \in \varrho(v)} \neg x_t^v\right) \rightarrow \bigwedge_{v' \in \mathbb{C}(v)} \left(\bigwedge_{t' \in \varrho(v')} \neg x_{t'}^{v'}\right)$$

Encoding for Verification We now turn to discuss how to exploit the SOG $\mathcal{S}(\Phi_K)$ of a PDT Φ_K to construct a SAT formula that asserts that the yield of an activated decomposition tree leads to the plan to be verified (i.e., task 3). Let $\mathcal{S}(\Phi_K) = (\mathcal{L}(\Phi_K), \prec_S)$ be the SOG of a PDT Φ_K and $\pi = \langle a_1 \cdots a_n \rangle$ the plan, we first define two types of matching between the actions in π and the vertices in the SOG that determines how the yield of the activated decomposition tree will be linearized. The first type of matching associates an action a_i in π to a vertex v of the SOG where the same action a_i is activated. Intuitively, this means that the vertex v will be the action a_i in the linearization of the yield of the activated decomposition tree. The second type of matching links a vertex v' where a primitive

task $a^* \in \mathcal{A}^*$ (which represents a method precondition) is activated to an action a_j in π. This indicates that the vertex v' will be placed *between* the actions a_{j-1} and a_j in the linearization (if $j = 1$, then v is simply placed before a_1).

Notably, for those vertices where an action $a^* \in \mathcal{A}^*$ is activated, it does not matter how they are ordered exactly between two actions in the plan. This is because those artificial actions have no effects, and hence, their positions in the linearization can only influence whether their preconditions are satisfied, but not the states.

For each action a_i (in π) and each vertex v in $\mathcal{S}(\Phi_K)$, we use the variable h_i^v to capture the first kind of matching (i.e., matching an action to a vertex). Naturally, h_i^v is set to false if $a_i \notin \varrho(v)$, and if a_i is matched to v, then a_i in v must be activated, i.e., $h_i^v \rightarrow x_{a_i}^v$. Further, a_i must be matched to *exact* one vertex. More concretely, let $\mathbb{H}(i) = \{h_i^v \mid v \in \mathcal{L}(\Phi_K)\}$. We have that

$$\texttt{AMO}\left(\mathbb{H}(i)\right) \wedge \left(\bigvee_{x \in \mathbb{H}(i)} x\right) \tag{1}$$

Importantly, a vertex v can be matched to *at most* one action, namely,

$$\texttt{AMO}\left(\{h_i^v \mid i \in \{1, \cdots, n\}\}\right) \tag{2}$$

Regarding the second type of matching, which places a vertex between two consecutive actions in the plan, we again define a variable $\hat{h}_i^v$ for each action a_i in π and each vertex v in $\mathcal{S}(\Phi_K)$. Note that if a vertex v contains no artificial actions representing a method precondition, then $\hat{h}_i^v$ must be set to false for all $i \in \{1, \cdots, n\}$ (i.e., for all actions in π), and if $\hat{h}_i^v$ is true, then at least one artificial action in v should be activated, that is,

$$\hat{h}_i^v \rightarrow \bigvee_{a \in \mathcal{A}^* \cap \varrho(v)} x_a^v \tag{3}$$

For any activated vertex v, it must be associated with some matching:

$$\left(\bigvee_{t \in \varrho(v)} x_t^v\right) \rightarrow \left(\bigvee_{i \in \{1, \cdots, n\}} \left(h_i^v \vee \hat{h}_i^v\right)\right) \tag{4}$$

This is to say that an activated vertex must be part of the linearization.

Next we shall encode that those two kinds of matching should respect the partial order defined over the vertices of the SOG, namely, its edges. To this end, we first define a variable f_i^v for each action a_i in π and vertex v in $\mathcal{L}(\Phi_K)$ indicating that the matching between a_i and v is forbidden. That is, if the forbiddenness holds, neither can a_i be matched to v nor can v be placed before a_i, i.e.,

$$f_i^v \rightarrow \left(\left(\neg h_i^v\right) \wedge \left(\neg \hat{h}_i^v\right)\right) \tag{5}$$

The ordering constraints over the vertices is encoded in terms of such forbiddenness. Specifically, if either h_i^v or $\hat{h}_i^v$ holds for some action a_i, assuming that $i > 1$, and vertex v, then the matching between a_{i-1} and *all* the successors of v should be forbidden. This indicates that if a_i is matched to v (*resp.* v is placed before a_i), then a_{i-1} cannot be matched to any successor of v (*resp.* none of v's successor can be placed before a_{i-1}). Let $S(v)$ be the set of all successors of the vertex v. The constraint is formulated as follows.

$$\left(h_i^v \vee \hat{h}_i^v\right) \rightarrow \bigwedge_{v' \in S(v)} f_{i-1}^{v'} \tag{6}$$

Similarly, we shall also encode the transitivity of forbiddenness, i.e., if the matching between a_i and v is forbidden, then so is the matching between a_i and all the successors of v, i.e.,

$$f_i^v \rightarrow \bigwedge_{v' \in S(v)} f_i^{v'} \tag{7}$$

Additionally, if f_i^v holds for some action a_i and vertex v, then the forbiddenness shall also hold between v and a_{i-1}, i.e., $f_i^v \rightarrow f_{i-1}^v$. In other words, the forbiddenness should be passed on to all the predecessors of a_i (i.e., to $a_1, \cdots, a_{i-1}$).

Having presented how the matching between the plan and the SOG is encoded, which determines how the yield of the activated decomposition tree is linearized, we now discuss the formula for enforcing that the linearization must be executable. That is, we want to ensure that the precondition of every action (including the actions in the plan and the artificial actions representing a method precondition) in the linearization is satisfied in the respective state.

For this, we first compute the state sequence $\overline{s} = \langle s_0 \cdots s_n \rangle$ obtained by applying the plan $\langle a_1 \cdots a_n \rangle$ in the initial state s_I in which $s_0 = s_I$ and $s_{i-1} \rightarrow_{a_i} s_i$ for each $1 \leq i \leq n$. If it turns out that the plan is *not* executable or $g \subseteq s_n$ does *not* hold, we can simply assert that the plan is not a solution.

Our next step is to certify that, in the linearization, the precondition of every artificial action is satisfied. To this end, for each proposition $p \in \mathcal{P}$ and state s_i in $\overline{s}$ ($0 \leq i \leq n$), we define a variable y_i^p which is set to true if $p \in s_i$, otherwise it is false. The variable thus encodes whether the proposition p holds in the state s_i. Intuitively, for every vertex v, if there is an artificial action a^* in it that is activated, and it is placed before some action a_i in the plan, then every proposition in the precondition must hold in s_{i-1}. Formally, let v be a vertex in the SOG, $a^* \in \mathcal{A}^* \cap \varrho(v)$ an artificial action in v, and a_i an action in the plan, the constraint is formulated as follows.

$$\left(\hat{h}_i^v \wedge x_{a^*}^v\right) \rightarrow \bigwedge_{p \in \texttt{prec}(a^*)} y_{i-1}^p \tag{8}$$

Soundness and Completeness Lastly, we prove that the SAT formula is satisfiable *iff* there exists a valid decomposition tree of height up to K ($K \in \mathbb{N}$) whose yield possesses an executable linearization of which the given plan is a subsequence. Previous works [7, 8] have already proved the correctness of the part of the formula that encodes the activation of a decomposition tree in the PDT. We thus focus on proving that the remaining formula is satisfiable *iff* the activated vertices in the SOG can constitute an executable linearization of which the plan is a subsequence.

Suppose that $tn = (\mathcal{T}, \prec, \gamma)$ is the yield of the activated decomposition tree that has a valid linearization $\overline{tn} = \langle t_1 \cdots t_{|\mathcal{T}|} \rangle$ such that the plan $\pi = \langle a_1 \cdots a_n \rangle$ is a subsequence of $\langle \gamma(t_1) \cdots \gamma(t_{|\mathcal{T}|}) \rangle$, and all actions in the sequence except those in π are artificial ones representing a method precondition. For convenience, we further define the function $\omega : \mathcal{T} \rightarrow \{\bot, a_1, \cdots, a_n\}$ such that $\omega(t_j) = a_i$ for some $j \in \{1, \cdots, |\mathcal{T}|\}$ and $i \in \{1, \cdots, n\}$ if t_j is corresponding to a_i, otherwise, $\omega(t_j) = \bot$ (i.e., undefined). The truth assignment which satisfies the formula is as follows. For every vertex $v \notin \mathcal{T}$, h_i^v and $\hat{h}_i^v$ are set to false for all $1 \leq i \leq n$. For every $1 \leq j \leq |\mathcal{T}|$ and $1 \leq i \leq n$, $h_i^{t_j}$ is set to true if $\omega(t_j) = a_i$, otherwise, it is set to false. Further, for each $1 < i \leq n$, $\hat{h}_i^{t_j}$ is set to true if $\omega(t_j) = \bot$ and there exist p, q with $p < j < q$ such that $\omega(t_p) = i - 1$ and $\omega(t_q) = i$, otherwise, it is false, and for $i = 1$, $\hat{h}_i^{t_j}$ is true if there exists $q > j$ with $\omega(t_q) = i$. For every action a_i in π, since there exists *exactly*

one j with $\omega(t_j) = a_i$, formulae 1 and 2 hold naturally. Note further that $\hat{h}_i^{t_j}$ for some a_i and t_j is set to true only if $\omega(t_j) = \bot$. It follows that formula 3 holds. The validation of formula 4 is the consequence of the fact that every activated vertex is a part of the linearization.

Regarding the variables that represent the forbiddenness, for every $1 \leq j \leq |\mathcal{T}|$, if $\omega(t_j) = a_i$ for some a_i in π with $i > 1$, we set $f_r^{t_k}$ to true for every r with $1 \leq r < i$ and t_k with $(t_j, t_k) \in \prec$. Similarly, if $\omega(t_j) = \bot$ and there exist p, q with $p < j < q$ such that $\omega(t_p) = i - 1$ and $\omega(t_q) = i$ for some $i > 1$, we let $f_r^{t_k}$ be true for every r with $1 \leq r < i$ and t_k with $(t_j, t_k) \in \prec$. The remaining variables representing forbiddenness are set to false. This assignment thus ensures that the formulae defined over the variables representing the forbiddenness are satisfied. Lastly, formula 8 holds because the linearization is executable.

Conversely, if there exists an assignment that satisfies the SAT formula, by the semantics of the defined variables, the yield of the activated decomposition tree has an executable linearization that has the given plan as a subsequence.

Maximal Heights of PDTs Having presented the construction of the SAT formula which is satisfiable if and only if there exists a decomposition tree within a PDT of a certain height whose yield leads to the plan, we now briefly introduce the computation for the upper bound of such heights such that for any decomposition tree of height larger than this bound, the length of its yield is *larger* than the length of the given plan. Such a bound K^* is used to terminate our verification procedure, namely, if no satisfiable SAT formulae are found for PDTs of height up to K^*, then the plan is not a solution. Having such a maximal height as the terminal signal is crucial especially for *cyclic* HTN problems in which a decomposition tree can grow indefinitely (an example of a cyclic HTN problem is the simulation of a classical planning problem [12] where a compound task can be recursively decomposed into any action).

Given a plan π and an HTN planning problem Π, it has been shown [6] that the maximal height of a PDT is $2 \times |\pi| \times (|\mathcal{C}| + 1)$ where $|\pi|$ is the length of π. This bound is computationally cheap, whereas it overly approximates the height. There exist some other approaches [1, 15] computing a better bound. They however either only work for a strict subset of HTN problems (called *acyclic* HTN problems) or require a problem to be transformed into a special format, which might result in exponential explosion. The best approach [9] that is applicable to general HTN problems and dominates all the others calculates the bound recursively. It is however computationally expensive.

4 Empirical Evaluation

In this section, we present the experimental results for our SAT-based approach. We compared our approach against the SOTA planning-based approach [17], which translates the plan verification problem into an HTN planning problem. Since the evaluation done by the authors of the planning-based approach already shown that their approach significantly outperforms all other existing ones [2, 4, 6], and we ran our experiments on a benchmark set that is an *extension* of the one used by them, we thus skipped the comparison against the other approaches. We ran the planning-based approach with both the SAT-based *planner* [8] and the progression-based planner [16] (a planner is used to solve the translated planning problem), and our approach was also run with the naive approach [5] and the recursive approach [9] for calculating the maximal height of a PDT, respectively. We evaluate those approaches in terms of both the coverage (i.e., the number of instances that can be solved) and the runtime, which are the two most important dimensions of the performance.

Configuration We used CryptoMiniSAT (5.8.0) [24] as the SAT solver in both our SAT-based verifier and the planning-based verifier (with the SAT-based planner). Note that the same SAT solver was also used in the experiments done by the authors of the planning-based verifier. Our experiments were run on the benchmark set that is an extension of the one which is constituted of the plans generated in the partial order track of the IPC 2020 on HTN planning. The original benchmark set has 1211 valid plans and 138 invalid ones. We increased the number of invalid plans from 138 to 339. We do so because the invalidity of 64 of the original 138 invalid plans is due to a very simple reason that those plans contain actions that are not in the respective planning problems. That is, those 64 plans can be verified in a preprocessing step without actually starting a verifier. We thus concern that the old benchmark set cannot precisely characterize the performance of a verifier in identifying invalid plans.

The new invalid plans were generated by randomly selecting 300 plans from the valid one, and for each selected plan, we introduced an error by either 1) adding an action to the plan, 2) deleting an action from it, or 3) swapping the positions of two actions in it. The reason for introducing only one single error to each plan is that we hypothesize that the complexity of identifying an invalid plan escalates when its proximity to a valid one increases. Lastly, we filtered out all plans that are still valid. The evaluation ran on an Intel Cascade Lake CPU, with 8Gb memory limit and 10-minutes timeout.

Experimental Results The experimental results on the valid plans are summarized in Fig. 1. The table presents the number of verified plans for each domain. The column labeled *loose* presents the results on running our approach with the naive algorithm for computing the maximal height of a PDT, which produces a loose bound, and the one labeled *tight* shows the results produced by the recursive algorithm, which returns a tight bound. The sub-column *sat* in the column *planning* contains the results for the planning-based approach employing the SAT planner, and the other sub-column *progression* is the results for using the progression-based planner. The best result achieved in each domain is highlighted by using bold text. As can be seen from the table, both two variants of our approach verified 1206 plans in all, which was better than the planning-based approach with the SAT planner (1203) and with the progression-based planner (1115).

Fig. 3 accordingly depicts the number of plans that can be verified by the two approaches (each with two variants) against the runtime, that is, each point (x, y) on a curve indicates that there are x plans that can be verified in y seconds by the respective approach. The plot shows that our approach is consistently faster than the planning-based one, and the computation for calculating a tight bound for the height of a PDT does not cause a significant overhead to our verifier.

The results for verifying invalid plans are presented in Fig. 2. The table indicates that our approach with a tight height bound for a PDT has the best performance that solves all 339 instances. It is followed by our approach with a loose height bound, solving 317 instances, and then by the planning-based approach with the progression-based planner, solving 300 instances. The planning-based approach using the SAT planner has the worst results, solving only 64 instances. In particular, those 64 plans are the trivial instances which are verified in preprocessing, i.e., the planning-based approach using the SAT planner failed to verify *any* instance that requires true verification.

The runtime for verifying invalid plans for both approaches is also depicted in Fig. 4. One can see that our approach with a tight

	Instances	SAT (Ours)		Planning	
		Loose	Tight	SAT	Progression
Satellite	269	**269**	**269**	**269**	**269**
Transport	219	**219**	**219**	**219**	141
Rover	171	**171**	**171**	**171**	163
Woodworking	162	**162**	**162**	**162**	**162**
Monroe (FO)	130	**130**	**130**	**130**	**130**
Monroe (PO)	103	**103**	**103**	**103**	**103**
Barman-BDI	68	**63**	**63**	58	58
UM-Translog	57	**57**	**57**	**57**	**57**
PCP	31	**31**	**31**	**31**	**31**
Zenotravel	1	**1**	**1**	**1**	**1**
	1211	**1206**	**1206**	1201	1115

Table 1. The number of valid plans that are verified per domain.

	Instances	SAT (Ours)		Planning	
		Loose	Tight	SAT	Progression
Satellite	66	**66**	**66**	23	**66**
Transport	64	54	**64**	18	33
UM-Translog	59	**59**	**59**	14	**59**
Rover	53	41	**53**	6	47
Monroe (FO)	24	**24**	**24**	1	**24**
Woodworking	21	**21**	**21**	0	**21**
Barman-BDI	18	**18**	**18**	0	16
Monroe (PO)	18	**18**	**18**	2	**18**
PCP	12	**12**	**12**	0	**12**
Zenotravel	4	**4**	**4**	0	**4**
	339	317	**339**	64	300

Table 2. The number of invalid plans that are verified per domain.

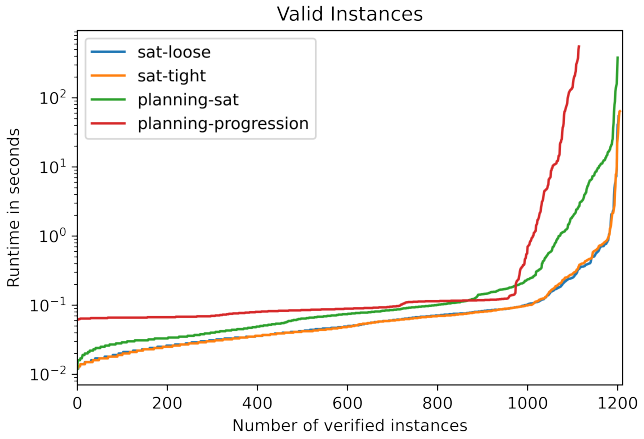

Figure 3. The number of verified valid plans against the runtime.

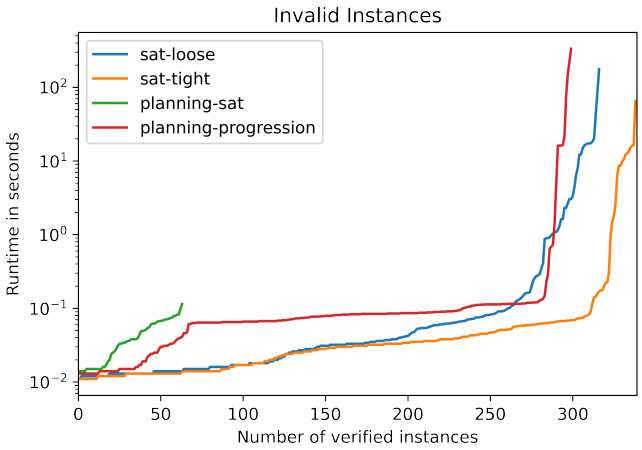

Figure 4. The number of verified invalid plans against the runtime.

height bound clearly outperforms the others (note that in both Fig. 3 and 4, those unsolved instances are not displayed). Additionally, one could also find that the runtime curves for our approach with the tight bound and with the loose bound are similar in verifying valid plans. This is because the bound only serves as the signal for stopping the iterations, and it is actually never reached in verifying valid plans. Thus, the only difference between these two variants when verifying valid plans is the computation for the bound, and computing a tight bound only has a polynomial overhead, which is a small fraction for the $\mathbb{NP}$-hard plan verification problem.

Discussion Although the *valid* plans verified by our approach only slightly outnumbered those verified by the planning-based approach with the SAT planner, the latter one underperformed the former one significantly in verifying invalid plans. In fact, as we have mentioned earlier, it is practically infeasible in verifying invalid plans. In contrast, the planning-based approach with the progression-based planner is effective in verifying invalid plans, however, it is still underperforms ours approach with both the loose and tight height bound. Further, the planning-based approach with the progression-based planner is also less powerful in verifying valid plans. We thus argue that our approach strictly dominates the planning-based one, which currently is the SOTA, independent of what internal planner is used.

Further, notice that our approach is persistently faster than the one based on planning (in verifying both valid and invalid plans). This is another significant advantage of our approach. The efficiency of an independent plan verifier is a crucial factor to be considered when it is used as an intermediate step in some applications, e.g., in solving planning problems [14].

5 Conclusion

In this paper, we have developed a novel plan verification approach for HTN planning based on SAT, exploiting the data structures PDTs and SOGs. This new SAT-based approach supports method preconditions and empty methods. Our empirical evaluation shows that it dominates the current SOTA planning-based approach in terms of both coverage and efficiency, independent of what internal planner is used, which makes our approach a powerful tool to be used when an independent plan verifier is required, e.g., when plan verification serves as an intermediate step in some applications.

References

[1] Ron Alford, Gregor Behnke, Daniel Höller, Pascal Bercher, Susanne Biundo, and David Aha, 'Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems', in *Proceedings of the 26th International Conference on Automated Planning and Scheduling, ICAPS 2016*, pp. 20–28. AAAI, (2016).

[2] Roman Barták, Adrien Maillard, and Rafael Cauê Cardoso, 'Validation of hierarchical plans via parsing of attribute grammars', in *Proceedings of the 28th International Conference on Automated Planning and Scheduling, ICAPS 2018*, pp. 11–19. AAAI, (2018).

[3] Roman Barták, Simona Ondrcková, Gregor Behnke, and Pascal Bercher, 'On the verification of totally-ordered HTN plans', in *Proceedings of the 33rd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2021*, pp. 263–267. IEEE, (2021).

[4] Roman Barták, Simona Ondrcková, Adrien Maillard, Gregor Behnke, and Pascal Bercher, '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*, pp. 118–125. IEEE, (2020).

[5] Gregor Behnke, Daniel Höller, and Susanne Biundo, 'On the complexity of HTN plan verification and its implications for plan recognition', in *Proceedings of the 25th International Conference on Automated Planning and Scheduling, ICAPS 2015*, pp. 25–33. AAAI, (2015).

[6] Gregor Behnke, Daniel Höller, and Susanne Biundo, '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*, pp. 20–28. AAAI, (2017).

[7] Gregor Behnke, Daniel Höller, and Susanne Biundo, 'totSAT - Totally-ordered hierarchical planning through SAT', in *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pp. 6110–6118. AAAI, (2018).

[8] Gregor Behnke, Daniel Höller, and Susanne Biundo, 'Bringing order to chaos - A compact representation of partial order in SAT-based HTN planning', in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence, AAAI 2019*, pp. 7520–7529. AAAI, (2019).

[9] Gregor Behnke, Daniel Höller, and Susanne Biundo, 'Finding optimal solutions in HTN planning - A SAT-based approach', in *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019*, pp. 5500–5508. IJCAI, (2019).

[10] Pascal Bercher, Ron Alford, and Daniel Höller, 'A survey on hierarchical planning - one abstract idea, many concrete realizations', in *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019*, pp. 6267–6275. IJCAI, (2019).

[11] Pascal Bercher, Daniel Höller, Gregor Behnke, and Susanne Biundo, 'More than a name? on implications of preconditions and effects of compound HTN planning tasks', in *Proceedings of the 22nd European Conference on Artificial Intelligence, ECAI 2016*, pp. 225–233. IOS, (2016).

[12] Kutluhan Erol, James A. Hendler, and Dana S. Nau, 'Complexity results for HTN planning', *Annals of Mathematics and Artificial Intelligence*, **18**, 69–93, (1996).

[13] Thomas Geier and Pascal Bercher, 'On the decidability of HTN planning with task insertion', in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011*, ed., Toby Walsh, pp. 1955–1961. AAAI, (2011).

[14] Daniel Höller, 'Translating totally ordered HTN planning problems to classical planning problems using regular approximation of context-free languages', in *Proceedings of the 31st International Conference on Automated Planning and Scheduling, ICAPS 2021*, pp. 159–167. AAAI, (2021).

[15] Daniel Höller, Gregor Behnke, Pascal Bercher, and Susanne Biundo, 'Language classification of hierarchical planning problems', in *Proceedings of the 21st European Conference on Artificial Intelligence, ECAI 2014*, pp. 447–452. IOS, (2014).

[16] Daniel Höller, Pascal Bercher, Gregor Behnke, and Susanne Biundo, 'HTN planning as heuristic progression search', *Journal of Artificial Intelligence Research*, **67**, 835–880, (2020).

[17] Daniel Höller, Julia Wichlacz, Pascal Bercher, and Gregor Behnke, 'Compiling HTN plan verification problems into HTN planning problems', in *Proceedings of the 32nd International Conference on Automated Planning and Scheduling, ICAPS 2022*, pp. 145–150. AAAI, (2022).

[18] Songtuan Lin, Gregor Behnke, Simona Ondrčková, Roman Barták, and Pascal Bercher, 'On total-order HTN plan verification with method preconditions – An extension of the CYK parsing algorithm', in *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI 2023*. AAAI, (2023).

[19] Songtuan Lin and Pascal Bercher, 'Change the world - how hard can that be? On the computational complexity of fixing planning models', in *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI 2021*, pp. 4152–4159. IJCAI, (2021).

[20] Songtuan Lin and Pascal Bercher, 'Was fixing this *Really* that hard? On the complexity of correcting HTN domains', in *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI 2023*. AAAI, (2023).

[21] Songtuan Lin, Alban Grastien, and Pascal Bercher, 'Towards automated modeling assistance: An efficient approach for repairing flawed planning domains', in *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI 2023*. AAAI, (2023).

[22] Karen L. Myers, Peter Jarvis, Mabry Tyson, and Michael Wolverton, 'A mixed-initiative framework for robust plan sketching', in *Proceedings of the 13th International Conference on Automated Planning and Scheduling ,ICAPS 2003*, pp. 256–266. AAAI, (2003).

[23] Carsten Sinz, 'Towards an optimal CNF encoding of boolean cardinality constraints', in *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming, CP 2005*, pp. 827–831. Springer, (2005).

[24] Mate Soos, Karsten Nohl, and Claude Castelluccia, 'Extending SAT solvers to cryptographic problems', in *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT 2009,*, pp. 244–257. Springer, (2009).