A Novel Parsing-based Approach for Verification of Hierarchical Plans

Roman Barták*, Simona Ondrčková*, Adrien Maillard†, Gregor Behnke‡ and Pascal Bercher§
*Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
Email: {bartak,ondrckova}@ktiml.mff.cuni.cz
†Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA
Email: adrien.maillard@jpl.nasa.gov
‡Faculty of Engineering, University of Freiburg, Freiburg, Germany
Email: behnkeg@informatik.uni-freiburg.de
§College of Engineering & Computer Science, The Australian National University, Canberra, Australia
Email: pascal.bercher@anu.edu.au

***Abstract*—Hierarchical Task Networks were proposed as a method to describe plans by decomposition of tasks to sub-tasks until primitive tasks, actions, are obtained. Valid plans – sequences of actions – must adhere both to causal dependencies between the actions and to the structure given by the decomposition of the goal task. Plan verification aims at finding if a given plan is valid, that is, if it is causally consistent and it can be obtained by decomposition of some task. The paper describes a novel parsing-based approach for hierarchical plan verification that is orders of magnitude faster than existing methods.**

***Index Terms*—hierarchical task networks, plan verification, parsing, attribute grammars**

I. Introduction

Hierarchical planning is a practically important approach to automated planning based on encoding abstract plans as *hierarchical task networks* (HTNs) [1], [2]. The network describes how compound tasks are decomposed, via decomposition methods, to sub-tasks and eventually to actions forming a plan. Depending on the chosen HTN formalization, the decomposition methods may specify additional constraints among the sub-tasks such as partial ordering, state constraints, or causal links. The obtained sequence of actions must be executable in the classical sense, but also adhere to the constraints specified by the decomposition methods.

Plan verification deals with the problem of determining whether a given plan is valid with respect to a given model. In other words, the question to answer is if the given plan can be generated from some task using the decomposition methods defined in the domain model. Automated plan verification is important for various reasons. First, it is required for planning competitions to ensure that plans returned by competing planners are indeed correct (there might be many valid plans to achieve a given goal and not all these plans may be known in advance). Next, when designing a new planner, one may need to independently check whether the planner returns correct plans before the planner is deployed (obviously, verifying several plans does not guarantee correctness of the planner but it gives some trust to the planner). Last but not least, when designing a new domain model while having a set of existing plans, plan verification can be used to check if the model complies with the given set of plans.

Research is supported by the Czech Science Foundation under the project P103-18-07252S.

Part of this research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004).

Some of the work by Gregor Behnke and Pascal Bercher was already done while at the Insitute of Artificial Intelligence, Ulm University, Germany.

While verifying classical plans is not a computationally hard task and there exists a widely-used classical plan verifier [3], the situation is very different with hierarchical plan verification. It is known that the hierarchical plan verification problem is computationally expensive, as it belongs to the NP-complete problems [4], [5]. As of this writing, there exist only two systems for verifying if a given plan complies with the HTN model, that is, if a given sequence of actions can be obtained by decomposing some task. One system is based on transforming the verification problem to SAT [6] and the other system is using parsing of attribute grammars [7].

Parsing became popular in solving the *plan recognition problem* [8] as researchers realized soon the similarity between hierarchical plans and formal grammars [9]–[11], specifically context-free grammars with parsing trees being close to decomposition trees from HTN planning. The HTN plan recognition problem can be formulated as the problem of adding a sequence of actions after some observed partial plan such that the joint sequence of actions forms a complete plan generated from some task (more general formulations also exist). Hence plan recognition can be seen as a generalization of plan verification. There exist numerous approaches to plan recognition using parsing or string rewriting [12]–[16], but they use hierarchical models that are weaker than the full expressive power of HTN planning. In particular, grammar-based models are frequently exploiting context-free grammars (as parse trees are somehow similar to decomposition trees, as mentioned above). Notably, these models assume that the order of the action symbols in the rules of the grammar is the same as in the final plan. Consequently, these grammar-based approaches do not support task interleaving [11] and various more elaborate HTN constraints, such as state constraints. The (formal) languages defined by HTN planning problems (with partial-order, recursion, and simple STRIPS preconditions and effects, but even without state constraints) lie somewhere between context-free (CF) and context-sensitive (CS) languages [9]. So to model even the most expressive HTN problems with formal grammars, one needs to go beyond the CF grammars. Currently, the only grammar-based model that fully covers HTNs uses attribute grammars [11] – which differentiate between the order of symbols in rules and the order implied on the final plan.

In this paper, we focus on verification of HTN plans using parsing. The uniqueness of the proposed method is that it covers the full expressive power of HTN planning, including task interleaving (i.e., partial order of sub-tasks), empty and recursive methods, and state constraints. Regarding the expressive power, we extended the previous parsing-based approach [7] by support of empty tasks and due to a more simplistic, yet elaborate algorithm design, we are several orders of magnitude faster. The major novelty lies in the way when and how the decomposition constraints are verified. In particular, all the decomposition constraints are verified immediately when a new task is added, so the algorithm only derives tasks that decompose to a set of actions in the verified plan. The new technique also uses simpler and more efficient data structures (Boolean arrays instead of complex timelines), which makes it easier to implement and, as we shall show experimentally, also practically more efficient.

The paper is organized as follows. We will first formally introduce necessary notions from planning and hierarchical task networks and formally define the verification problem. Then we will describe the verification algorithm and prove its properties such as soundness, completeness and worst-case time and space complexities. Finally, we will empirically compare this novel approach with existing HTN plan verification techniques and justify the theoretical expectation that the novel technique is indeed dominating all previous ones in terms of efficiency.

II. Background on Planning

Hierarchical planning combines information about states, i.e., the causal precondition/effect relations between actions with specific plan structures expressed via task decomposition methods. In this section we shall formally introduce these models, namely classical STRIPS planning and hierarchical task networks.

A. Classical Planning

Classical STRIPS planning [17] deals with sequences of actions transferring the world from a given initial state to a state satisfying certain goal conditions. World states are modelled as sets of propositions that are true in those states, and actions are modelled to change the validity of certain propositions. Formally, let P be a set of all propositions modelling properties of world states. Then a state $S \subseteq P$ is a set of propositions that are true in that state (every other proposition is false). Later, we will use the notation $S^+ = S$ to describe explicitly the valid propositions in the state S and $S^- = P \setminus S$ to describe explicitly the propositions not valid in the state S.

Each action a is described by three sets of propositions (B_a^+, A_a^+, A_a^-), where $B_a^+, A_a^+, A_a^- \subseteq P, A_a^+ \cap A_a^- = \emptyset$. The set B_a^+ describes positive preconditions of action a, that is, propositions that must be true right before the action a. Some modeling approaches allow also negative preconditions, but these preconditions can be compiled away [18]. For simplicity reasons we assume positive preconditions only (the techniques presented in this paper can also be extended to cover negative preconditions directly, but the compilation technique for STRIPS works in the HTN setting as well [19]). Action a is applicable to state S iff $B_a^+ \subseteq S$. Sets A_a^+ and A_a^- describe positive and negative effects of action a, that is, propositions that will become true and false in the state right after executing the action a. If an action a is applicable to state S then the state right after the action a is:

$$\gamma(S, a) = (S \setminus A_a^-) \cup A_a^+. \tag{1}$$

$\gamma(S, a)$ is undefined if an action a is not applicable to state S.

The classical planning problem, also called a STRIPS problem, consists of a set of actions A, a set of propositions S_0 called an initial state, and a set of goal propositions G^+ describing the propositions required to be true in the goal state (again, negative goal is not assumed as it can be compiled away). A solution to the planning problem is a sequence of actions $a_1, a_2, \ldots, a_n$ such that $S = \gamma(...\gamma(\gamma(S_0, a_1), a_2), ..., a_n)$ and $G^+ \subseteq S$. This sequence of actions is called a *plan*.

The *plan verification problem* is formulated as follows: given an initial state S_0, a sequence of actions $a_1, a_2, \ldots, a_n$, and goal propositions G^+, does the sequence of actions form a valid plan leading from S_0 to a goal state? This problem is addressed for example by the VAL system [3] and our method does this classical plan verification during pre-processing, where we calculate all the intermediate states.

B. Hierarchical Task Networks

To simplify and speed up the planning process, several extensions of the basic STRIPS model were proposed to include some control knowledge. Hierarchical Task Network Planning [1] was proposed as a planning framework that includes control knowledge in the form of recipes on how to solve specific tasks. Our formalization is loosely based on the formalization by Erol et al. [1]. The recipe is represented as a task network, which is a set of sub-tasks to solve a given task together with the set of constraints between the sub-tasks. Let T be a compound task and $(\{T_1, ..., T_k\}, C)$ be a task network, where C are its constraints (see later). We can describe the decomposition method as a derivation (rewriting) rule saying that T decomposes to sub-tasks $T_1, ..., T_k$:

$$T \rightarrow T_1, ..., T_k \quad [C]$$

Note that the order of tasks on the right side of the rule does not matter (opposite to rewriting rules in grammars) as the order is explicitly described by the precedence constraints in C.

HTN planning problems are specified by an initial state (the set of propositions that hold at the beginning) and by an initial task representing the goal. This goal task needs to be decomposed via decomposition methods until a set of primitive tasks – actions – is obtained. Moreover, these actions need to be linearly ordered to satisfy all the constraints obtained during decompositions and the obtained plan – a linear sequence of actions – must be applicable to the initial state in the same sense as in classical planning. We denote an action as a_i, where the index i means the order number of the action in the plan (a_i is the i-th action in the plan). The state right after the action a_i is denoted S_i, while S_0 is the initial state. We denote the set of actions to which a task T decomposes as $act(T)$. If U is a set of tasks, we define $act(U) = \cup_{T \in U} act(T)$. The index of the first action in the decomposition of T is denoted $start(T)$, that is, $start(T) = min\{i | a_i \in act(T)\}$. Similarly, $end(T)$ means the index of the last action in the decomposition of T, that is, $end(T) = max\{i | a_i \in act(T)\}$.

We can now define formally the constraints C used in the decomposition methods. They are the ones available in the HTN formalization by Erol et al. [1]. The constraints can be of the following three types, where the first is also known as an ordering constraint and the latter two are essentially state constraints:

- $t_1 \prec t_2$: a precedence constraint meaning that in every plan the last action obtained from task t_1 is before the first action obtained from task t_2, $end(t_1) < start(t_2)$,
- *before*(U, p): a precondition constraint meaning that in every plan the proposition p holds in the state right before the first action obtained from tasks U, $p \in S_{start(U)-1}$,
- *between*(U, V, p): a prevailing constraint meaning that in every plan the proposition p holds in all the states between the last action obtained from tasks U and the first action obtained from tasks V,
$\forall i \in \{end(U), \dots, start(V) - 1\}, p \in S_i$.

The precedence constraints are used to define a specific order of sub-tasks. As an example consider the *deliver* task. A decomposition method for it is depicted in Fig. 1. The task *deliver* may decompose to four sub-tasks: we first *move* the robot to a location, where we *load* the item to a robot, then the robot *moves* to the destination, where we *unload* the item. The precondition constraints are used in the same way as for actions. For example, before the robot starts moving to the place where the item is located, we may require the robot to be empty (note that this is not a precondition of task move, but it may be required for the move task used in the context of item delivery). The prevailing constraint is used to maintain some property between sub-tasks. For example, we may require the item to be loaded in the robot all the time between load and unload sub-tasks. Though the prevailing constraint was part of the original definition of HTN planning [1], we are not aware

deliver $\rightarrow$ t_1:*unload*, t_2:*move*, t_3:*move*, t_4:*load*
[
$t_2 \prec t_4$, $t_4 \prec t_3$, $t_3 \prec t_1$,
before($\{t_2\}$,robot_empty),
between($\{t_4\}$,$\{t_1\}$, package_loaded)
]

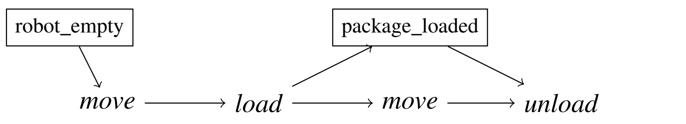

Fig. 1. A decomposition method for *deliver*. First we depict the method as written in terms of an Attribute Grammar rule. Note that the order of the tasks one the right hand side of the formula is irrelevant – and in this case completely different from the actually imposed order. This order is always and solely determined by the constraints shown in the square brackets below the rule. At the bottom we depict the method's task network (i.e. the rules right-hand side) in terms of a DAG. Tasks are names without boxes, while the state constraints are depicted by boxes. The tasks they refer to are indicated with arrows.

about any HTN system that supports this constraint. Still, we see this constraint practically very useful as demonstrated in the example above. Task interleaving, as mentioned before, allows the robot to do other tasks when doing the delivery task. For example, when going to the destination of the item, the robot may visit another location, take a sample there, and then deliver the sample after delivering the item. Hence actions for the delivery task and for the take-sample task interleave in the final plan.

Let us now look at how the *move* task can be realized. If the robot is not at the required location, the move task may decompose to a *drive* action, which moves the robot to some neighboring location, followed by a *move* sub-task describing the movement from that neighboring location to the destination. This shows that the decomposition methods can be recursive. On the other hand, if the robot is already at the required location, the move task is accomplished without using any action. This can be described by an *empty decomposition method* (empty task):

$$T_e \rightarrow \epsilon \quad [C]$$

For such methods, the only useful decomposition constraint C is a special form of the before constraint – *before*(p) – that requires some property p to be true at the world state, where the task T_e is supposed to be applied. As $act(T_e) = \emptyset$, we define values of $start(T_e)$ and $end(T_e)$ as follows. Let S_i be the state such that $p \in S_i$ (the before constraint holds there), then $start(T_e) = end(T_e) = i + 0.5$. This places the empty task T_e between actions a_i and a_{i+1} and allows T_e to participate in other decomposition constraints in its parent task.

Note finally that a task may have several decomposition methods describing alternative ways how to fulfill the task.

The *HTN plan verification problem* is formulated as follows: Given a sequence of actions $a_1, a_2, \dots, a_n$ and an initial state S_0, is the sequence of actions a valid classical plan applicable to S_0 and obtained from some compound task? Hence the HTN plan verification problem includes the classical plan verification problem. Additionally, the sequence of actions must be obtained by decomposing some task and all its sub-tasks (no extra action is allowed) and the action sequence must satisfy all the constraints C from methods used in the decomposition.

As mentioned above, there exist only two systems for HTN plan verification. The pioneering PANDA system [6] does not support the before and between constraints and requires the goal task to be given at input as well (the before constraints can be compiled away). The parsing-based system [7] supports fully the above definition of HTN except the empty methods (they can be compiled away in the domain model). We shall now describe a novel parsing-based approach for HTN plan verification that significantly improves efficiency over the existing techniques.

III. The Plan Verification Algorithm

The existing parsing-based HTN verification algorithm [7] uses a complex structure of a timeline. This structure maintains the decomposition constraints C so that they can be checked when composing sub-tasks to a compound task. As a consequence, these constraint checks are done repeatedly (anytime when two timelines are merged) and, moreover, possible conflicts are discovered later.

We propose a novel verification method that does not require the complex structure of a timeline, as the method checks all the constraints directly in the input plan. We first calculate all the intermediate states so the before and between constraints can be checked directly in these states rather than in the timeline. This makes the algorithm much easier to implement and also much faster (see evaluation).

The novel hierarchical plan verification algorithm is shown in Algorithm 1. It first calculates all intermediate states (lines 2-6) by propagating information from the initial state through the actions. At this stage, we actually solve the classical plan validation problem as the algorithm verifies that the given plan is causally consistent (action preconditions are provided by previous actions or by the initial state). The original verification algorithm did this calculation repeatedly each time it composed a compound task.

When the states are calculated, we apply a parsing algorithm to compose tasks. Parsing starts with the set of primitive tasks (line 7), each corresponding to an action from the input plan. Among the initial tasks we also include all empty tasks (line 8), that is, the tasks with an empty decomposition method, whose before constraint is satisfied at some state. For each task T, we keep a data structure describing the set $act(T)$, that is, the set of actions to which the task decomposes. We use a Boolean vector I of the same size as the plan to describe this set; $a_i \in act(T) \Leftrightarrow I(i) = 1$. To simplify checks of decomposition constraints, we also keep information about the index of first and last actions from $act(T)$. Together, the task is represented using a quadruple (T, b, e, I) in which T is a task, b is the index in the plan of the first action (begin) generated by T, e is the index in the plan of the last action (end) generated by T (we say that $[b, e]$ represents the interval of actions over which T *spans*), and I is a Boolean vector as described above.

```
Data: a plan $\mathbf{P} = (a_1, ..., a_n)$, an $InitialState$, and a
      set of decomposition methods (domain model)
Result: a Boolean equal to true if the plan can be derived
        from some compound task, false otherwise
1  Function VerifyPlan
2      $S_0 = InitialState$
3      for $i$ = 1 to $n$ do
4          if $\neg(\mathrm{pre}(a_i) \subseteq S_{i-1})$ then
5              return false
6          $S_i = (S_{i-1} \setminus \mathrm{eff}^-(a_i)) \cup \mathrm{eff}^+(a_i)$
7      sp $\leftarrow \emptyset$; new $\leftarrow \{(A_i, i, i, I_i) \mid i \in 1..n\}$
       Data: $A_i$ is a primitive task corresponding to action
             $a_i$, $I_i$ is a Boolean vector of size $n$, such that
             $\forall i \in 1..n, I_i(i) = 1, \forall j \neq i, I_i(j) = 0$
8      new $\leftarrow$ new $\cup \{(T_e, i + 0.5, i + 0.5, I_e) \mid T_e \rightarrow$
       $\epsilon[before(p)], p \in S_i\}$
       Data: $T_e$ is an empty task, $I_e$ is a Boolean vector of
             size $n$, such that $\forall i \in 1..n, I_e(i) = 0$
9      while new $\neq \emptyset$ do
10         sp $\leftarrow$ sp $\cup$ new; new $\leftarrow \emptyset$
11         foreach decomposition method $R$ of the form
            $T_0 \rightarrow T_1, ..., T_k$ $[\prec, \mathrm{pre}, \mathrm{btw}]$ such that
            $\{(T_j, b_j, e_j, I_j) \mid j \in 1..k\} \subseteq$ sp do
12             if $\exists(i, j) \in \prec: \neg(e_i < b_j)$ then
13                 continue with the next method
14             $b_0 \leftarrow \min\{b_j \mid j \in 1..k\}$
15             $e_0 \leftarrow \max\{e_j \mid j \in 1..k\}$
16             for $i$ = 1 to $n$ do
17                 $I_0(i) \leftarrow \sum_{j=1}^{k} I_j(i)$;
18                 if $I_0(i) > 1$ then
19                     continue with the next method
20             if $\exists(U, p) \in \mathrm{pre} : p \notin S_{\min\{b_j \mid j \in U\}-1}$ then
21                 continue with the next method
22             if $\exists(U, V, p) \in \mathrm{btw}\ \exists i \in \max\{e_j \mid j \in$
                $U\}, \ldots, \min\{b_j \mid j \in V\} - 1 : p \notin S_i$ then
23                 continue with the next method
24             new $\leftarrow$ new $\cup \{(T_0, b_0, e_0, I_0)\}$
25             if $\forall k, I_0(k) = 1$ then
26                 return true
27     return false
```

Algorithm 1: Parsing-based HTN plan verification

The algorithm applies each decomposition rule to compose a new task from the already known sub-tasks (line 11). Note that we do grounding (all attributes of tasks are instantiated by some constant from the initial state), so from the same abstract rule we may get several tasks T_0 with different attributes that span over different sets of actions. The composition consists of merging the sub-tasks, when we check that every action in the decomposition is obtained from a single sub-task (line 16), that is, $act(T_0) = \bigcup_{j=1}^{k} act(T_j)$ and $\forall i \neq j : act(T_i) \cap act(T_j) = \emptyset$. We also check all the decomposition constraints; the pseudo-code is a direct rewriting of constraint definitions (actually, their negations). If all tests pass, the new task is added to a set of tasks (line 24). Then we know that the task decomposes to actions, which form a sub-sequence (not necessarily continuous) of the plan to be verified. If any constraint check is violated, the algorithm continues with another instance of the decomposition method (or another decomposition method). Hence, the algorithm greedily finds all the (ground) tasks that decompose to some already known sub-tasks. The process is repeated until a task that decomposes to all actions is obtained (line 26) or no new task can be composed (line 9).

Proposition 1 (Soundness and completeness)**.** *The Algorithm 1 is a sound and complete technique for HTN plan verification.*

Proof. The algorithm is *sound* as if it finishes with success then there exists a task that decomposes to all actions in the input plan and that plan is causally consistent. If the algorithm finishes with the value **false** then the input plan is either causally inconsistent (line 5) or no other task can be derived (line 27). As the input plan is of a finite length, therefore there is a finite number of possible tasks (even if recursive rules exist) that decompose to a subset of actions in the original plan. Hence, all these tasks are eventually found and the algorithm stops (lines 9 and 27), so the method is *complete*. ☐

Proposition 2 (Time and space complexity)**.** *The worst-case time and space complexity of Algorithm 1 is $O(m \cdot 2^n)$, where the m is a number of grounded tasks and n is a number of actions in the input plan.*

Proof. In the worst case, the algorithm generates all grounded tasks that decompose to some subset of actions in the input plan. As the number of actions is n there is as many as 2^n subsets of actions. The same subset of actions might be generated by different tasks. So in the worst case, the set **sp** will store as many as $m \cdot 2^n$ elements. ☐

Recall that the problem of verifying HTN plans is NP-complete [4], [5] so the exponential time complexity is inevitable in the worst-case (unless P=NP). In comparison with the original parsing method, if the new algorithm generates some task then the original algorithm generates that task as well. However, as the new algorithm checks the decomposition constraints immediately when it introduces the task, the original algorithm checks these constraints when merging the timelines, so the original algorithm may introduce more tasks that are proved later to be inconsistent with some constraint.

IV. Example

In this section, we will present an example of parsing-based HTN plan verification. The example also demonstrates task interleaving – actions generated from different tasks may interleave to form a plan. This is the property that parsing techniques based on CF grammars cannot handle.

Assume that a complete plan consisting of actions $a_1, a_2, \ldots, a_7$ is given together with a set of decomposition methods (Figure 2). For simplicity, grounded tasks with no attributes are used. In the first iteration, the parsing algorithm composes tasks T_2, T_3, T_4 (in some order) as these tasks decompose to actions directly. One may see that actions from these tasks interleave in the plan. For each task, Figure 2 shows to which actions the task decomposes (roughly speaking, these are the timelines from the original algorithm) as well as the Boolean vector that is used to represent these actions in the novel algorithm. In the second iteration, only the task T_1 is composed from already known tasks T_3 and T_4. Finally, in the third iteration, tasks T_1 and T_2 are merged to a new task T_0 and the algorithm stops there as the final task spans over the whole plan.

Let us assume that there is a constraint *between*$(\{a_1\}, \{a_3\}, p)$ in the decomposition method for T_3. For example, this constraint may model a causal link[1] between a_1 and a_3. When composing the task T_3, the novel algorithm checks this constraint immediately in the states of the original plan. The original algorithm [7] stores the proposition p in the so-far empty second slot of task T_3. When T_3 and T_4 are merged, the algorithm checks that p can still be in the slot, in particular, that p is not required to be false at the same slot. The same check will be done again when merging T_1 and T_2 (this is the time when the

[1]Causal links are used in some Hierarchical Planning formalizations [5] and they encode that some action's precondition is achieved by some other (preceding) action's effect. No other action with a negating effect may be moved in between, so the link "protects" its condition.

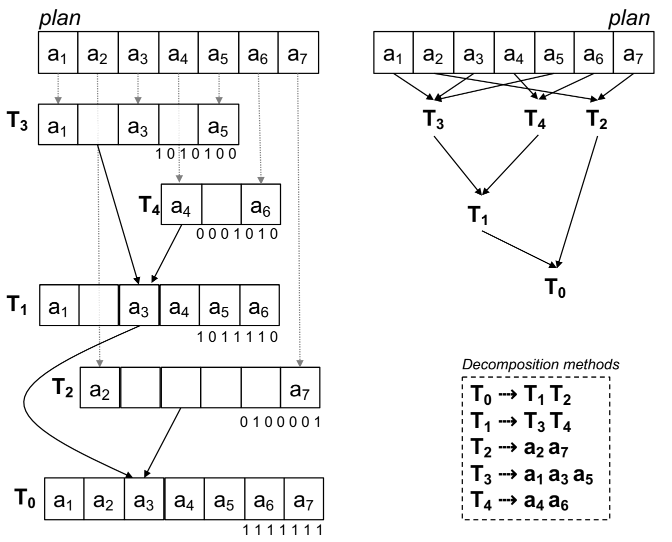

Fig. 2. Example of parsing-based plan verification (the right side shows the decomposition tree with the decomposition rules below it; the left side shows the tasks with timelines and filled slots)

algorithm finally fills the slot with action a_2). So the original verification algorithm checks the constraints repeatedly, while the novel algorithm does the check just once. Hence, it may happen that the original algorithm introduces a task that the new algorithm refuses as some decomposition constraint is violated.

V. Empirical Evaluation

We compared the proposed verification technique empirically with the original parsing-based technique [7], which achieved best so-far performance, and with the PANDA verifier [6]. All the experiments run under 64-bit Windows 10 on Intel Core i7 7700 processor and 16GB RAM.

The original parsing-based verification algorithm was implemented in Ruby 2.5 and it uses syntax of the SHOP2 and SHOP3 planners [20], [21] to specify the domain models, problems, and plans. Our new verification algorithm[2] is implemented in C# 7 (from .NET 4.7) and it uses the new PDDL-like representation HDDL [22] that is used by the PANDA planners and verifier.

In the first experiment, we compared all three systems using 13 instances from two planning domains, namely Transport and Satellite. The first ten of these instances were also used in the paper proposing the original parsing-based verification algorithm [7]. We added three larger instances to demonstrate the efficiency gap between the methods. In the Transport domain, the plan size increases from 2 up to 14 actions. In the Satellite domain, the plan size increases from 2 up to 19 actions. Invalid instances were created by removing a decomposition method from the domain model. In total, we evaluate the verification algorithms over 52 unique instances. Figure 3 shows the comparison of runtimes of all existing techniques. It confirms that the new verification technique outperforms significantly (orders of magnitude) the original technique (notice the logarithmic scale) as well as the PANDA verifier. Though Ruby is known to be slower than C#, the significant runtime improvement cannot be attributed to difference between programming languages only.

[2]The code is available at: https://github.com/siprog/HTNPlanValidation

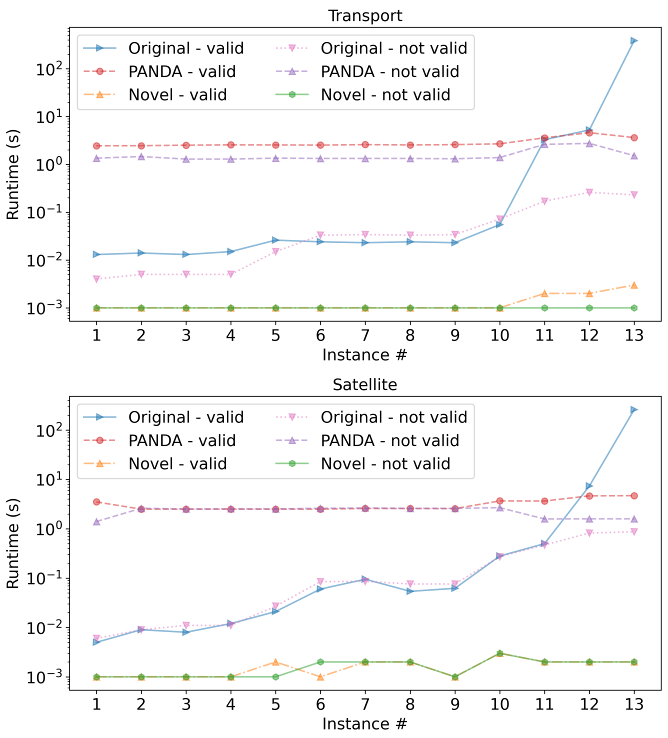

Fig. 3. Comparison of runtimes (seconds, logarithmic scale) for the plan verification task.

Next, we compared our new technique with the PANDA verifier using six additional domains: PCP, UMTranslog, Kitchen, Woodworking, Monroe and Rover used to evaluate the PANDA verifier [6]. The original parsing-based verification algorithm in Ruby was not able to handle these domains due to various reasons (such as missing support for empty methods and partial order of tasks). The plan lengths were 10 to 30 actions for the PCP domain, 7 to 26 actions for UMTranslog, 16 to 49 for Kitchen, 3 to 18 for Woodworking, 5 to 29 for Monroe, and 17 to 99 for Rover. We used a runtime limit of 300 seconds. Results are reported in Figure 4. The empirical results confirm that the novel parsing-based verifier outperforms the PANDA verifier with one exception. In the Monroe domain, PANDA is faster for larger instances (but some instances were not solved by PANDA). The Monroe domain is using empty methods/tasks and the parsing-based verifier greedily generates a lot of tasks, which decreases its performance. The PANDA verifier on the other hand was not designed to handle domains with empty methods. In such domains, a task network with n tasks can be decomposed into a plan with less than n actions – which is normally not possible. By construction, the PANDA verifier cannot handle such a case and is thus unable to verify an instance if such a situation occurs in any possible derivation of the plan. This is the case in two instances of the Monroe domain. Figure 4 omits the respective data point.

To evaluate scalability of the proposed technique, we added a few more longer plans to the Satellite and Transport domains (the original verification approach cannot handle them within the 10 minutes cutoff time) and we also tried plans of various lengths from four other domains: Monroe and Kitchen which are benchmarks for plan recognition [23] and UMTranslog and Woodworking – common HTN benchmark domains. In previous experiments, we used modified versions of Monroe and Kitchen domains as the PANDA verifier does not support some features of them such as method preconditions. In this experiment, we use full versions of the domain models. Figure 5 shows the dependence of runtime on the plan length for valid plans from these six domains. It demonstrates that the domain model influences the runtime significantly. For the parsing-based verification, the runtime depends mainly on the number of tasks that decompose to subplans of the input plan. The Monroe domain contains empty methods, which significantly increases the number of tasks generated. Similarly, the Woodworking domain contains tasks with shared parameters, which increases the number of grounded tasks (tasks with all attributes instantiated) so again many tasks are generated during parsing. This explains the larger runtime in comparison with other domains.

VI. Conclusions

In the paper we proposed a novel technique for verifying hierarchical plans by parsing. Like the previous parsing-based verification approach [7], this method covers HTN models fully including recursive tasks, partially-ordered tasks, task interleaving and various decomposition constraints, specifically the prevailing condition. Recursive methods as well as empty methods are also supported. The major innovation with respect to the existing parsing-based verification approach is checking the decomposition constraints directly in the plan and immediately, when a new task is added. The consequence is that we never add/explore more tasks than the previous approach and we can use a much simpler data structure (Boolean array) to keep information about actions generated from the task. Moreover, the Ruby implementation of the previous parsing approach supported only total order of tasks, so the new approach is first one that implements all above features. We showed experimentally that the novel technique is orders-of-magnitude more efficient than the previous parsing-based approach [7] and a SAT-based approach [6].

Hierarchical plan verification is still a computationally challenging task. The parsing-based approaches greedily generate tasks that span over some actions in the input plan – they use a bottom-up approach. The advantage is that they can find any task that decomposes to a given plan (rather than requiring information about the goal task in the input). The disadvantage is that many irrelevant tasks might be introduced. The open question is if some heuristics can be incorporated to prefer composition of specific tasks or if information about candidate goal tasks may be used to eliminate some tasks that are not "reachable" from any goal task.

All current hierarchical plan verifiers provide only binary output – either the plan is valid or invalid with respect to the domain model. In case the plan is invalid, the major research challenge is finding the source of invalidity. Note that the plan is validated with respect to the domain model. Hence, the bug can be in the plan but also in the model of decomposition methods. When the bug is identified, another research challenge is how to automatically repair it.

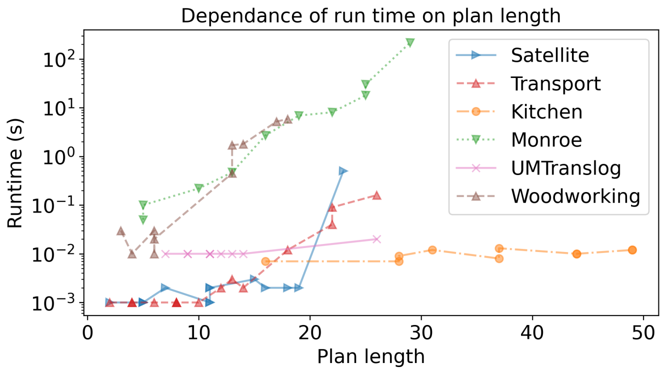

Fig. 5. Dependence of runtime (seconds, logarithmic scale) on plan length for the plan verification task.

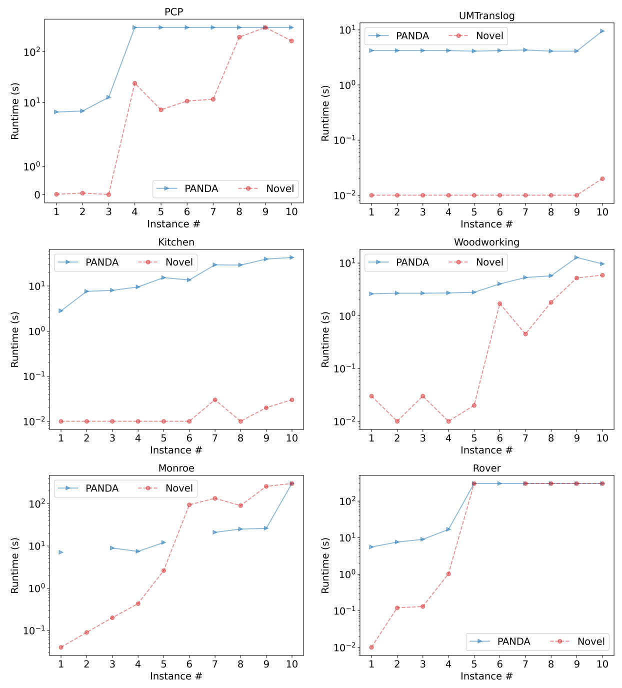

Fig. 4. Comparison of runtimes (seconds, logarithmic scale) for the plan verification task on other domains (using valid plans).

References

[1] K. Erol, J. A. Hendler, and D. S. Nau, "Complexity Results for HTN Planning," *Annals of Mathematics and Artificial Intelligence*, vol. 18, no. 1, pp. 69–93, 1996.
[2] P. Bercher, R. Alford, and D. Höller, "A survey on hierarchical planning – one abstract idea, many concrete realizations," in *Proc. of IJCAI*. IJCAI, 2019, pp. 6267–6275.
[3] R. Howey and D. Long, "VAL's Progress: The Automatic Validation Tool for PDDL2.1 used in the International Planning Competition," in *Proc. of the ICAPS'03 Workshop on the Competition: Impact, Organization, Evaluation, Benchmarks*, 2003.
[4] G. Behnke, D. Höller, and S. Biundo, "On the complexity of HTN plan verification and its implications for plan recognition," in *Proc. of ICAPS*. AAAI Press, 2015, pp. 25–33.
[5] P. Bercher, D. Höller, G. Behnke, and S. Biundo, "More than a name? on implications of preconditions and effects of compound HTN planning tasks," in *Proc. of ECAI*. IOS Press, 2016, pp. 225–233.
[6] G. Behnke, D. Höller, and S. Biundo, "This is a solution! (... but is it though?) - verifying solutions of hierarchical planning problems," in *Proc. of ICAPS*. AAAI Press, 2017, pp. 20–28.
[7] R. Barták, A. Maillard, and R. C. Cardoso, "Validation of hierarchical plans via parsing of attribute grammars," in *Proc. of ICAPS*. AAAI Press, 2018, pp. 11–19.
[8] M. Vilain, "Getting serious about parsing plans: A grammatical analysis of plan recognition," in *Proc. of AAAI*. AAAI Press, 1990, pp. 190–197.
[9] D. Höller, G. Behnke, P. Bercher, and S. Biundo, "Language classification of hierarchical planning problems," in *Proc. of ECAI*. IOS Press, 2014, pp. 447–452.
[10] D. Höller, G. Behnke, P. Bercher, and S. Biundo, "Assessing the expressivity of planning formalisms through the comparison to formal languages," in *Proc. of ICAPS*. AAAI Press, 2016, pp. 158–165.

[11] R. Barták and A. Maillard, “Attribute grammars with set attributes and global constraints as a unifying framework for planning domain models,” in *Proc. of the 19th Int. Symposium on Principles and Practice of Declarative Programming (PPDP’17)*. ACM, 2017, pp. 39–48.

[12] D. Avrahami-Zilberbrand and G. A. Kaminka, “Fast and complete symbolic plan recognition,” in *Proc of. IJCAI*. Morgan Kaufmann Publishers Inc., 2005, pp. 653–658.

[13] C. W. Geib, J. Maraist, and R. P. Goldman, “A new probabilistic plan recognition algorithm based on string rewriting,” in *Proc of. ICAPS*. AAAI Press, 2008, pp. 91–98.

[14] C. W. Geib and R. P. Goldman, “A probabilistic plan recognition algorithm based on plan tree grammars,” *Artificial Intelligence*, vol. 173, no. 11, pp. 1101–1132, 2009.

[15] F. Kabanza, J. Filion, A. R. Benaskeur, and H. Irandoust, “Controlling the hypothesis space in probabilistic plan recognition,” in *Proc. of IJCAI*. IJCAI/AAAI, 2013, pp. 2306–2312.

[16] R. Mirsky, Y. K. Gal, and S. M. Shieber, “CRADLE: an online plan recognition algorithm for exploratory domains,” *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 3, pp. 45:1–45:22, 2017.

[17] R. E. Fikes and N. J. Nilsson, “STRIPS: A new approach to the application of theorem proving to problem solving,” in *Proc. of IJCAI*, 1971, pp. 608–620.

[18] B. C. Gazen and C. A. Knoblock, “Combining the expressivity of UCPOP with the efficiency of graphplan,” in *Proc. of the 4th European Conference on Planning: Recent Advances in AI Planning (ECP’97)*. Springer, 1997, pp. 221–233.

[19] G. Behnke, D. Höller, A. Schmid, P. Bercher, and S. Biundo, “On succinct groundings of HTN planning problems,” in *Proc. of AAAI*. AAAI Press, 2020.

[20] D. Nau, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, “SHOP2: An HTN planning system,” *Journal of Artificial Intelligence Research*, vol. 20, pp. 379–404, 2003.

[21] R. P. Goldman and U. Kuter, “Hierarchical task network planning in common Lisp: the case of SHOP3,” in *Proceedings of the 12th European Lisp Symposium (ELS 2019)*. ACM, 2019, pp. 73–80.

[22] D. Höller, G. Behnke, P. Bercher, S. Biundo, H. Fiorino, D. Pellier, and R. Alford, “HDDL: An extension to PDDL for expressing hierarchical planning problems,” in *Proc. of AAAI*. AAAI Press, 2020.

[23] D. Höller, G. Behnke, P. Bercher, and S. Biundo, “Plan and goal recognition as HTN planning,” in *Proc. of ICTAI*. IEEE, 2018, pp. 466–473.