This is a solution! (. . . but is it though?)
Verifying solutions of hierarchical planning problems

Gregor Behnke and **Daniel Höller** and **Susanne Biundo**
Institute of Artificial Intelligence, Ulm University, D-89069 Ulm, Germany
{gregor.behnke, daniel.hoeller, susanne.biundo}@uni-ulm.de

Abstract

Plan-Verification is the task of determining whether a plan is a solution to a given planning problem. Any plan verifier has, apart from showing that verifying plans is possible in practice, a wide range of possible applications. These include mixed-initiative planning, where a user is integrated into the planning process, and local search, e.g., for post-optimising plans or for plan repair. In addition to its practical interest, plan verification is also a problem worth investigating for theoretical reasons. Recent work showed plan verification for hierarchical planning problems to be $\mathbb{NP}$-complete, as opposed to classical planning where it is in $\mathbb{P}$. As such, plan verification for hierarchical planning problem was – until now – not possible. We describe the first plan verifier for hierarchical planning. It uses a translation of the problem into a SAT formula. Further we conduct an empirical evaluation, showing that the correct output is produced within acceptable time.

1 Introduction

The task of plan verification plays a significant role in both planning research and application. In mixed-initiative planning, a solution is usually presented to the user and the system inquires for his opinion of it, which is (in general) interpreted as a request to alter the current plan in some way (Ai-Chang et al. 2004; Fernández-Olivares et al. 2006; Ferguson, Allen, and Miller 1996). The necessary modifications can, e.g., be performed by changing the plan and verifying the result. If it is a solution, the user is given the new plan for critique, if not, the user has to be informed of the failure to adhere to his wishes. For researchers, it is crucial to be able to check correctness of planners and their results using an independent technique (i.e. an algorithm which is not planning). A plan verifier provides such an independent system. Like in mixed-initiative planning, local search procedures (Gerevini and Serina 2002) usually change a current plan in some way – in order to obtain a solution, to lower costs via post-processing, or to repair a plan whose execution has failed. After these changes have been performed, the resulting plan must (again) be a solution to the original planning problem. This is ensured by applying a plan verifier. Lastly, when planners participate in planning competitions, all produced solutions need to be checked for correctness.

However, the *classical* planning literature rarely considers the task of plan verification, due to its simplicity in the classical setting. For more extended classical formalisms, like PDDL+, there are dedicated verification tools, the best-known of which is VAL, which also provides verification for pure classical planning (Howey, Long, and Fox 2004). For hierarchical planning – which we consider in form of Hierarchical Task Network (HTN) planning (Erol, Hendler, and Nau 1996) – there are until now neither such tools nor are plan verification capabilities integrated into any planning system or application, although HTN planning is widely used in planning-based applications (Nau et al. 2005; Bercher et al. 2014; 2015). Recently, we have shown that HTN plan verification is an $\mathbb{NP}$-complete problem (Behnke, Höller, and Biundo 2015). We present the first HTN plan verifier which lays the necessary groundwork for future research in this area.

We propose an approach for HTN plan verification based on a translation into a SAT formula, which is satisfiable if and only if a given plan is a solution to the planning problem. By this translation, we utilise the efficiency of modern SAT solvers. The translation process itself requires a bound K, which (intuitively spoken) is the maximum height of the decomposition hierarchy any witness for a plan being a solution can have. Such a bound was used to show that HTN plan verification is in $\mathbb{NP}$. We show that our previous estimation (2015) was unnecessarily high, and that for existing benchmark domains it can be reduced by a factor of up to 1000, by providing two new, more succinct estimations for K. Without this reduction, it would be impossible to build the SAT formulae which would have well over one billion clauses in many cases. The presented reductions are not only applicable for our plan verification technique, but might also be used for other verification approaches or even as a blocking technique for planning itself. To show that the translation approach is usable in practice, we conducted an empirical evaluation involving several hierarchical benchmark domains, which have either been used in the past to test the performance of hierarchical planning or plan recognition approaches. In our experiments, we show that the translation approach is able to verify all provided solutions and to reject non-solutions, within reasonable time. Most instances are solved in under 100 seconds, while only a few hard ones needed up to 1123 seconds.

2 Formal Framework

This section introduces the HTN formalism by Geier and Bercher (2011) with some extensions (see Höller et al. 2016). In HTN planning, there are two distinct types of tasks, *primitive* tasks (also called actions) are those that can be executed directly; *compound* tasks (also called abstract) need to be decomposed until only primitive tasks are left. We will call these sets A and C, respectively. Let N be the set of all task names, i.e., $N = A \cup C$.

Tasks are organised in so-called *task networks*, which represent partial plans. A task network is a triple $tn = (T, \prec, \alpha)$. T is a non-empty set of identifiers. The function $\alpha : T \rightarrow N$ maps the identifiers to the actual task names. That way, a task network can contain a task more than once. A set $\prec : T \times T$ of ordering constraints defines a partial ordering on the identifiers. We will use $T(tn)$, $\prec(tn)$ and $\alpha(tn)$ to denote the set of identifiers of a task network, its ordering and the task name mapping, respectively. If two task networks $tn = (T, \prec, \alpha)$ and $tn' = (T', \prec', \alpha')$ differ only in their identifiers, i.e. there is a bijection $\sigma : T \rightarrow T'$ so that for all identifiers $t, t' \in T$ holds that $[(t, t') \in \prec] \Leftrightarrow [(\sigma(t), \sigma(t')) \in \prec']$ and $\alpha(t) = \alpha'(\sigma(t))$, they are called *isomorphic* (written $tn \cong tn'$).

The set of decomposition methods M defines how compound tasks may be refined. A method $m \in M$ is a pair (c, tn) of a compound task $c \in C$ and a task network tn, called the method's subnetwork. When a task c is decomposed, it is removed from the task network, the subnetwork is added and all the ordering constraints that have been in the network for c are introduced for the added tasks.

Formally, a method (c, tn) decomposes a task network $tn_1 = (T_1, \prec_1, \alpha_1)$ into a task network $tn_2 = (T_2, \prec_2, \alpha_2)$ if $t \in T_1$ with $\alpha_1(t) = c$ and if there exists a task network $tn' = (T', \prec', \alpha')$ with $tn' \cong tn$ and $T_1 \cap T' = \emptyset$. The task network tn_2 is defined as

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

To denote that a task network tn can be decomposed into tn' by applying an arbitrary number of decompositions, we will write $tn \rightarrow^*_{TD} tn'$.

Applicability and state transition of primitive tasks is defined in a STRIPS-like manner using a set of state-fluents L. The functions $prec$, add, and del (all functions $A \rightarrow 2^L$) map a primitive task to its preconditions, add- and delete-effects, respectively. Whether a primitive task a is applicable to a state s is given by the function $\tau : A \times 2^L \rightarrow \{true, false\}$ with $\tau(a, s) \Leftrightarrow prec(a) \subseteq s$. Given that $\tau(a, s)$ holds, the state resulting from the application is given by the state transition function $\gamma : A \times 2^L \rightarrow 2^L$ with $\gamma(a, s) = (s \setminus del(a)) \cup add(a)$.

Now we can define an HTN planning problem as a tuple $\mathcal{P} = (L, C, A, M, s_0, tn_I, g, \delta)$ with $\delta = (prec, add, del)$. L is a set of propositional environment facts, C and A the sets of task names, M the set of decomposition methods, $s_0 \in 2^L$ the initial state, $g \in 2^L$ the goal description and tn_I the initial task network.

A task network $tn = (T, \prec, \alpha)$ is a solution to a planning problem $\mathcal{P}$ if and only if

- all tasks are primitive,
- $tn_I \rightarrow^*_{TD} (T, \prec', \alpha)$ with $\prec \supseteq \prec'$,
- all sequences $(t_1 t_2 \ldots t_n)$ of the task identifiers that are in line with $\prec$ are applicable, and
- for all these sequences $s_n = \gamma(\alpha(t_n), \gamma(\alpha(t_{n-1}), \ldots \gamma(\alpha(t_1), s_0) \ldots))$ is a goal state, i.e., $s_n \supseteq g$.

We will denote the set of all solutions to an HTN planning problem $\mathcal{P}$ as $Sol(\mathcal{P})$.

The formalism is similar to the one used by the UMCP planner (Erol, Hendler, and Nau 1994), but slightly more restrictive than the one used in SHOP2 (Nau et al. 2003), as we do not allow decomposition methods to have preconditions. This is not a real restriction, since one can always compile a planning problem with method preconditions into one without them by inserting a new primitive task into each method having the method's precondition as its precondition, no effect, and preceding all other tasks in the method. In fact, SHOP2 uses this transformation already internally.

3 HTN Plan Verification

We have previously (2015) defined HTN plan verification as the problem of determining whether a task network tn is a solution to a planning problem $\mathcal{P}$. Instead of verifying a task network tn, we restrict ourselves to verifying task sequences π (or in other words a totally ordered task network). By definition, every linearisation of a solution to an HTN planning problem must be executable and reach a goal state, so this restriction is mere technicality. In practice, HTN planners like SHOP2 (Nau et al. 2003) often generate such totally ordered solutions. The problem of verifying task sequences is formally defined as follows:

Definition 1 (VERIFYSEQ)**.** *Given a planning problem $\mathcal{P}$ and a sequence of primitive tasks $\pi = (t_1, \ldots, t_n)$, the problem* VERIFYSEQ *is to decide whether for*

$$tn(\pi) = ([n], \{(i, i{+}1) \mid 1 \leq i < n\}, \{(i, t_i) \mid 1 \leq i \leq n\})$$

the relation $tn \in Sol(\mathcal{P})$ holds.

Clearly, we can separate the problem VERIFYSEQ into two independent subproblems: to verify executability and to determine whether the task sequence is a valid decomposition of the initial plan. Since checkers for the first (like VAL (Howey, Long, and Fox 2004)) are readily available, we will focus on the latter task that is defined as:

Definition 2 (DECOMPSEQ)**.** *Given a planning problem $\mathcal{P}$ and a sequence of primitive tasks π. The problem* DECOMPSEQ *is to decide whether $tn_I \rightarrow^*_{TD} tn(\pi)$ holds.*

Both VERIFYSEQ and DECOMPSEQ are $\mathbb{NP}$-complete (Behnke, Höller, and Biundo 2015). This shows that the difficulty in verifying solutions to HTN planning problems lies in finding a valid decomposition leading to π. This is more than merely checking constraints contained in the given sequence, as we first have to find these constraints – in form of applied methods.

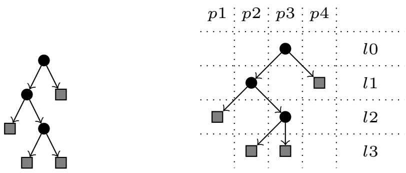


Figure 1: A decomposition tree. Grey boxes indicate primitive tasks and black circles abstract tasks. The right figure shows an assignment of the tasks to layers and positions.

4 Translating DecompSEQ into SAT

In this section we describe how a plan verification problem $(\mathcal{P}, \pi)$ can be transformed into a SAT-formula and show that the translation is correct. A satisfying assignment of the formula will represent a *decomposition tree* (Geier and Bercher 2011), a means to describe the process of decomposition which led from the initial task network to a solution. The vertices of the tree represent the tasks (both compound and primitive) that occur in task networks tn during the decomposition of the initial task network, while its directed edges represent which task was decomposed into which other tasks. An example of such a decomposition tree is depicted in Figure 1. As shown by Geier and Bercher (2011, Prop. 1), a task network tn is a solution if and only if a decomposition tree, whose leafs are the actions of tn, exists such that $\prec(tn)$ is a superset of the ordering constraints imposed by the methods applied in the tree. Since the original formulation of decomposition trees was done for HTN problems where the initial plan only contains a single abstract action, we (technically) have to change the tree into a forest, since every task in the initial plan is a root of a separate part of the tree. To show equivalence one introduces a new artificial initial task, which has only a single method containing the original initial plan.

Our formula is constructed such that it can express every possible decomposition tree rooted at the initial task network whose height is limited to a value K and whose leafs constitute the tasks of π. If the formula is satisfiable, we can construct a valid decomposition tree for π from the valuation, if not we have shown that no such tree – with a height of up to K – exists. This reduction has some similarities to the Steiner Tree problem, where given a graph $G = (V, E)$ and a set of vertices $S \subseteq V$ we are asked to find the smallest subtree of G which contains all nodes from S. In our setting the initial abstract task, and the nodes representing tasks in the given sequence are the nodes of S and we have to decide whether a subtree of the graph of all possible decompositions – fulfilling some additional criteria – exists. As such, our encoding bears some similarities to the one proposed for the Steiner Tree problem by Kautz, Selman, and Jiang (1997), which is based on a DFS traversal of the Steiner Root. Since our graph is acyclic, directed and the nodes in S are very specifically placed, the formula we construct is significantly simpler than theirs.

To ensure correctness of our technique, we have to determine a value for K such that if there is a decomposition tree whose leafs are π, then there is also one which has a height of at most K. Note that requiring the existence of K is not a restriction to the model but is a directly implied by the sequence to be verified. In previous work, we have shown that such a bound K_{theo} exists, meaning that a deeper recursion can never be necessary (Behnke, Höller, and Biundo 2015). Section 5 describes how to compute a succinct bound for K.

The formula describes the tree in terms of K layers, where the layer l contains exactly the tasks having the distance l from any of the roots of the tree. The assignment of a decomposition tree into layers is illustrated in Figure 1. Here the tasks in the initial plan always belong to layer 0, while the tasks contained in π might occur at any layer. To make our construction easier, we assume from this point on that if a primitive task is contained in layer i, it is also contained in all layers $j > i$. With this, π has to be equal to the Kth layer of the decomposition tree – to be more precise it has to be a valid total ordering of the Kth layer. Each task in each layer is also assigned a position in that layer as shown in Figure 1. Primitive tasks which are also contained in the previous layer will always have the same position they had in the previous layer. Due to the fact that the task networks in all methods are non-empty, we know that the number of tasks in a layer cannot decrease (which would require a method with an empty task network). Given that π has a fixed length and that the tasks in the Kth layer have to be equal to π, every layer prior to the Kth can contain at most $|\pi|$ tasks.

To ease the description of the SAT formula, we make the following assumptions and introduce the following abbreviation: We assume that the task identifiers T in all methods are the natural numbers from 1 to $|T|$. We further define $\Delta = \max_{(c,tn)\in M} |T(tn)|$ as the size of the largest method. The formula will contain several at-most-one constraints, stating that only a single atom of a given set can be true, which we denote by the functor $\mathbb{M}$. The intuitive way to encode $\mathbb{M}$ leads to a quadratic number of clauses (each pair of variables excluding each other). In some of our evaluation domains this would lead to a very high numbers of clauses (up to 442 trillion), making the construction of the SAT formula and with that verification hard or even practically impossible. So we have used the so-called *binary encoding* for all our at-most-one constraints, which reduces the number of clauses needed to express the constraint over n variables to $n \log n$ while only introducing $\log n$ additional variables (Frisch et al. 2005). This resulted in 390 million clauses for the largest instance. We will not define the SAT formula $\mathcal{F}$ in disjunctive normal form (as required by most SAT solvers) but rather as a general formula, since the transformation into DNF can be performed easily and only increases the number of clauses linearly.

All variables contained in the formula will have a superscipt l and a subscript p indicating the layer and position they belong to. Figure 1 shows an example for the assignment of a decomposition tree to such layers and positions. The overall structure of the formula is depicted visually in Figure 2. While constructing the formula, we will use the terms "child" and "father" of t to refer to the tasks connected to t in the next and the previous layer, respectively. The formula contains seven different kinds of variables, who represent the following statements.

- a^l_p – task a is selected
- m^l_p – method m is applied to a^l_p
- used^l_p – the position has some selected task
- abs^l_p – the task is abstract
- $\mathit{cPos}^l_{p_1,p_2,t}$ – the decomposition of $a^l_{p_1}$ leads to a task $a^{l+1}_{p_2}$ such that it is the task t of the applied method, or if $a^l_{p_1}$ is primitive then $a^{l+1}_{p_2}$ is kept to the next layer
- $\mathit{ch}^l_{p_1,p_2}$ – $a^{l+1}_{p_2}$ is a successor of $a^l_{p_1}$
- $\mathit{bef}^l_{p_1,p_2}$ – $a^l_{p_1}$ is ordered before $a^l_{p_2}$

Initially we divide the formula F into three parts, which encode the decomposition tree, describe the task network to be verified, and the initial task network.

$$\mathcal{F} = \mathit{decomposition} \wedge \mathit{solution} \wedge \mathit{initialTN}$$

The decomposition formula can be subdivided into the formulae pertaining to each layer of the graph and to each position therein. This is possible as the tasks in each layer do not influence each other – with the exception of their ordering.

$$\mathit{decomposition} = \bigwedge_{l=0}^{K} \left(\bigwedge_{p=1}^{|\pi|} \mathit{dec}(l,p) \right) \wedge \mathit{order}(l)$$

Next, we can subdivide each such formula into nine formulae describing the decomposition process. We will next explain the meaning of each of these formulae.

$$\begin{aligned}\mathit{dec}(l,p) = {} & \mathit{selectAction}(l,p) \wedge \mathit{methodChildren}(l,p) \wedge \\ & \mathit{maintainOrder}(l,p) \wedge \mathit{applyMethod}(l,p) \wedge \\ & \mathit{mustBeChildOf}(l,p) \wedge \mathit{fatherMustExist}(l,p) \wedge \\ & \mathit{childImpliesChildOf}(l,p) \wedge \mathit{maintainPrimitive}(l,p)\end{aligned}$$

The first four subformulae ensure that every satisfying valuation of $\mathcal{F}$ describes a valid tree, in the sense that every possible node position is used only once, that every node in a non-last layer has a successor in the next, and every node in a non-first layer has a predecessor in the previous layer. Additionally, they contain clauses which introduce abbreviation literals (like *used* and *ch*) which will be used later on to allow for a smaller formula.

selectAction enforces that at most a single action can be chosen at each position. It further requires to mark the respective position as used and (if so) abstract.

$$\mathit{selectAction}(l,p) = \mathop{\mathbb{M}}_{a \in A} a^l_p \wedge$$
$$\left(\bigvee_{a \in A \cup C} a^l_p \right) \leftrightarrow \mathit{used}^l_p \wedge \left(\bigvee_{c \in C} c^l_p \right) \leftrightarrow \mathit{abs}^l_p$$

The *mustBeChildOf* formulae state that each task must be the child of a task in the previous layer (his *father*), ensuring that tasks are only added through decomposition.

$$\mathit{mustBeChildOf}(l,p) = \bigwedge_{t=1}^{\Delta} \mathop{\mathbb{M}}_{p'=1}^{|\pi|} \mathit{cPos}^{l+1}_{p',p,t} \wedge$$
$$\mathop{\mathbb{M}}_{p'=1,t=1}^{|\pi|,\Delta} \mathit{cPos}^l_{p,p',t} \wedge \mathit{used}^l_p \rightarrow \bigvee_{p'=1}^{|\pi|} \bigvee_{t=1}^{\Delta} \mathit{cPos}^l_{p,p',t}$$

Similarly, *fatherMustExist* describes that the father must exist, and that it must be abstract if it is not the first child of the father (since primitive tasks can have only a single child – their own copy in the next layer).

$$\mathit{fatherMustExist}(l,p) = \bigwedge_{p'=1}^{|\pi|} \bigwedge_{t=1}^{\Delta} \mathit{cPos}^l_{p,p',t} \rightarrow \mathit{used}^{l+1}_{p'} \wedge$$
$$\texttt{if } t \neq 1 \vee p \neq p' \texttt{ then } \mathit{cPos}^l_{p,p',t} \rightarrow \mathit{abs}^{l+1}_{p'} \texttt{ else } \mathit{true}$$

Lastly *childImpliesChildOf* introduces an abbreviation of the *cPos* variables, to make the ordering formulae smaller.

$$\mathit{childImpliesChildOf}(l,p) = \bigwedge_{p'=1}^{|\pi|}$$
$$\left(\mathit{ch}^l_{p,p'} \rightarrow \bigvee_{t=1}^{\Delta} \mathit{cPos}^l_{p,p',t} \right) \wedge \bigwedge_{t=1}^{\Delta} \mathit{cPos}^l_{p,p',t} \rightarrow \mathit{ch}^l_{p,p'}$$

The *applyMethod* and *methodChildren* formulae are the most important part of the overall formula, as they encode the mechanism of decomposition itself, which is the creation of a method's subtasks in the next layer. *applyMethod* enforces that if a compound task is chosen at some position, an applicable decomposition method has to be applied and that at most one method can be applied.

$$\mathit{applyMethod}(l,p) = \mathop{\mathbb{M}}_{m \in M} m^l_p \wedge$$
$$\left(\bigwedge_{m=(c,tn) \in M} m^l_p \rightarrow c^l_p \right) \wedge \left(\mathit{abs}^l_p \rightarrow \bigvee_{m \in M} m^l_p \right)$$

The *methodChildren* formula comprises three main parts. The first asserts that, if a certain method with s subtasks is applied at the position p, then p has to have exactly s children in the next layer. The second determines the tasks assigned to these children, while the last encodes the ordering contained in the method.

$$\mathit{methodChildren}(l,p) = \bigwedge_{m=(c,tn) \in M} \left[\bigwedge_{t \in T(tn)} \left[\right. \right.$$
$$m^l_p \rightarrow \left(\left(\bigvee_{p'=1}^{|\pi|} \mathit{cPos}^l_{p,p',t} \right) \wedge \bigwedge_{t=|T(tn)|+1}^{\Delta} \bigwedge_{p'=1}^{|\pi|} \neg \mathit{cPos}^l_{p,p',t} \right)$$
$$\left. \wedge \left(\bigwedge_{p'=1}^{|\pi|} \mathit{cPos}^l_{p,p',t} \wedge m^l_p \rightarrow \alpha(tn)(t)^{l+1}_{p'} \right) \right] \wedge$$
$$\bigwedge_{(t_b,t_a) \in \prec(tn)} \bigwedge_{p_a=1}^{|\pi|} \bigwedge_{p_b=1}^{|\pi|} m^l_p \wedge \mathit{cPos}^l_{p,p_b,t_b} \wedge \mathit{cPos}^l_{p,p_a,t_a}$$
$$\left. \rightarrow \mathit{bef}^{l+1}_{p_b,p_a} \right]$$

We allow for the tasks contained in a method to be positioned at arbitrary positions in the next layer. This introduces additional non-determinism, which is generally discouraged when constructing SAT formulae. We have also experimented with a version of the formula where the positioning of tasks is kept from one layer to the next. Using this formula we have observed that it seems empirically to be harder to prove satisfiability. We presume that this is due to the fact that satisfiability of the formula does not depend on the choice of the position – if the formula is satisfiable, any positioning is valid.

The $\mathit{maintainPrimitive}$ formula ensures that every primitive task in a layer is also present in the next layer. It also enforces that those inherited primitive tasks do not change their positions. This allows for fast pruning of wrong choices through unit propagation, as the primitive tasks of the task network to be verified are asserted in the last layer.

$$\mathit{maintainPrimitive}(l,p) = \bigwedge_{o\in O} o_p^{l-1} \rightarrow o_p^l \wedge \mathit{cPos}^{l-1}_{p,p,1}$$

The $\mathit{maintainOrder}$ formula enforces that if two tasks were ordered in the previous layer all the tasks which were generated from them, either by decomposition or by maintaining primitives, will inherit their ordering.

$$\mathit{maintainOrder}(l,p_1^2) = \bigwedge_{p_2^2=1}^{|\pi|} \bigwedge_{p_1^1=1}^{|\pi|} \bigwedge_{p_2^1=1}^{|\pi|} \mathit{ch}^l_{p_1^1,p_1^2} \wedge \mathit{ch}^l_{p_2^1,p_2^2} \wedge \mathit{bef}^l_{p_1^1,p_2^1} \rightarrow \mathit{bef}^{l+1}_{p_1^2,p_2^2}$$

The order clauses ensure that the partial order in each layer is always valid. Here, we allow adding *additional* ordering constraints, which does not influence the correctness of the formula, as the same ordering constraint could have been added when choosing the linearisation π.

$$\mathit{order}(l) = \bigwedge_{p_1=1}^{|\pi|} \bigwedge_{p_2=1,p_2\neq p_1}^{|\pi|} \left(\mathit{bef}^l_{p_1,p_2} \rightarrow \neg \mathit{bef}^l_{p_2,p_1} \bigwedge_{p_3=1,p_3\neq p_1,p_3\neq p_2}^{|\pi|} \mathit{bef}^l_{p_1,p_2} \wedge \mathit{bef}^l_{p_2,p_3} \rightarrow \mathit{bef}_{p_1,p_3} \right)$$

The last part of the planning problem to be encoded is the initial task network. It is inserted as the content of the zeroth layer of the decomposition tree by the following formula.

$$\mathit{initialTN} = \left(\bigwedge_{t\in T(tn_i)} \alpha(tn)(t)^0_t \wedge \mathit{used}^0_t \wedge \texttt{if } \alpha(tn)(t) \in C \texttt{ then } \mathit{abs}^0_t \texttt{ else } \neg \mathit{abs}^0_t \right) \wedge \left(\bigwedge_{t=|T(tn_i)|+1}^{|\pi|} \left(\bigwedge_{a\in A} \neg a^0_t \right) \wedge \neg \mathit{used}^0_t \wedge \neg \mathit{abs}^0_t \right) \wedge \bigwedge_{(t_b,t_a)\in \prec(tn_i)} \mathit{bef}^0_{t_b,t_a} \wedge \left(\bigwedge_{t\in T(tn_i)} \mathit{applyMethod}(0,t) \wedge \mathit{methodChildren}(0,t) \wedge \mathit{selectAction}(0,t) \right)$$

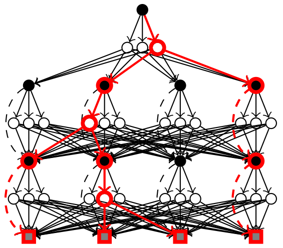


Figure 2: Structure of the constructed formula. A decomposition tree corresponding to the one in Fig. 1 is marked red.

Finally, the task sequence π itself is encoded by enforcing that the tasks in the last layer are the primitives of the solution, and that they are totally ordered as given in the sequence.

$$\mathit{solution} = \bigwedge_{p=1}^{|\pi|} \pi(p)^K_p \wedge \mathit{bef}^K_{p,p+1}$$

Figure 2 contains a simplified (e.g. omitting ordering constraints) visual representation of a verify SAT formula where $K = 3$, $|\pi| = 4$, and $|M| = 3$. The selected actions are not depicted, but the structure of the decomposition tree is highlighted in red. The depicted tree is the one presented in Figure 1. Here we see that for every abstract task a method (white circles) is chosen, leading to the appropriate children in the next layer, while every primitive task is directly inherited to the next layer (marked by the dashed lines).

The following theorem provides that the presented construction is in fact correct.

Theorem 1. *Given a* DECOMPSEQ *problem* $(\mathcal{P}, \pi)$ *and a bound* K*, then* $\mathcal{F}$ *constructed using the bound* K *is satisfiable if and only if* $\pi \in \mathit{Sol}(\mathcal{P})$ *and the height of* π*'s decomposition tree is at most* K*.*

Proofsketch: $\Rightarrow$ Let $\mathcal{F}$ be satisfiable. Then the valuation of $\mathcal{F}$ specifies for every abstract task a single method which should be applied to it, due to the constraint $\mathit{applyMethod}$. Further in any layer of the formula, there are exactly the children of the chosen method as enforced by $\mathit{methodChildren}$. The clauses from $\mathit{selectAction}$, $\mathit{mustBeChildOf}$, $\mathit{fatherMustExist}$ and $\mathit{childImpliesChildOf}$ ensure that no additional actions are contained in the formula. From these observations, we know that the asserted actions and their interconnections form a valid decomposition tree. Lastly, the leafs of the represented tree are the tasks contained in π. Using Proposition 1 of Geier and Bercher (2011), we know that π is a solution.
$\Leftarrow$ Let π be a solution to $\mathcal{P}$. Following the aforementioned theorem and the premises of this theorem, there is a valid decomposition tree whose leafs are π and whose height is at most K. We can construct a satisfiable valuation for $\mathcal{F}$ by assigning each node in the decomposition tree to its layer and arbitrarily to a position within that layer, which is not occupied by a primitive task in the previous layer. Any primitive task is also asserted at its position in all following layers. Using this assignment we can construct a satisfying valuation for $\mathit{selectAction}$, $\mathit{mustBeChildOf}$, $\mathit{fatherMustExist}$,

$childImpliesChildOf$ and $maintainPrimitive$. By tracing the ordering constraints contained in the decomposition tree, we can also create a valuation satisfying the $maintainOrder$ constraints. Since the decomposition tree is valid, it specifies a single decomposition method for every abstract task and the methods' subtasks are contained in the following layer, which ensures that our constructed valuation also fulfils the $methodChildren$ and $applyMethod$ clauses. As the leafs of the decomposition tree are the tasks of π (in the correct order) we have constructed a satisfying valuation for $\mathcal{F}$. □

5 Height of Decomposition Trees

The constructed formula is based on the value K – an upper bound to the height of a decomposition tree that can lead to the task sequence to be verified. Obviously, it is often possible to construct a tree of arbitrary height for a given solution, given there is a subnetwork in a method containing only a single task. The value of K can be limited to the smallest height necessary such that a decomposition tree leading to the solution exists. As a part of proving that VERIFYSEQ is in $\mathbb{NP}$ we proved[1] that if a task network tn is reachable from the initial task network tn_i there is always a sequence of $S = 2|T(tn)|(|C| + 1)$ decompositions which leads to that task network (Behnke, Höller, and Biundo 2015). S is clearly an upper bound, which we call K_{theo}, since in any layer of the decomposition tree at least a single decomposition method has to be applied.

A better bound can be derived based on the task schema transition graph (TSTG), which was implicitly introduced by Alford et al. (2016a). Its vertices are the tasks of the domain and it contains an edge (u, v) if u has a method containing the task v. If this graph is acyclic, the domain is called to be acyclic[2]. If so, we can use the length of the longest path l in the graph as an upper bound K_{TSTG}, since it is not possible to apply more than l decompositions in a row to the same task without reaching only primitive tasks.

Another bound is based on a compilation for HTN planning problems originally introduced by Höller et al. (2014). It removes all methods having subnetworks with less than 2 tasks in it. Afterwards, the number of tasks in each layer of the decomposition tree will grow by at least one (or all tasks are already primitive), since at least one method has to be applied in each layer. This gives the natural bound $K_{unit} = \frac{|\pi| - |T(tn_i)|}{\delta - 1}$ for the height of the tree, where δ is the size of the smallest method in the domain.

6 Evaluation

In order to show that our approach is feasible for verifying plans in existing HTN planning domains, we have implemented our transformation and conducted an empirical evaluation. The code of the implementation can be found at `https://www.uni-ulm.de/in/ki/panda`.

Contrary to classical planning, there is no well-established set of benchmark domains for hierarchical planning. Hence, we used several benchmark domains for *Hybrid Planning* (Biundo and Schattenberg 2001), a formalism which extends standard HTN planning in that it adds preconditions and effects to abstract tasks and allows for so-called causal links inside methods (McAllester and Rosenblitt 1991). For our evaluation, we have stripped both these extensions from all domains. As we have generated the solutions to be verified using the reduced problems, this reduction does not influence the evaluation. For more detailed descriptions of these domains, we refer to Bercher, Keen, and Biundo (2014). Further, we used the Monroe domain (Blaylock and Allen 2005), which is a benchmark domain from plan and goal recognition. It seems to be an appropriate challenge, as it has a large set of methods, actions, and constants. It seems well suited to test a plan verification system, since plan verification can be seen as a special case of plan recognition (where the observed action sequence has ended). For all test instances, we have generated an arbitrary solution which was subsequently verified. Since all our benchmark domains are given in a lifted formalism and our translation works on a grounded one, we have first grounded all actions and methods and pruned them if obviously possible (removing actions with preconditions occurring neither as effects nor in the initial state, abstract tasks without methods, and methods containing removed tasks). We have also removed all constants not contained in the solution to be verified from the problem when they cannot appear in free variables of any methods (i.e. parameters of the abstract task that do not occur in the subnetwork). In this case, they cannot possibly be part of any decomposition tree leading to the solution.

Table 1 summarises some properties for the grounded domains and the values of the three variants to bound K. It was not possible to compute K_{unit} for the Monroe domain, since the compiled domain would have contained too many methods. Similarly some SmartPhone instances have acyclic TSTGs, making the computation of K_{TSTG} for these instances impossible. We have denoted these cases with ∞ in the table. We were able to compute a suitably small bound K for all our benchmark domains. However, if we were to encounter a domain where unit compilation is not possible for practical reasons and whose TSTG is acyclic, we would have to fall back to the theoretical upper bound, which can be extremely high, but is always finite. For the evaluation we have used the SAT solver `cryptominisat5` (Soos 2016), which was amongst the top-performing SAT solvers at the SAT Competition 2016. The experiments were conducted on an Intel Xeon E5-2660 with 503 GB available RAM.

We were able to verify the plans of all our benchmark problems in less than 451 seconds, while only four instances took more than 100 seconds. As shown in Figure 3a, the time needed for verification correlates with both plan length and the number of actions in the domain, while neither is a dominating factor. If we look at the dependence of time of both of these values, we see that a combination of a long plan in a large grounded domain seems to result in a longer computation time (see Figure 4). We also tested whether it is possible to verify any of the solutions with a smaller value

[1] The paper erroneously states a bound of $|T(tn)|(|C| + 1)$, as we did not take decompositions into account that occurred after the task network has reached the size of tn.

[2] Plan verification for acyclic domains is still $\mathbb{NP}$-complete.

domain	K_{theo}		K_{unit}		K_{TSTG}		$\|L\|$		$\|C\|$		$\|A\|$		$\|M\|$	
	min	max	min	max	min	max	min	max	min	max	min	max	min	max
UMTranslog ●	70	1258	5	37	3	6	19	88	4	16	7	22	4	17
Satellite ●	20	510	5	17	1	4	8	70	1	17	7	78	11	541
SmartPhone ●	132	324	11	15	3	∞	44	47	5	8	16	18	14	99
Woodworking ●	12	48	2	4	1	2	32	59	1	4	6	24	4	76
Monroe ●	198	11032	∞	∞	4	8	1220	3152	32	265	436	6017	408	5476

Table 1: Height and statistics of the evaluation domains. We tested 21 UMTranslog instances, 22 Satellite instances, 3 SmartPhone instances, 5 Woodworking instances, and 50 Monroe instances.

of K to test the conciseness of our upper bounds. In total 11 of the Monroe instances were solvable with a bound up to 3 smaller than the computed K, while only one instance each from the SmartPhone and the Satellite domain were solvable with a bound 1 less than the computed one.

In a second evaluation we showed that the presented technique not only verifies solutions, but also refutes non-solutions. We generated non-solutions by randomly applying applicable actions until a plan of the same length as the solution to that problem had been reached. For every problem five such sequences were generated (505 in total). For all but five of the randomly generated plans, the verifier showed that they are non-solutions. The remaining five have been manually checked to be solutions. We also checked various non-solutions to be correctly classified. The random solutions have been excluded from the further evaluation, i.e., our diagrams for non-solutions do not include these datapoints. The time needed to refute these non-solutions was always less than 120 seconds, as depicted in Figure 3b. Interestingly – and contrary to usual expectations in SAT solving – it was usually easier to refute an incorrect plan than to show that solutions were indeed solutions. We suspect that this is due to certain combinations of primitive tasks which cannot occur together (usually caused by the decomposition hierarchy) in any solution. They form a small unsatisfiable subset of the formula's clauses, which the solver might detect early showing unsatisfiability.

Lastly, we have also evaluated our verifier on non-solutions which are similar to actual solutions (see Figure 3c). For every instance five "almost solutions" were generated by randomly selecting an action within a solution and replacing it with another random primitive action from the domain. This may result in an unexecutable plan, but since our formula does not check executability, we can reasonably assume that the results are identical to the case where the resulting plan would be executable. Our evaluation shows higher (up to 1123 seconds) runtimes for these "almost solutions", which is expected as the formulae for these task networks should be "almost satisfiable". Such SAT formulae are in general deemed hard. However all these non-solutions were refuted in time and most of them in an acceptably low timeframe – less than 300 seconds.

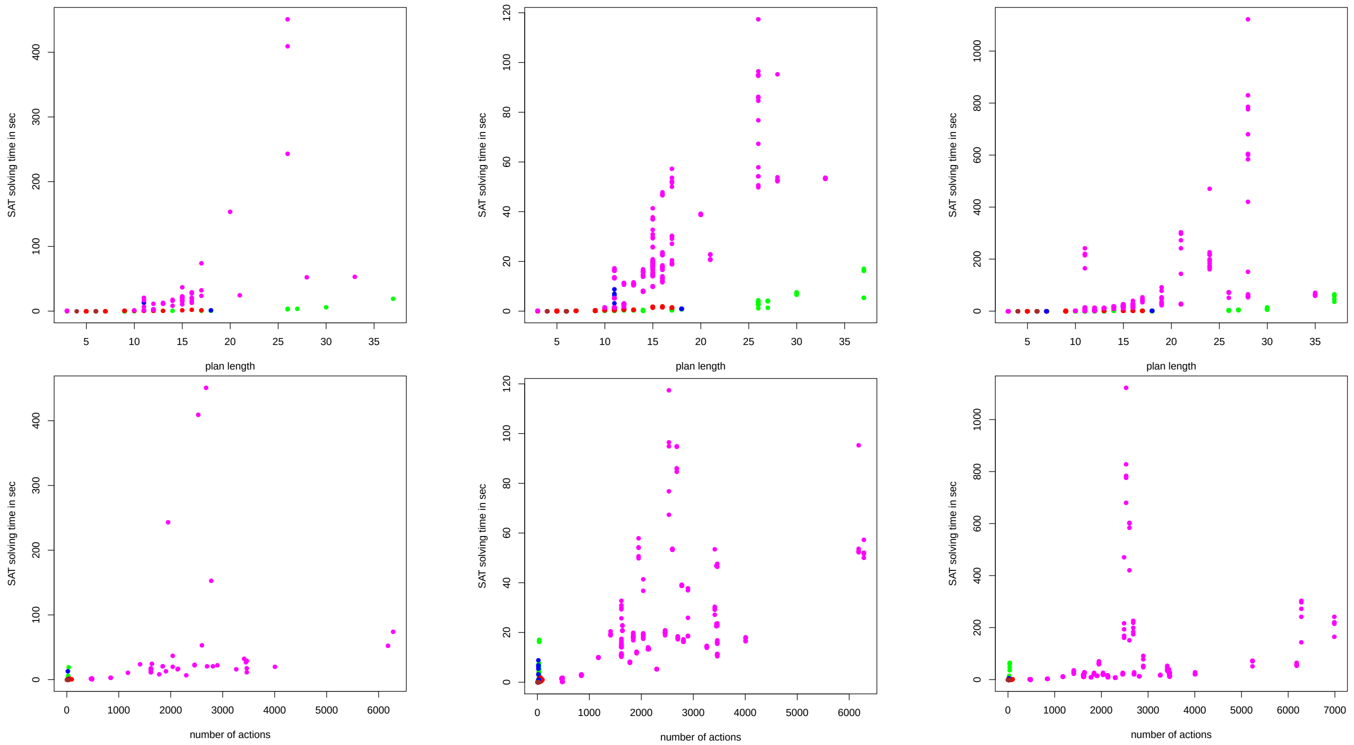
7 Related Work

The overall idea to use logic-based techniques in hierarchical planning is not entirely new. Most notably, there are two translations of planning problems into SAT (Mali and Kambhampati 1998) and Answer Set Programming (Dix, Kuter, and Nau 2003). However both differ significantly in the employed formalism (although both call it HTN planning for historic reasons). Mali and Kambhampati's formalism restricts the model by disallowing recursion (which we can handle completely), and simultaneously allowing for both task sharing (Alford et al. 2016b) and task insertion (Geier and Bercher 2011). All these features make their hierarchical planning language far less expressive and in conjunction with the fact, that their problems don't contain an initial task network, equivalent to classical planning. Dix, Kuter, and Nau on the other hand use a formalism which is equivalent to ours, with the sole additional restriction that the task networks in all methods must be totally ordered. As such, they are using the older formalism of SHOP, not the new one of SHOP2 which allows for partial order in methods. In the case of HTN plan verification, this restriction lowers the complexity of the problem from $\mathbb{NP}$ to $\mathbb{P}$, as the problem becomes equivalent with parsing a word of a context free grammar (Höller et al. 2014).

There has also been a lot of work in the community on verification of HTN planning domains and problems. Their objective for these investigations was not to determine whether a certain plan is a solution to a given planning problem, but rather to check some criterion (ususally ensuring internal consistency) of the model. Examples include the work of Goldman (2009), Biundo and Schattenberg (2001), McCluskey, Liu, and Simpson (2003), or Marthi, Russell, and Wolfe (2007) (for a survey see Bercher et al. (2016)).

Shivashankar et al. (2013) have proposed a new hierarchical planning formalism, called Hierarchical Goal Network (HGN) planning, which uses networks of sub-goals instead of networks of sub-tasks. In this formalism plan verification is presumably easier, but – so far – there have neither been theoretical or practical investigation for plan verification. Further HGNs are seemingly less expressive than HTNs, as every HGN can be compiled into an HTN, while the reverse is yet unknown and potentially impossible (Alford et al. 2016b). While this can be beneficial when designing planning algorithms, a plan verifier should always be able to handle the most general formalism, i.e., HTNs.

Other techniques for verifying HTN plans can be thought of. For example, one could also use the existing translation of HTNs into ConGolog (Shapiro, Lespérance, and Levesque 2002) and subsequently use one of the existing verifiers for ConGolog. However we deem such translations to be future work.



(a) Runtime on actual solutions.

(b) Runtime on non-solutions, generated by randomly applying actions until the same length as in Fig 3a was reached.

(c) Runtime on non-solutions, generated by replacing a single action in a solution.

Figure 3: For each sub-figure the top diagram shows the runtime in relation to the length of the plan, while the bottom one depicts the runtime in relation to the number of ground action in the domain. The colours encode the domain (see Table 1).

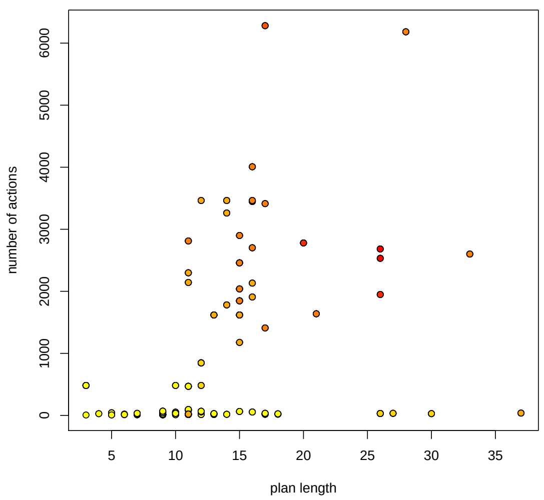


Figure 4: Length of the plan to be verified and number of grounded actions in the domain, where the time to verify the solution is indicated by the colour of the point. The red indicates higher and yellow lower times. The colour was determined according to the logarithm of the verification time.

8 Conclusion

In this paper we studied the practical aspects of the task of plan verification. It has a wide range of applications in planning-based systems, e.g., whenever it is necessary to post-optimise plans based on local-search, and is one of the requirements when conducting a planning competition.

We presented the first plan verifier for HTN plans, which solves the problem by translating it (due its $\mathbb{NP}$-completeness) into a SAT formula. We showed that our technique allows to verify HTN plans in reasonable time and is thus useable in practice. Although the solving times for the created formulae are already low, it might prove beneficial to further reduce the size of the formula or to investigate new structures that could be applied when constructing the formula. By combining our approach with methods from SAT-based classical planning, it might be possible to construct a SAT-based HTN planning system. Such a planner could subsequently also be used to implement requests to change a solution – based on the theoretical results of Behnke et al. (2016) – with a procedure different from local search. As such, this paper lays the groundwork for both the practical investigation of plan verification, as well as for a novel planning techniques for HTN planning.

9 Acknowledgements

We want to thank Pascal Bercher and the anonymous reviewers for their help to improve the paper. This work was done within the Transregional Collaborative Research Centre SFB/TRR 62 “Companion-Technology for Cognitive Technical Systems” funded by the German Research Foundation (DFG).

References

Ai-Chang, M.; Bresina, J.; Charest, L.; Chase, A.; Hsu, J.; Jonsson, A.; Kanefsky, B.; Morris, P.; Rajan, K.; Yglesias, J.; Chafin, B.; Dias, W.; and Maldague, P. 2004. MAPGEN: mixed-initiative planning and scheduling for the mars exploration rover mission. *Intelligent Systems, IEEE* 19(1):8–12.

Alford, R.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and Aha, D. W. 2016a. Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *Proc. of ICAPS*.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016b. Hierarchical planning: relating task and goal decomposition with task sharing. In *Proc. of IJCAI*.

Behnke, G.; Höller, D.; Bercher, P.; and Biundo, S. 2016. Change the plan - how hard can that be? In *Proc. of ICAPS*.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the complexity of HTN plan verification and its implications for plan recognition. In *Proc. of ICAPS*.

Bercher, P.; Biundo, S.; Geier, T.; Hörnle, T.; Nothdurft, F.; Richter, F.; and Schattenberg, B. 2014. Plan, repair, execute, explain - how planning helps to assemble your home theater. In *Proc. of ICAPS*.

Bercher, P.; Richter, F.; Hörnle, T.; Geier, T.; Höller, D.; Behnke, G.; Nothdurft, F.; Honold, F.; Minker, W.; Weber, M.; and Biundo, S. 2015. A planning-based assistance system for setting up a home theater. In *Proc. of AAAI*.

Bercher, P.; Höller, D.; Behnke, G.; and Biundo, S. 2016. More than a name? On implications of preconditions and effects of compound HTN planning tasks. In *Proc. of ECAI*.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proc. of SoCS*.

Biundo, S., and Schattenberg, B. 2001. From abstract crisis to concrete relief – a preliminary report on combining state abstraction and HTN planning. In *Proc. of ECP*.

Blaylock, N., and Allen, J. 2005. Generating artificial corpora for plan recognition. In *Proc. of UM*.

Dix, J.; Kuter, U.; and Nau, D. 2003. Planning in answer set programming using ordered task decomposition. In *Proc. of KI*.

Erol, K.; Hendler, J.; and Nau, D. 1994. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proc. of AIPS*.

Erol, K.; Hendler, J.; and Nau, D. 1996. Complexity results for HTN planning. *Annals of Mathematics and AI* 18(1):69–93.

Ferguson, G.; Allen, J.; and Miller, B. 1996. TRAINS-95: towards a mixed-initiative planning assistant. In *Proc. of AIPS*.

Fernández-Olivares, J.; Castillo, L.; García-Pérez, Ó.; and Palao, F. 2006. Bringing users and planning technology together. Experiences in SIADEX. In *Proc. of ICAPS*.

Frisch, A.; Peugniez, T.; Doggett, A.; and Nightingale, P. 2005. Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. *Journal of Automated Reasoning (JAR)* 35(1-3):143–179.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proc. IJCAI*.

Gerevini, A., and Serina, I. 2002. LPG: A planner based on local search for planning graphs with action costs. In *Proc. of AIPS*.

Goldman, R. 2009. A semantics for HTN methods. In *Proc. of ICAPS*.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proc. of ECAI*.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proc. of ICAPS*.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Proc. of ICTAI*.

Kautz, H. A.; Selman, B.; and Jiang, Y. 1997. A general stochastic approach to solving problems with hard and soft constraints. *Satisfiability Problem: Theory and Applications* 35:573–586.

Mali, A., and Kambhampati, S. 1998. Encoding HTN planning in propositional logic. In *Proc. of AIPS*.

Marthi, B.; Russell, S.; and Wolfe, J. 2007. Angelic semantics for high-level actions. In *Proc. of ICAPS*.

McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proc. of AAAI*.

McCluskey, T.; Liu, D.; and Simpson, R. 2003. GIPO II: HTN planning in a tool-supported knowledge engineering environment. In *Proc. of ICAPS*.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J.; Wu, D.; and Yaman, F. 2003. SHOP2: an HTN planning system. *Journal of AI Research (JAIR)* 20:379–404.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Wu, D.; Yaman, F.; Muñoz-Avila, H.; and Murdock, J. 2005. Applications of SHOP and SHOP2. *Intelligent Systems, IEEE* 20:34–41.

Shapiro, S.; Lespérance, Y.; and Levesque, H. 2002. The cognitive agents specification language and verification environment for multiagent systems. In *Proc. of AAMAS*.

Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. S. 2013. Hierarchical goal networks and goal-driven autonomy: Going where AI planning meets goal reasoning. In *ACS Workshop*.

Soos, M. 2016. The CryptoMiniSat 5 set of solvers at SAT Competition 2016. In *Proc. of SAT Competition 2016*.