Validation of Hierarchical Plans via Parsing of Attribute Grammars

Roman Barták, Adrien Maillard
Charles University
Faculty of Mathematics and Physics
Prague, Czech Republic

Rafael C. Cardoso
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, Brazil

Abstract

An important problem of automated planning is validating if a plan complies with the domain model. Such validation is straightforward for classical sequential planning but until recently there was no plan validation approach for Hierarchical Task Networks (HTN). In this paper we propose a novel technique for validating HTN plans by parsing of attribute grammars with the timeline constraint.

Introduction

Automated planning deals with the problem of finding a sequence of actions to reach a certain goal (Ghallab, Nau, and Traverso 2004). Actions are specified via preconditions and postconditions (also called effects) describing propositions that must be true in the state before action application (preconditions) and that will become true after action application (postconditions). An action is a formal model of a state transition, and a plan – a sequence of actions – describes a valid evolution of the world from a given initial state.

To increase efficiency of planning, Hierarchical Task Networks (HTN) were proposed to describe sets of actions as recipes for solving specific tasks (Erol, Hendler, and Nau 1996). HTN models are based on decomposing compound tasks to subtasks until primitive tasks – actions – are obtained. The decomposition may include extra constraints describing precedence relations between subtasks or specific properties of world states (propositions that must hold before or between certain subtasks). The planning problem is specified as a goal task that needs to be decomposed to a sequence of actions applicable to an initial state, while satisfying all the task decomposition constraints and all the causal constraints between the actions. This sequence must be a valid plan in terms of action applicability.

An important problem in automated planning is validating plans with respect to a domain model. Such validation is easy for classical sequential planning, where it can be realized by simulating plan execution (Howey and Long 2003). However, until recently, there was no method to validate HTN plans, that is, to validate if a given plan can be obtained from the goal task by decomposition steps. There is a recent validation method based on modeling all possible decompositions as a SAT problem (Behnke, Höller, and Biundo 2017), but this method does not assume decomposition constraints (except decomposition preconditions that are compiled away to a dummy action). In this paper we suggest a more general approach that covers HTN models completely, including all decomposition and causal constraints.

It has already been noted that derivation trees of Context-Free (CF) grammars resemble the structure of Hierarchical Task Networks (HTN). This has been used in (Erol, Hendler, and Nau 1996) to show the expressiveness of planning formalisms. Then, there have been some attempts to represent HTNs as CF grammars or equivalent formalisms (Nederhof, Shieber, and Satta 2003), but as demonstrated in (Höller et al. 2014), the languages defined by HTN planning problems (with partial-order, preconditions and effects) lie somewhere between CF and context-sensitive (CS) languages. In (Geib 2016), the author presents an approach with a similar intention with the help of *Combinatory Categorial Grammars* (CCGs), which are part of a category lying between CF and CS grammars. The author proposes a single model for both plan recognition and planning based on CCGs. However, it appears that this modelling process is counter-intuitive as it requires a *lexicalization* (the hierarchical structure is contained in the terminal symbols) while the decomposition approach is more natural in planning. Also, it is not yet confirmed whether this formalism and its planning technique can produce the full range of HTN plans. Recently, a model of HTNs based on attribute grammars has been proposed (Barták and Maillard 2017). The underlying grammar describes proper task decompositions, while a so called *timeline* constraint over the task attributes describes valid orders of actions based on causal relations. It is the only model that handles all HTN constraints including interleaving of actions. String shuffling used in plan recognition (Maraist 2017) allows some form of task interleaving, but it is not clear how it maintains the causal constraints.

In this paper, we will use attribute grammars to validate HTN plans. We will describe how HTN domain model is represented as an attribute grammar, and we will present a parsing technique that does plan validation for this grammar. Note that due to interleaving of actions and presence of extra constraints, the parsing technique needs to be more general than classical parsing for CF grammars.

Background on Planning

In this paper we work with classical STRIPS planning that deals with sequences of actions transferring the world from an initial state to a state satisfying a goal condition. States are modelled as sets of propositions that are true in those states. Actions change validity of certain propositions.

Classical Planning

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 that are not valid in the state S.

Each action a is described by four sets of propositions $(B_a^+, B_a^-, A_a^+, A_a^-)$, where $B_a^+, B_a^-, A_a^+, A_a^- \subseteq P, B_a^+ \cap B_a^- = \emptyset, A_a^+ \cap A_a^- = \emptyset$. Sets B_a^+ and B_a^- describe positive and negative preconditions of action a, that is, propositions that must be true and false right before the action a. Action a is applicable to state S iff $B_a^+ \subseteq S \wedge B_a^- \cap S = \emptyset$. 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

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

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

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 disjoint sets of goal propositions G^+ and G^- describing the propositions required to be true and false in the goal state. 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 $G^+ \subseteq S \wedge G^- \cap S = \emptyset$. This sequence of actions is called a *plan*.

Hierarchical Task Networks as Attribute Grammars

To simplify the planning process, several extensions of the basic STRIPS model were proposed to include some control knowledge. Hierarchical Task Networks (Erol, Hendler, and Nau 1996) were proposed as a planning domain modeling framework that includes control knowledge in the form of recipes on how to solve specific tasks. The recipe is represented as a task network, which is a set of subtasks to solve a given task, together with the set of constraints between the subtasks. The constraints can be of the following types:

- $t_1 \prec t_2$: 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,
- *before*(U, l): a precondition constraint meaning that in every plan the literal l holds in the state right before the first action obtained from tasks U,
- *after*(a, l): a postcondition constraint meaning that the literal l will hold in the state right after the action a,
- *between*(U, V, l): a prevailing condition meaning that in every plan the literal l holds in all the states between the last action obtained from tasks U and the first action obtained from tasks V.

Notice that, as usually, the *after* constraints are used only in relation to actions to model their effects, while the other constraints can be used in relation to compound tasks and their decomposition methods. In HTN, a compound task is solved by decomposing it to a task network via a (decomposition) *method*. The method can naturally be described as a rewriting rule of an attribute grammar. Attribute grammars (Knuth 1968) use the same type of rewriting rules as context-free grammars, but the grammar symbols may by annotated by attributes connected by constraints. This makes attribute grammars stronger than CF grammars in the sense of recognizing a larger class of languages. It has been shown that attribute grammars with the global timeline constraint and set attributes can fully model HTNs (Barták and Maillard 2017). In this paper we use an abstract model, where the method constraints are used in the grammar directly.

Let $T(\overrightarrow{X})$ be a compound task with parameters $\overrightarrow{X}$ and $(\{T_1(\overrightarrow{X_1}), ..., T_k(\overrightarrow{X_k})\}, C)$ be a task network, where C are its constraints. We can encode the decomposition method as an attribute grammar rule:

$$T(\overrightarrow{X}) \to T_1(\overrightarrow{X_1}), ..., T_k(\overrightarrow{X_k}) \;\; [C]$$

The planning problem in HTN is specified by an initial state (the set of propositions that hold at the beginning) and by an initial task representing the goal. The compound tasks need 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.

If we do planning by application of rewriting rules, we get a linear sequence of actions (a terminal word in terms of formal grammars), but this sequence does not necessarily form a valid plan as the actions from different tasks may interleave to satisfy the ordering and causal constraints (see Figure 1). So the actions obtained by applying the rewriting rules need to be re-ordered to get a valid plan. The attribute grammar models the valid action orderings via the global timeline constraint (Barták and Maillard 2017).

To give a particular example of the decomposition rule, let us assume a task to transfer a container c from one location $l1$ to another location $l2$ by a robot r. To solve this task, we need to load the container first, then to move it to its destination location, and to unload it there. The following rule describes this decomposition method[1]:

$$\begin{aligned} \text{Transfer1}(c, l1, l2, r) \to\, & \text{Load-rob}(c, r, l1). \\ & \text{Move-rob}(r, l1, l2). \\ & \text{Unload-rob}(c, r, l2) \\ & [C] \end{aligned} \tag{2}$$

[1]There are other ways to model the task. For example, the *before* constraints can be omitted as they will be part of the actions.

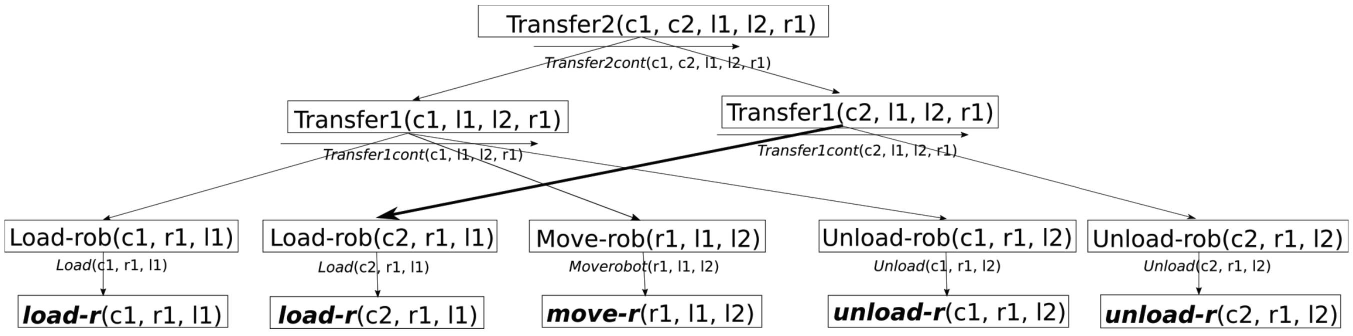

Figure 1: A task decomposition tree showing interleaving of actions obtained from decompositions of different tasks - denoted by the bold arc.

where

$$\begin{aligned} C = \{\, & \text{Load-rob} \prec \text{Move-rob},\ \text{Move-rob} \prec \text{Unload-rob}, \\ & \textit{before}(\{\text{Load-rob}\}, at(r, l1)), \\ & \textit{before}(\{\text{Load-rob}\}, at(c, l1)), \\ & \textit{between}(\{\text{Load-rob}\}, \{\text{Move-rob}\}, at(r, l1)), \\ & \textit{between}(\{\text{Move-rob}\}, \{\text{Unload-rob}\}, at(r, l2)), \\ & \textit{between}(\{\text{Load-rob}\}, \{\text{Unload-rob}\}, in(c, r))\} \end{aligned} \tag{3}$$

The decomposition constraints specify the following restrictions:

- the robot and the container must be at the same location $l1$ before loading,
- the robot does not change its location between loading and the start of moving,
- the container stays in the robot between loading and unloading,
- the robot stays at the destination location $l2$ between the end of moving and the start of unloading.

An alternative decomposition method omits the Move-rob task as it assumes that this task is introduced by decomposition of another compound task. See the task for $c2$ in Figure 1. Still, we need to ensure that the robot is at the right location before unloading, which is done by the constraint *before*$(\{\text{Unload-rob}\}, at(r, l2))$. The alternative decomposition rule looks as follows:

$$\begin{aligned} \text{Transfer1}(c, l1, l2, r) \rightarrow\ & \text{Load-rob}(c, r, l1). \\ & \text{Unload-rob}(c, r, l2) \\ & [C] \end{aligned} \tag{4}$$

where

$$\begin{aligned} C = \{\, & \text{Load-rob} \prec \text{Unload-rob}, \\ & \textit{before}(\{\text{Load-rob}\}, at(r, l1)), \\ & \textit{before}(\{\text{Load-rob}\}, at(c, l1)), \\ & \textit{before}(\{\text{Unload-rob}\}, at(r, l2)), \\ & \textit{between}(\{\text{Load-rob}\}, \{\text{Unload-rob}\}, in(c, r))\} \end{aligned} \tag{5}$$

The top task for transferring two containers using the same robot and between the same locations can be described using the following decomposition method:

$$\begin{aligned} \text{Transfer2}(c1, c2, l1, l2, r) \rightarrow\ & \text{Transfer1}(c1, l1, l2, r). \\ & \text{Transfer1}(c2, l1, l2, r) \\ & [] \end{aligned} \tag{6}$$

Notice that having the *before* and *after* constraints allows us to describe action preconditions and postconditions as decomposition constraints rather than having them specified separately. This is done by having a primitive task that decomposes only to a single corresponding action. There is one-to-one relation between the primitive tasks and actions. For example, the primitive tasks Load-rob corresponds to action load-r. The corresponding decomposition method is:

$$\text{Load-rob}(c, r, l) \rightarrow \text{load-r}(c, r, l). \qquad [C] \tag{7}$$

where

$$\begin{aligned} C = \{ & \textit{before}(\{\text{load-r}\}, at(r, l)), \\ & \textit{before}(\{\text{load-r}\}, at(c, l)), \\ & \textit{after}(\text{load-r}, in(c, r))\} \\ & \textit{after}(\text{load-r}, \neg at(c, l)\} \end{aligned}$$

HTN Validation Algorithm

The plan validation problem is a problem reverse to the planning problem. We have a plan as the input and the problem is to validate if that plan can be obtained by decomposition from the goal task. In terms of grammars, it means using the grammar rules analytically to do validation of hierarchical plans via parsing.

Recall that the order of actions in the plan does not necessarily correspond to the order of actions obtained by application of grammar rules. Hence, during parsing, we ignore the order of tasks on the right side of grammar rules and we model the action (task) order explicitly by using indexes assigned to tasks. Each task will be annotated by two indexes describing the order numbers of the first and the last actions obtained from task decomposition. For example, the task $\text{Load-rob}_{1,1}(c1, r1, l1)$ from Figure 1, that gives the action $\text{load-r}(c1, r1, l1)$, is annotated by indexes 1,1.

Let us now demonstrate a single parsing step. Assume that we already have the primitive tasks $\text{Load-rob}_{1,1}(c1, r1, l1)$, $\text{Move-rob}_{3,3}(r1, l1, l2)$, and $\text{Unload-rob}_{4,4}(c1, r1, l2)$ and we continue parsing using the grammar rule (2). The tasks on the right side of the rule already exist and we can verify the ordering constraints $1 \prec 3$ and $3 \prec 4$ by comparing the respective indexes. The result of the parsing step will be a new parsed task $\text{Transfer1}_{1,4}(c1, l1, l2, r1)$, where the indexes are taken as minimal and maximal indexes of its subtasks.

We still need to verify the other constraints in the rule. This will be done by maintaining a timeline for each task. The *timeline* is a sequence of slots describing validity of literals in time steps corresponding to the task. For every time step, the slot will describe the literals that hold in the state before the action at that time (a Pre part), and literals that must hold in the state right after the action (a Post part). For example, the task $\text{Load-rob}_{1,1}(c1, r1, l1)$ will use a single slot $(\{at(r1, l1), at(c1, l1)\}, \{in(c1, r1), \neg at(c1, l1)\})_1$, where the index represents time and the literals are basically preconditions and postconditions of action $\text{load-r}(c1, r1, l1)$ that were encoded as *before* and *after* constraints (see the rule (7)).

During the parsing step, we first *merge* the timelines for the subtasks with possible insertion of empty slots for times not covered by the subtasks (slot 2 in our example). Empty slot does not contain any action, but its Pre part may contain literals obtained by propagation (see below). Two slots with the same index can only be merged if (at least) one of them is empty. This way we ensure that each action is generated exactly once. For example, when merging timelines for tasks $\text{Transfer1}_{1,4}(c1, l1, l2, r1)$ and $\text{Transfer1}_{2,5}(c2, l1, l2, r1)$ we are merging non-empty slots 1,3,4 for the first task with non-empty slots 2, 5 of the second task. If the slots cannot be merged as they both already contain an action, then processing of the grammar rule stops and the algorithm continues by trying the next rule.

After merging the timelines for subtasks we add literals based on the rule *constraints*. The literals are added to the Pre parts of respective slots for the *before* and *between* constraints. Note that because we know the indexes of tasks, we know exactly to which slots the literals should be added.

After that, we *propagate* the literals between the slots. This propagation goes from left to right, where the literals from the postcondition part are added to the precondition part of the next slot and, if the slot is not empty (contains some action), the literals in preconditions, that are not deleted by the action, are added to the precondition part of the next slot. This basically follows the state transition formula as specified in (1). The right-to-left propagation adds literals in preconditions to preconditions of the previous slot provided that the slot is not empty and the literal is not added by the action in it. The goal of propagation is to keep information about states up-to-date. Notice that propagation changes only the Pre parts of the slots describing the states.

Finally, we verify that the slots are consistent, which consists of checking that no slot contains a literal and its negation in any of its parts. Table 1 demonstrates this process — it shows how literals are added to the slots in each step (slot merging, constraint addition, and propagation).

The validation algorithm first transfers each action to a primitive task with the index corresponding to the order of the action in the plan, with the timeline containing a single slot with that action, and with the Pre and Post parts that are filled by preconditions and postconditions of the action as specified in the rewriting rule for the primitive task; see the rule (7). The literals of the initial state are added to the Pre part of the first slot (for simplicity, we ignored them in the previous example of a parsing step). Then the algorithm takes any grammar rule such that the tasks from its right side are already known and it does the above-described parsing step. If it is successful then a new parsed task is introduced. This process is repeated while some new task is introduced or until a goal task is introduced whose indexes span the whole plan. If the goal task is found then the plan is sound, otherwise, the plan is not sound. Note that the algorithm always finishes as there is only a finite number of compound tasks that can be introduced during parsing. We will now describe the validation algorithm formally.

Data structures

First we will describe the data structures that are used later in the algorithm. Basically, we will introduce slots, timelines, and the parsed tasks:

We define the type `slot` as a tuple $(\text{Pre}^+, \text{Pre}^-, a, \text{Post}^+, \text{Post}^-)$ where

- Pre^+ is a set of atoms (positive propositions in the state)
- Pre^- is a set of atoms (negative propositions in the state)
- $a \in A \cup \{\textit{empty}\}$ is an action name (or an empty slot)
- Post^+ is a set of atoms (positive postconditions of a)
- Post^- is a set of atoms (negative postconditions of a)

To simplify verification of slot/timeline soundness we use separate sets for positive and negative propositions. Note also that the sets $\text{Pre}^+, \text{Pre}^-$ are not only related to action a but they will describe the state right before the action. More precisely, these sets describe the propositions that must hold in the state, but until all slots are non-empty, the state may be described only partially (see Table 1).

Next, we define the type `subplan` that represents a parsed task T as a tuple $(T, b, e, timeline)$ with

- T being a task name (with attributes of the task),
- b and e ($b \leq e$) being two integers equal to the indexes in the original plan of the first and last actions in the subplan generated from T; this pair shows how much the subplan generated from T *spans over* the verified plan,
- $timeline$ being an ordered sequence of $(e - b + 1)$ elements of the `slot` type; we have $timeline = \{s_b, ..., s_e\} \subseteq \textbf{slots}$.

The algorithm formally

The validation algorithm is shown in Algorithm 1. At the beginning, primitive tasks TP corresponding to actions in the plan are put individually in the set **subplans** (lines 2-7). They are all subplans of size 1. The initial state is added

Table 1: The process of building a timeline during parsing the compound task $\text{Transfer1}_{1,4}(c1, l1, l2, r1)$.

	1: load-r$(c1, r1, l1)$		2: *empty*		3: move-r$(r1, l1, l2)$		4: unload-r$(c1, r1, l2)$	
	Pre_1	$Post_1$	Pre_2	$Post_2$	Pre_3	$Post_3$	Pre_4	$Post_4$
merge	$at(r1, l1)$ $at(c1, l1)$	$in(c1, r1)$ $\neg at(c1, l1)$			$at(r1, l1)$	$\neg at(r1, l1)$ $at(r1, l2)$	$in(c1, r1)$ $at(r1, l2)$	$\neg in(c1, r1)$ $at(c1, l2)$
constrain			$at(r1, l1)$ $in(c1, r1)$		$in(c1, r1)$			
propagate			$\neg at(c1, l1)$				$\neg at(r1, l1)$	

```
Data: a plan $\mathbf{P} = (a_1, ..., a_n)$, initial state InitState, a goal
      task Goal, an attribute grammar
      $G = (\Sigma, N, \mathcal{P}, S, A, C)$
Result: a Boolean equal to true if the plan can be derived
      from the hierarchical structure, false otherwise
1  Function VERIFYPLAN
       /* Initialization of the set of subplans */
2      subplans ←
         $\{(TP_i, i, i, \{(\text{Pre}_i^+, \text{Pre}_i^-, a_i, \text{Post}_i^+, \text{Post}_i^-)_i\})|$
3        $a_i \in \mathbf{P}, (TP_i \to a_i\ [pre, post]) \in \mathcal{P},$
4        $\text{Pre}_i^+ = \{p | before(\{a_i\}, p) \in pre\},$
5        $\text{Pre}_i^- = \{p | before(\{a_i\}, \neg p) \in pre\},$
6        $\text{Post}_i^+ = \{p | after(a_i, p) \in post\},$
7        $\text{Post}_i^- = \{p | after(a_i, \neg p) \in post\}\}$ ;
8      $\text{Pre}_1^+ \leftarrow \text{Pre}_1^+ \cup InitState^+$;
9      $\text{Pre}_1^- \leftarrow \text{Pre}_1^- \cup InitState^-$;
10     while ¬PLANISVALID(subplans, P, Goal) do
11         for each rule $R \in \mathcal{P}$ of the form
             $T_0 \to T_1, ..., T_k\ [\prec, pre, btw]$ such that
             $subtasks = \{(T_i, b_i, e_i, tl_i) | i \in 1..k\} \subseteq$
             subplans do
12             verify $\prec$ from rule $R$ else break;
13             $timeline \leftarrow$ MERGEPLANS$(subtasks)$;
14             APPLYPRE$(timeline, pre)$;
15             APPLYBETWEEN$(timeline, btw)$;
16             PROPAGATE$(timeline)$;
17             if $\exists(\text{Pre}^+, \text{Pre}^-, a, \text{Post}^+, \text{Post}^-) \in$
                 $timeline, \text{Pre}^+ \cap \text{Pre}^- \neq$
                 $\emptyset \vee \text{Post}^+ \cap \text{Post}^- \neq \emptyset$ then
18                 break
19             end
20             $b = \min_{(T_i, b_i, e_i, tl_i) \in subtasks} b_i$;
21             $e = \max_{(T_i, b_i, e_i, tl_i) \in subtasks} e_i$;
22             subplans ←
                 subplans $\cup \{(T_0, b, e, timeline)\}$;
23         end
24         if size of subplans has not increased since the
             last iteration then
25             return false
26         end
27     end
28     return true
29 end
```

Algorithm 1: Verification procedure

```
Data: the set of subplans: subplans, the plan to be
      validated P, the goal task Goal
Result: true or false
1  Function PLANISVALID
2      return $(\exists(Goal, 1, |\mathbf{P}|, timeline) \in$
         subplans, $s.t. \bigcup_{(\_,\_,a_i,\_,\_) \in timeline} \{a_i\} = \mathbf{P})$
3  end
```

Algorithm 2: The end condition of the valid plan

to the Pre parts of the slot of the first primitive task. Then, at each iteration the algorithm fires rules in the grammar, where all subtasks are elements of **subplans**. When such a rule is found, the precedence constraints are checked (line 12). Then the timelines of subtasks are merged (line 13) and before and between constraints from the grammar rule are applied to this merged timeline (lines 14 and 15). Preconditions and postconditions are then propagated from left to right and from right to left (line 16). Finally, the resulting timeline is verified (17). If no inconsistency is detected, then the new parsed task is added to the set **subplans** so it can be used further for building a higher-level task. Inconsistency means that some atom is both in the positive and in the negative parts of the state.

The positive exit condition (cf. Algorithm 2) is met when there is a *Goal* task in **subplans** that contains all the elements of the verified plan $\mathbf{P}$.

If, it is not possible to find a rule that applies to the current elements of **subplans** and produces a *new* subplan, then it means that the plan $\mathbf{P}$ is not valid with regards to the grammar. In other words, the set **subplans** has not grown during the execution of the for-loop (lines 11 to 22). At this point, the algorithm returns false (line 24).

We also include all the sub-procedures for merging the timelines and for applying the constraints. To simplify notations in the procedures for constraint application (Algorithms 5-6), we use the following notation – if l is a positive literal p then $l^+ = \{p\}$ and $l^- = \{\}$; if l is a negative literal $\neg p$ then $l^+ = \{\}$ and $l^- = \{p\}$.

Soundness and Complexity

We shall now show that the algorithm correctly recognizes plans that can be derived from a given *Goal* task and an initial state.

First, one should realize that the algorithm always finishes. All sub-procedures clearly finish as they consist of *for* loops and *if-then-else* conditions only. During each itera-

```
Data: a set of subplans : $subplans$
Result: a set of slots $newtimeline$, the aggregation of the
        slots of every subplan
Function MERGEPLANS($subplans$)
    $lb = \min_{(T_i,b_i,e_i,timeline_i)\in subplans} b_i$;
    $ub = \max_{(T_i,b_i,e_i,timeline_i)\in subplans} e_i$;
    $newtimeline \leftarrow \{(\emptyset, \emptyset, empty, \emptyset, \emptyset)_i | i \in lb..ub\}$;
    for $(T, b, e, timeline) \in subplans$ do
        for $s_k \in timeline, s_k^{'} \in newtimeline$ do
            $s_k^{'} \leftarrow$ MERGESLOTS$(s_k, s_k^{'})$
        end
    end
    return $newtimeline$
end
```

Algorithm 3: Merge timelines

```
Data: two slots
      $s_1 = (\text{Pre}_1^+, \text{Pre}_1^-, a_1, \text{Post}_1^+, \text{Post}_1^-), s_2 =$
      $(\text{Pre}_2^+, \text{Pre}_2^-, a_2, \text{Post}_2^+, \text{Post}_2^-)$
Result: merged slots
Function MERGESLOTS($s_1, s_2$)
    if $a_1 = empty$ or $a_2 = empty$ then
        $\text{Pre}^+ = \text{Pre}_1^+ \cup \text{Pre}_2^+$;
        $\text{Pre}^- = \text{Pre}_1^- \cup \text{Pre}_2^-$;
        $\text{Post}^+ = \text{Post}_1^+ \cup \text{Post}_2^-$;
        $\text{Post}^- = \text{Post}_1^- \cup \text{Post}_2^-$;
        $a = a_1 (if\ a_2 = empty)$ or $a_2 (if\ a_1 = empty)$;
        return $(\text{Pre}^+, \text{Pre}^-, a, \text{Post}^+, \text{Post}^-)$
    end
    break
end
```

Algorithm 4: Merge slots

```
Data: a set of slot : $slots$, a set of $before$ constraints
Result: an updated set of slots
Function APPLYPRE($slots, pre$)
    for $before(U, l) \in pre$ do
        $id = \min\{b_i | T_i \in U\}$;
        $\text{Pre}_{id}^+ \leftarrow \text{Pre}_{id}^+ \cup l^+$;
        $\text{Pre}_{id}^- \leftarrow \text{Pre}_{id}^- \cup l^-$
    end
end
```

Algorithm 5: Apply before constraints

```
Data: a set of slot : $slots$, a set of $between$ constraints
Result: an updated set of slots
Function APPLYBETWEEN($slots, between$)
    for $between(U, V, l) \in between$ do
        $s = \max\{e_i | T_i \in U\} + 1$;
        $e = \min\{b_i | T_i \in V\}$;
        for $id = s$ to $e$ do
            $\text{Pre}_{id}^+ \leftarrow \text{Pre}_{id}^+ \cup l^+$;
            $\text{Pre}_{id}^- \leftarrow \text{Pre}_{id}^- \cup l^-$
        end
    end
end
```

Algorithm 6: Apply between constraints

```
Data: a set of slots $slots$
Result: an updated set of slots
Function PROPAGATE($slots$)
    $lb = \min_{(\text{Pre}_j^+, \text{Pre}_j^-, a_j, \text{Post}_j^+, \text{Post}_j^-)\in slots} j$;
    $ub = \max_{(\text{Pre}_j^+, \text{Pre}_j^-, a_j, \text{Post}_j^+, \text{Post}_j^-)\in slots} j - 1$;
    /* Propagation to the right          */
    for $i = lb$ to $ub$ do
        $\text{Pre}_{i+1}^+ \leftarrow \text{Pre}_{i+1}^+ \cup \text{Post}_i^+$;
        $\text{Pre}_{i+1}^- \leftarrow \text{Pre}_{i+1}^- \cup \text{Post}_i^-$;
        if $a_i \neq empty$ then
            $\text{Pre}_{i+1}^+ \leftarrow \text{Pre}_{i+1}^+ \cup (\text{Pre}_i^+ \setminus \text{Post}_i^-)$;
            $\text{Pre}_{i+1}^- \leftarrow \text{Pre}_{i+1}^- \cup (\text{Pre}_i^- \setminus \text{Post}_i^+)$
        end
    end
    /* Propagation to the left           */
    for $i = ub$ downto $lb$ do
        if $a_i \neq empty$ then
            $\text{Pre}_i^+ \leftarrow \text{Pre}_i^+ \cup (\text{Pre}_{i+1}^+ \setminus \text{Post}_i^+)$;
            $\text{Pre}_i^- \leftarrow \text{Pre}_i^- \cup (\text{Pre}_{i+1}^- \setminus \text{Post}_i^-)$
        end
    end
end
```

Algorithm 7: Propagate

tion of the main *while* loop, some new task may be added to the set of **subplans**. The input plan is finite and we have only a finite number of constants so the number of tasks that can be derived is obviously finite. Hence the *while* loop must finish sometime, either when no new task is added (line 24) or when the $Goal$ task is derived (line 10). In the worst case, the algorithm explores all subsets of actions in the plan so its worst-case time complexity is exponential in the size of the plan. This is not surprising as the HTN plan validation problem is NP-complete (Behnke, Höller, and Biundo 2015). Nevertheless, as the experiments showed the practical performance is very good (see the next section).

Assume that the algorithm finished successfully (with the answer **true**). It means that it found the $Goal$ task that spans over the full plan (test in Algorithm 2). By reconstructing how this task was added to the set **subplans**, we get the derivation tree (such as the one in Figure 1). We indeed get a tree as during merging of timelines, two slots can only be merged if at least one of them is empty. Hence each task in the tree has exactly one parent. If the same task appears two (or more) times in the tree then its slots would eventually merge with themselves, which is not possible (see Algorithm 4). All the constraints used in this derivation (decomposition) are satisfied as the algorithm verified the precedence constraints and added the literals from the before, after, and between constraints to the timeline and this timeline is consistent.

Notice that the $Post$ parts of the slots in the timeline contain exactly the propositions from the after constraints modeling the effects of actions. The Pre parts (in particular the Pre^+ sets) model the states between the actions and we shall show that the sequence of states is correct with respect to the plan. First, each state is sound as it does not contain an atom and its negation ($\text{Pre}^+ \cap \text{Pre}^- = \emptyset$). Next, two subsequent states Pre^+_i and Pre^+_{i+1} model a correct state transition thanks to the propagation:

$$\text{Pre}^+_{i+1} = (\text{Pre}^+_i \setminus \text{Post}^-_i) \cup \text{Post}^+_i$$
$$\text{Pre}^-_{i+1} = (\text{Pre}^-_i \setminus \text{Post}^+_i) \cup \text{Post}^-_i$$

This realizes the state transition formula (1). We will show it for the positive part of the state (the proof is identical for the negative part). Assume slots i and $i+1$ with some action filled in the slot i (the action must appear there eventually as the final timeline has all slots non-empty). Thanks to left-to-right propagation, it must hold $\text{Post}^+_i \subseteq \text{Pre}^+_{i+1}$ (line 5 of Algorithm 7) and $\text{Pre}^+_i \setminus \text{Post}^-_i \subseteq \text{Pre}^+_{i+1}$ (line 8 of Algorithm 7). Thanks to right-to-left propagation, it must hold $\text{Pre}^+_{i+1} \setminus \text{Post}^+_i \subseteq \text{Pre}^+_i$ (line 14 of Algorithm 7). It means that if a proposition $p \in \text{Pre}^+_{i+1}$ is not added by the action ($p \notin \text{Post}^+_i$) then p must already be part of the previous state ($p \in \text{Pre}^+_i$). Assume that p is deleted by the action. Then $p \in \text{Post}^-_i$ and hence also $p \in \text{Pre}^-_{i+1}$ (line 6 of the Algorithm 7). Thanks to soundness of the state, it implies $p \notin \text{Pre}^+_{i+1}$, which is a contradiction. Hence $p \notin \text{Post}^-_i$. Together, we get:

$$\text{Pre}^+_{i+1} = (\text{Pre}^+_i \setminus \text{Post}^-_i) \cup \text{Post}^+_i$$

Notice that the algorithm works even if no initial state is provided. Then the final sets Pre^+_1 and Pre^-_1 specify the propositions that must and must not be valid at the beginning to have a valid plan. If the initial state is provided then it is propagated through the slots.

In summary, the set of actions in the plan is generated by the grammar and forms a valid plan. If the algorithm finishes with the answer **false** then no derivation exists as no other task can be parsed. Being the plan correct, the derivation tree would be reconstructed by the algorithm as the algorithm finds all the tasks that decompose to any subset of the plan.

We showed that the algorithm always finishes. If it returns **true** then the plan can be derived from the *Goal* task. If it returns **false** then the plan cannot be derived from the *Goal* task. Hence the algorithm validates the plans with respect to the domain model.

Results of the Experiments

The algorithm is implemented[2] using the Ruby programming language. We use the Hype module of the HyperTensioN[3] HTN planner. We adapted the module to parse from SHOP2 syntax and to compile attribute grammars based on domain and problems described in SHOP2 syntax. Then, the attribute grammar validator is called to validate a plan using the attribute grammar.

We performed experiments across two different domains, comparing performance of the implementation of our algorithm against the PANDA verifier (Behnke, Höller, and Biundo 2017), the only other known implementation of HTN plan validation. The PANDA verifier validates a plan by translating it into a SAT formula. This translation requires a bound, the maximum height of the decomposition that any candidate for a solution can have.

Our implementation can parse directly from SHOP2 planner's (Nau et al. 2003) input files. Only basic HTN syntax from SHOP2 is supported for now, but support for new syntax elements are gradually being added. The PANDA verifier uses its own input, which is a PDDL-like representation of HTN, and it does not offer any support for SHOP2 syntax.

The first domain that we used in our experiments is the Transport domain, initially used in the International Planning Competition (IPC) of 2008. In this domain, each truck vehicle can transport packages between different locations based on road connections. The second domain is the Satellite domain, first used in the 2002 IPC, where multiple satellites can be equipped with different and possibly overlapping capabilities that can be used in observation tasks.

We ran both implementations for 20 problems in each domain, resulting in a total of 40 problems and 80 executions. Each domain has 10 problems with different configurations, increasing the size of the plan to be validated and the number of literals used to represent the initial state. All plans provided with these problems are valid. We then made additional 10 variants of these problems with non-valid plans of the same size as their valid counterparts. In the Transport domain, the first problem has a plan of size 2, increasing up to a plan of size 10 in the last problem. In the Satellite domain, the first problem starts with a plan of size 2, and increases up to a plan of size 15 in the last problem.

We collected real time spent on each execution. These times include any parsing done by both approaches, and was calculated from the beginning to the end of each validation process. Both approaches seem to benefit from multi-core processors, but in our experiments PANDA had a substantial difference between real and CPU time, which indicates that it might benefit the most from using multi-core processors, although its performance on single threaded processors will be worse than the results shown here. To run these experiments we used a computer with an Intel Xeon Processor E5645 (12M Cache, 2.40 GHz, 6 cores, 12 threads), 32 GB of memory, and the Ubuntu 16.04 operating system. Our implementation requires Ruby (we used version 2.3.1), while the PANDA verifier requires Java (we used OpenJDK 1.8) and the MiniSat solver (we used version 2.2.1).

Our Transport domain model in the SHOP2 syntax contains three primitive tasks and three non-primitive tasks. The model used in the PANDA verifier has four primitive tasks and six non-primitive tasks. The additional primitive task is a *noop* action, which in our model is encoded directly as a non-primitive task. The extra non-primitive tasks from PANDA's model are dummy methods that represent primitive tasks. The Satellite domain model in the SHOP2 syntax contains five primitive tasks and eight non-primitive tasks. The PANDA verifier, in its syntax, uses five primitive tasks and 15 non-primitive tasks.

Figure 2 contains the results for time spent validating each problem from the Transport domain for both valid and non-valid versions of the problems. Our approach ap-

[2]The source code for the attribute grammar validator is available at https://github.com/rafaelcaue/attribute-grammar-htn

[3]https://github.com/Maumagnaguagno/HyperTensioN

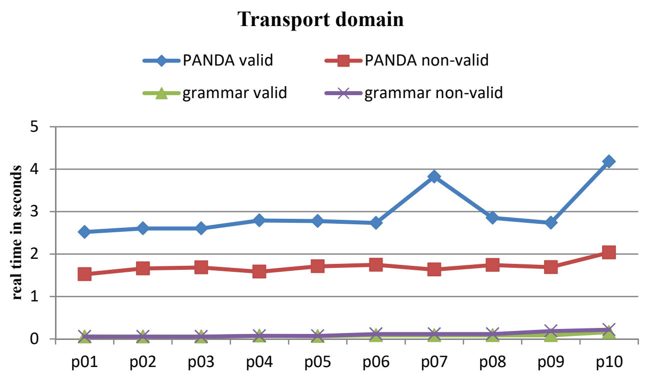

Figure 2: Transport domain results.

pears to scale linearly independently of whether the plan is valid or not, but as expected takes a bit more time for non-valid plans. From our experiments in the Transport domain, PANDA verifier appears to scale poorly when increasing the number of literals, which has the largest increase from p09 to p10.

In Figure 3 we show the results for the Satellite domain. PANDA verifier maintains a similar performance to the one shown in the previous domain, although in some of the last problems it seems to scale better when validating non-valid plans. Our implementation of the attribute grammar validation algorithm is also able to maintain the same performance, except for the last problem which has a steep increase in time, but it is still faster than PANDA. This increase was mostly due to the larger plan size in combination with the increase in total number of non-primitive tasks when compared to the results obtained in the Transport domain.

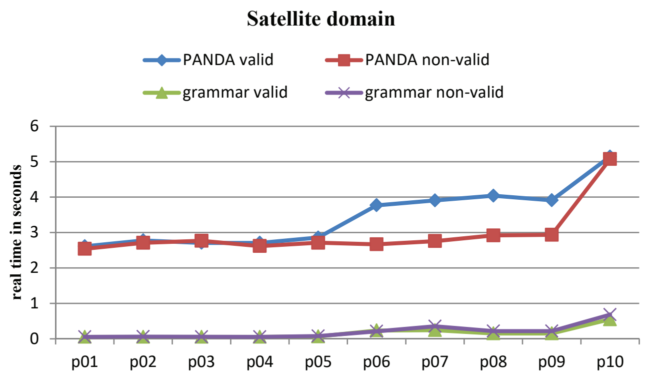

Figure 3: Satellite domain results.

Summary

In this paper we proposed an algorithm for validating HTN plans via parsing of an attribute grammar describing the HTN domain model. The algorithm mimics classical parsing of context-free grammars customized to attribute grammars with the timeline constraint.

The algorithm starts with the plan and applies the decomposition rules in a reverse order to group actions into tasks. The decomposition constraints are verified by keeping information about propositions that must be true at states before and after actions. The algorithm stops when it finds a task that covers the complete plan. Then the plan is valid. The other way of stopping the algorithm is when no other compound task can be constructed. In such a case, the plan does not correspond to any task. Note, that the plan might still be a correct sequence of actions, but it cannot be obtained by decomposition of any task described in the domain model.

The major innovation of the proposed technique is that it is the first approach that covers HTN models fully, including interleaving of actions and various decomposition constraints. In particular, the proposed algorithm is more general than the existing SAT-based approach in covering precedence, before, between, and after constraints. The SAT-based approach only covers specific before constraints (the constraint is applied to the set of all tasks on the right side of the rule) that must be encoded as dummy actions. These dummy actions must be part of the plan to be validated, so for the original plan to be validated one must find proper places where to insert these dummy actions, which is not discussed in (Behnke, Höller, and Biundo 2017).

Furthermore, our experiments indicate that converting HTN models to attribute grammars may provide better time-performance results in comparison to converting to SAT. More experiments with other domains are needed to ascertain in which types of domain each approach performs better. The size of the plan to be validated has a high impact on the scaling of these techniques, and in future work, as both implementations get better optimized overtime, we expect to make new experiments to see how far they can scale.

Our current implementation of the algorithm uses a straightforward approach to find rules used for parsing. The more efficient implementation of the algorithm may exploit principles of the Rete algorithm (Forgy 1982) used for production rule systems.

As other planning models, such as procedural domain control knowledge (Baier, Fritz, and McIlraith 2007), can be translated to attribute grammars (Barták and Maillard 2017) the proposed method can verify plans for these models too.

Formal grammars are used frequently in plan and goal recognition, for example in systems such as ELEXIR (Geib 2009), but none of the existing grammar-based systems covers HTN fully with all types of method constraints. These systems use partial plans (for example, plan prefix) as their input. The proposed HTN plan validation algorithm expects a complete plan as its input and an open question is how to modify the method to work with partial plans.

Acknowledgments Research is supported by the Czech Science Foundation under the projects P103-15-19877S and P103-18-07252S.

References

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting Procedural Domain Control Knowledge in State-of-the-Art Planners. In Boddy, M. S.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007*, 26–33. AAAI.

Barták, R., and Maillard, A. 2017. Attribute grammars with set attributes and global constraints as a unifying framework for planning domain models. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, 39–48. New York, NY, USA: ACM.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the complexity of htn plan verification and its implications for plan recognition. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2017*, 25–33.

Behnke, G.; Höller, D.; and Biundo, S. 2017. This is a solution! (... but is it though?) verifying solutions of hierarchical planning problems. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017*, 20–28.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity Results for HTN Planning. *Ann. Math. Artif. Intell.* 18(1):69–93.

Forgy, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19(1):17 – 37.

Geib, C. W. 2009. Delaying Commitment in Plan Recognition Using Combinatory Categorial Grammars. In Boutilier, C., ed., *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 1702–1707.

Geib, C. 2016. Lexicalized reasoning about actions. *Advances in Cognitive Systems* 4:187–206.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated planning - theory and practice*. Elsevier.

Howey, R., and Long, D. 2003. VAL's Progress: The Automatic Validation Tool for PDDL2.1 used in the International Planning Competition. In *Proceedings of ICAPS'03 Workshop on the Competition: Impact, Organization, Evaluation, Benchmarks*.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., eds., *ECAI 2014 - 21st European Conference on Artificial Intelligence - Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, 447–452. IOS Press.

Knuth, D. E. 1968. Semantics of Context-Free Languages. *Mathematical Systems Theory* 2(2):127–145.

Maraist, J. 2017. String shuffling over a gap between parsing and plan recognition. In *The AAAI-17 Workshop on Plan, Activity, and Intent Recognition WS-17-13*, 835–842.

Nau, D.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. Shop2: An htn planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Nederhof, M.-J.; Shieber, S.; and Satta, G. 2003. Partially ordered multiset context-free grammars and ID/LP parsing. *Proceedings of the Eighth International Workshop on Parsing Technologies* 171–182.