On Semantics of Hierarchical Planning Domain Models with Decomposition Constraints and Empty Methods

Simona Ondrčková, Roman Barták
Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
{ondrckova,bartak}@ktiml.mff.cuni.cz

***Abstract*—There are multiple formalisms describing hierarchical planning domain models, however many of them do not show semantics of some features such as empty decomposition methods and extensive constraints. In this short paper we describe the semantics of a hierarchical domain model with these extensive constraints and show how empty decomposition methods would work within it. We also compare this model with other models and present some transformations of model properties.**

***Index Terms*—planning, hierarchical models, semantics, transformations**

I. Introduction

Planning focuses on selecting and organizing actions into a sequence (plan) to achieve a specific goal from a given initial state. *Hierarchical planning* is a form of planning that is similar to how people solve complex problems. It is based on idea of decomposing complex tasks to simpler sub-tasks until obtaining primitive executable tasks (actions). The task decomposition may contain additional constrains such as task ordering and state conditions. Hierarchical planning is used for example in robotics [1], automated assistance [2], and machine learning [3]. Reverse process to hierarchical planning is *hierarchical plan verification*. The task is to verify whether a given action sequence is a valid hierarchical plan. This means that the action sequence is executable (action preconditions are satisfied in respective states) and it can be obtained by decomposing the goal task (solves the required problem).

There exists a variety of formalisms for *hierarchical planning domain models*, however not all of them show how to handle empty decomposition methods (the task is already achieved and hence it decomposes to nothing) or they do not use extensive constraints (like the prevailing condition). The three formalisms we will focus on in this paper are: *Textbook* formalism [4], *Erol* formalism [5], and the *HDDL* formalism [6] (names given by us). The *Textbook* formalism will be described in detail in Section III as it shows the extensive state constraints (see Section II). The *Erol* formalism was the original hierarchical task network formalism that inspired others. The *HDDL* formalism is an extension to PDDL (Planning Domain Description Language) for expressing hierarchical planning problems. It is currently widely used for example in planning competitions. There is also a formalism by Geier et al. [7] but it does not support the extensive constraints (the original version also does not support empty methods). We will present a formalism that handles them and empty methods and compare it with the three formalisms we mentioned.

II. Formal Background

We use the STRIPS model [8] of actions for hierarchical planning [9], [10]. Let P be a set of propositions describing properties of the world. Then a world state is modeled using a set of propositions that are true in that state (and every other proposition is false in that state). Each action is given by a tuple ($\texttt{pre}^+(a)$, $\texttt{pre}^-(a)$, $\texttt{eff}^+(a)$, $\texttt{eff}^-(a)$), where $\texttt{pre}^+(a)$, $\texttt{pre}^-(a)$, $\texttt{eff}^+(a), \texttt{eff}^-(a) \subseteq P$. Positive (negative) *preconditions* are preconditions that must (cannot) be true in the state for the action to be applicable. Formally action a is applicable to state s if $\texttt{pre}^+(a) \subseteq s$ and $\texttt{pre}^-(a) \cap s = \emptyset$. The last two sets represent the *effects* of an action. The state after applying action a to state s looks like this: $(s \setminus \texttt{eff}^-(a)) \cup \texttt{eff}^+(a)$.

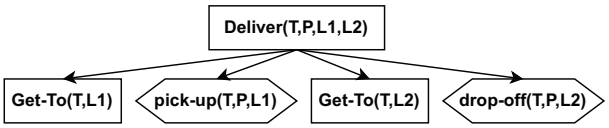

Fig. 1. Example of task decomposition for task Deliver

A compound task describes an activity that can be further decomposed into sub-tasks (compound tasks or actions). This is described using a decomposition method (Fig. 1). A task may have multiple decomposition methods. A decomposition method for a task T decomposing to sub-tasks $T_1, ..., T_k$ under the constraints C can be modeled as a rewriting rule $T \rightarrow T_1, ..., T_k \quad [C]$. The order of sub-tasks is irrelevant as it is described explicitly in C. If sub-tasks in all methods are totally ordered then the domain is called *totally-ordered*.

Let U, V be subsets of tasks from T_1,...,T_k or T in which case the set contains all sub-tasks. We will use constraints as in the *Textbook* formalism [4]:

- $T_i \prec T_j$: an ordering constraint meaning that task T_i is before task T_j.
- *before*(p, U): a precondition constraint meaning that in every plan, the proposition p holds in the state right before the first action to which set U decomposes.

2375-0197/23/$31.00 ©2023 IEEE
DOI 10.1109/ICTAI59109.2023.00057

- *after*(p, U): a postcondition constraint meaning that in every plan, the proposition p holds in the state right after the last action to which set U decomposes.
- *between*(U, p, V): a prevailing constraint such that in every plan, the proposition p holds in all states between the state after the last action to which set U decomposes and the state represented by the first action to which set V decomposes.

Let us look at example of a decomposition of task *Get-To(T,L)*[1] (truck T moves to location L) using the following method:

$$\textit{Get-To(T,L)} \rightarrow \textit{drive(T,L0,L)} \quad [\emptyset] \tag{1}$$

This method decomposes the task directly to an executable action and does not use any constraints. However, what if the truck T is already in location L? This can be naturally modeled using a decomposition method with an empty set (ε) of sub-tasks – a so called *empty method* – that uses *before* constraint to check that the truck is at the required location (notice that the constraint is applied to the task itself):

$$\textit{Get-To(T,L)} \rightarrow \varepsilon \quad [\textit{before(at(T,L), Get-To(T,L))}] \tag{2}$$

Let us present another decomposition method (Fig. 1):

$$\begin{aligned} &\textit{Deliver(T,P,L1,L2)} \rightarrow \textit{Get-To(T,L1), pick-up(T,P,L1),} \\ &\qquad\qquad \textit{Get-To(T,L2), drop-off(T,P,L2)} \ [C] \\ &C = \{\textit{Get-To(T,L1)} \prec \textit{pick-up(T,P,L1)} \prec \textit{Get-To(T,L2)} \\ &\prec \textit{drop-off(T,P,L2)}, \\ &\textit{between(pick-up(T,P,L1), loaded(T,P),drop-off(T,P,L2))}\} \end{aligned} \tag{3}$$

This method is totally ordered and uses a *between* constraint checking that the package is constantly loaded on the truck between actions *pick-up()* and *drop-off()*. This is important as due to task interleaving (explained later) other actions may be inserted while the task *Deliver()* is executed.

The *before* constraints can be added to sub-tasks to model additional conditions specific for the decomposition method. For example we might have a second method that describes the decomposition of the task *Deliver()* for hazardous packages. This new method might have additional *before* constraints for actions *pick-up()* and *drop-off()* that require a safety team to be present: *before(at(L1,Safety),pick-up(T,P,L1))* *before(at(L2,Safety),drop-off(T,P,L2))*

Let us now discuss the *after* constraint. It is important to highlight that the *after* constraint is not the same as an *effect* of an action. The *after* constraint only checks that something is true after a task is completed. Let us provide an example of task *Get-To-Refuel* representing a combined activity of getting to a location and refueling gas. The *after* condition is used to check that the truck is out of gas after movement:

$$\begin{aligned} &\textit{Get–To–Refuel(T,L)} \rightarrow \textit{drive(T,L1,L),} \\ &\qquad\qquad\qquad\qquad \textit{refuel(T,L)} \ \ [C] \\ &C = \{\textit{drive(T,L1,L)} \prec \textit{refuel(T,L)}, \\ &\qquad \textit{after(not(hasgas(T)), drive(T,L1,L),} \\ &\qquad \textit{before(at(L,GasStation), refuel(T,L))}\} \end{aligned} \tag{4}$$

III. Hierarchical Planning Via the Textbook Model

In this section we will describe the *Textbook* formalism and we will show the challenges that empty methods present to this formalism. First, let us define a task network [4].

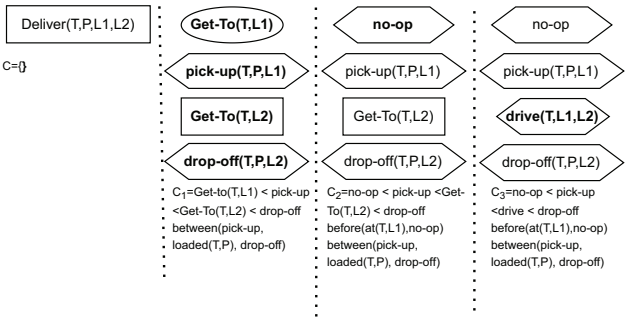

Fig. 2. Example of network decomposition

A *Task network* is a pair $w = (U, C)$, where U is a set of tasks[2] and C is a set of constraints as described in Section II. Task network is *primitive* if every task in it represents an *action*. Let $w = (U, C)$ be a task network, $t \in U$ a task, and m an instance of method decomposing task t into $subtasks(m)$ with constraints $constr(m)$. We can apply the method to a task network creating a new task network:

$$\delta(w, t, m) = ((U \setminus \{t\}) \cup subtasks(m), C' \cup constr(m)) \tag{5}$$

where C' is obtained from C by the following modifications:

- For precedence constraints containing t, we replace t with $subtasks(m)$, i.e., $\forall t' \in subtasks(m)$: $t \prec v$ is replaced by $t' \prec v$ and $v \prec t$ is replaced by $v \prec t'$.
- For *before*, *between*, and *after* constraints on a subset of tasks U' which contains t, we replace U' with $(U' - \{t\}) \cup subtasks(m)$.

Let us now define the *hierarchical planning problem* [4]. Given planning domain model $D = (P, T, M)$, where P is a set of propositions, T is a set of tasks (including actions), and M is a set of decomposition methods, a hierarchical planning problem is formulated as a triple (s_0, w, D), where s_0 is the initial state and w is the initial (goal) task network. A solution to the hierarchical planning problem is a sequence of actions (plan) $\pi = \langle a_1, a_2, \ldots, a_k \rangle$ executable from state s_0 (each a_i

[1]As usual in planning, we allow attributes in tasks. These attributes are substituted by constants describing specific objects, which is called grounding. For simplicity, we assume everything grounded in our formal definitions.

[2]It may happen that the task network contains several identical tasks so it is actually a multi-set of tasks. We assume that tasks are uniquely indexed to distinguish between occurrences of the same task.

is applicable to state s_{i-1}, where s_i is a state after applying action a_i). These actions are from the primitive task network $w' = (\{a_1, a_2, \ldots, a_k\}, C')$ which has been obtained from w by application of decomposition methods:

$w' = \delta(\delta(\ldots\delta(\delta(w, t_1, m_1), t_2, m_2), \ldots), t_n, m_n)$.

Moreover, the constraints C' must be satisfied by plan π as follows:

- $a_i \prec a_j$: $i < j$,
- *before*(p, U): $p \in s_{min\{i|a_i \in U\}-1}$,
- *after*(p, U): $p \in s_{max\{i|a_i \in U\}}$,
- *between*(U, p, V): $\forall j, max\{i|a_i \in U\} \leq j < \{min\{i|a_i \in V\} : p \in s_j$.

One may notice that the above formalism has a problem with empty methods that have state constraints or that decompose a task participating in some constraints. The problem is that function δ simply removes the task from the task network and from all constraints. Assume the following empty method:

$$E \rightarrow \varepsilon \quad [C], C = \{\textit{before(p,E)}\} \tag{6}$$

If this method is applied to a task network, it adds the before constraint but as the task E disappears from the network, it is not clear to which state the constraint should be applied – the constraint in the network looks as *before(p,∅)*.

IV. No-op Based Model

We have shown that the *Textbook* formalism does not correctly handle empty methods. In this section, we propose a modification of the model such that it works well. The idea is that empty methods become regular methods by adding a *no-op()* action. The *no-op()* action is an action that does not modify the world state and is applicable to any state. Formally, the preconditions and effects of the *no-op()* action are empty sets. Here is an example of modified empty method (6):

$$E \rightarrow \quad \textit{no-op()}\ [C]C = \{\textit{before(p,no-op())}\} \tag{7}$$

The *no-op()* action serves as a marker in the plan for state and precedence constraints. As these actions were not part of the original planning domain model, they need to be removed from the plan after checking all the constraints.

Fig. 2 shows a visual decomposition of task *Deliver* (we omitted some task parameters if it was clear to which tasks we refer). We begin with a task network $w = (U, C)$, where U contains one task *Deliver* and C are its constraints. We pick this task and use the decomposition method (3). This creates a new task network $w_1 = (U_1, C_1)$ with four tasks and new constraints shown in the picture. We continue the decomposition until we get to a primitive task network $w_3 = (U_3, C_3)$. Then we find an ordering of the actions that satisfies the ordering constraints (in our example from top to bottom). After omitting the *no-op()* action we get a plan $\pi = \langle$*pick-up,drive,drop-off*$\rangle$ that solves the original problem.

The model with no-op actions works well for planning, where the extra no-op actions can be easily removed from the obtained plan. However, this approach will not work for hierarchical plan verification. We will first define plan verification problem more precisely and then we will explain why the no-op actions cannot be used there naturally.

V. Hierarchical Plan Verification Problem

The hierarchical plan verification problem is defined as a somehow reverse problem to the planning problem. Given a domain model $D = (P, T, M)$, the hierarchical plan verification problem is formulated as a tuple (s_0, g, π, D), where s_0 is the initial state, g is a task network representing the goal task, and π is a plan. A solution to the verification problem is a yes/no answer to the question, whether π is a solution for the planning problem (s_0, g, D). In other words, we are asking whether plan π can be obtained from task network g using decomposition methods M. Sometimes, the goal network is not given and part of the verification problem is also identifying the goal task network consisting of a certain compound task [11]. This problem can be seen as a problem of recognizing which task is being executed by observing a sequence of actions. Now, if we use the no-op based modification of the domain model, there is an issue with hierarchical plan verification as the no-op actions are not part of the plan to be verified. One can try to insert them in the plan to be verified, but obviously it is not clear to which locations the no-op actions should be placed. Therefore, we propose a novel approach to semantics of hierarchical planning models that covers both planning and plan verification problems.

VI. Index-based Model

In this section, we propose the index-based semantics for hierarchical planning models. This semantics is based on an idea of assigning two indexes to each task t, $start(t)$ and $end(t)$, indicating the order number of the first and last action to which the task t decomposes. If the task does not decompose to any real action, which is the case of applying an empty method, the index will be pointing to a certain state between two actions (this is where the no-op action would be placed). We call these indexes half-indexes. Obviously, task indexes are unknown until a complete plan is obtained so we will represent them as variables in the style of constraint satisfaction. The initial domain of these variables will be a set $\{0.5, 1, 1.5 \ldots\}$. The integers represent positions of real actions in the plan, while the "half-numbers" represent the positions in the plan for empty tasks.

Let us now reformulate the notion of task network and application of a method to task network using the indexes. A *Task network* is a pair $w = (U, C)$, where U is a set of tasks annotated by indexes (variables) $start$ and end and C is a set of constraints, which are versions of constraints described in Section II working with indexes rather than tasks (see below). Let $w = (W, C)$ be a task network, $t \in W$ a task, and m an instance of method decomposing task t into $subtasks(m)$ with constraints $constr(m)$. We can apply the method to a task network creating a new task network:

$$\delta(w, t, m) = ((W \setminus \{t\}) \cup subtasks(m), C \cup constr(m) \cup \{c\}) \tag{8}$$

For each new task $t' \in subtasks(m)$ we also create two variables $start(t')$ and $end(t')$ with initial domains $\{0.5, 1, 1.5, \dots\}$. The constraint c connects these variables with the variables for (non-empty) task t as follows: $start(t) = min\{start(t')|t' \in subtasks(m)\}$ and $end(t) = max\{end(t')|t' \in subtasks(m)\}$. If the method is empty, i.e., $subtasks(m) = \emptyset$ we will add a constraint $start(t) = end(t)$ and we will reduce the domain of variables $start(t)$ and $end(t)$ to half-numbers $\{0.5, 1.5, 2.5, \dots\}$. $constr(m)$ are method constraints represented using task indexes as follows:

- $T_i \prec T_j$: $\lfloor$*end(*T_i*)*$\rfloor < \lceil$*start(*T_j*)*$\rceil$,
- *before*(p, U): *before*$(p, \lceil$*start(U)*$\rceil)$,
- *after*(p, U): *after*$(p, \lfloor$*end(U)*$\rfloor)$,
- *between*(U, p, V): *between*$(\lfloor$*end(U)*$\rfloor$,p,$\lceil$*start(V)*$\rceil)$,

where $start(U) = min\{start(t')|t' \in U\}$, which is modelled using a new variable whose value equals the value of the formula. Similarly, $end(U) = max\{end(t')|t' \in U\}$. The use of the ceiling ($\lceil\rceil$) and floor ($\lfloor\rfloor$) function is important due to the half-numbers (see below).

Notice that when modifying the task network, we only remove the decomposed task while keeping its constraints and $start$ and end indexes. We also connect the indexes of the removed task with those for newly added sub-tasks (constraint c). This way, we can propagate information about actions belonging to a given task after obtaining a complete plan.

A primitive task network $w' = (U, C)$, is a solution to the planning problem (s_0, w, D) if w' has been obtained from w by application of decomposition methods, actions in U can be linearly ordered to a plan $\pi =< a_1, a_2, \dots, a_k >$ executable from state s_0 (k is plan length) and all task indexes can be instantiated while satisfying all the constraints:

- $start(t) \leq k + 0.5$, $end(t) \leq k + 0.5$,
- $\forall i : start(a_i) = end(a_i) = i$ (action position in plan),
- *before*(p, I): $p \in s_{I-1}$, *after*(p, I): $p \in s_I$,
- *between*(I, p, J): $\forall l, I \leq l < J : p \in s_l$.

Specifically, notice that all empty tasks are mapped to half-numbers so these tasks are located between real actions. This is similar to having a no-op action there (index 0.5 models no-op at the beginning of plan, and $(k+0.5)$ models no-op at the end of plan). For example, if the index of empty task is 3.5 then its $before$ and $after$ constraints are checked in state s_3 ($3 = \lceil 3.5 \rceil - 1 = \lfloor 3.5 \rfloor$), the task is properly placed between actions a_3 and a_4 ($\lfloor 3 \rfloor < \lceil 3.5 \rceil$, $\lfloor 3.5 \rfloor < \lceil 4 \rceil$), and the task can be properly ordered with another empty tasks placed at the same location ($\lfloor 3.5 \rfloor < \lceil 3.5 \rceil$).

The proposed formalism models the hierarchical planning problem and during task decomposition, it does not require rewriting the set of constraints as it rather adds new constraints. Also, because it models the task network using a constraint satisfaction problem, it is possible to detect some inconsistencies (via constraint propagation) earlier than obtaining the primitive task network. Moreover, it does not require adding no-op actions to model empty tasks explicitly, so the proposed index-based semantics can be used for plan verification over the original plans [12].

VII. Comparisons with other Models

In this section we will compare the proposed formalism with other models. The Erol formalism [5] introduces two types of compound tasks: a regular compound task and a goal task. They define goal task as properties that we wish to make true in the world. They define Compound tasks as tasks that denote desired changes that involve several goal tasks and primitive tasks. They then describe how to use empty methods only for goal task but not for compound tasks. As for constraints it allows all four types of constraints but it only allows for one task to be part of the constraints (two for *between* constraints), not multiple tasks as our model.

Compared to the formalism of the Textbook model [4] our formalism uses the same constraints but we showed how to use empty methods as it is not clear how one would use empty methods in their transformation of networks.

The main differences between HDDL [6] and our formalism are in the constraints. The HDDL *before* condition requires that the condition must be true before the entire method. We allow *before* conditions for a subset of sub-tasks in the method not necessarily all of them. Another big difference is when a *before* condition is checked. In the formalism provided here we check the condition in the state immediately preceding the task (or tasks this condition relates to), but in the HDDL formalism it is only required that the condition is true sometime before (we shall call these *sometime before* conditions) the task starts (and it could have changed before the task actually executes). This is because some of the planners handle conditions by adding a new primitive task before the method with given conditions. This works just like our conditions in totally ordered domains but not in partially ordered domains. We don't see much practical use for the *sometime before* condition as it does not ensure that the condition is true when the task begins. Another difference is in *between* conditions that are not defined in HDDL. The *after* condition is an intuitive counterpart to the *before* condition. While we have provided its example (4), the *after* condition is not commonly used. One use of the *after* condition could be for checking that something is true after the goal task. This is akin to a goal state in classical planning. While HDDL does not support after conditions they do allow for a goal state to be defined.

VIII. Modifications of Model Properties

In previous sections we described the differences between some of the formalism describing hierarchical domain models. In this section we will describe possible transformations of some of the models properties. This is also useful because not every planner or verifier is able to handle the full extent of the models. Often they use some simplification or transformation of the model into an alternative one or easier one.

One might look at partially and totally ordered domains and wonder if the partially ordered domains can be transformed into totally ordered domains by trying all the possible orderings. For example, for a task decomposition $T \rightarrow M, N$ we may create two possible decompositions: $T \rightarrow M, N[C]$, where $C = \{M \prec N\}$ and $T \rightarrow M, N[C]$, where $C = \{N \prec$

$M\}$. This will not work, because partial ordering allows task interleaving (Fig. 3) while total ordering does not.

Not every planner or verifier is able to handle all of the constraints mentioned. In fact only one verifier [12] is able to handle all four types of constraints. Also most verifiers can only handle *before* conditions before the entire method (not for a subset of sub-tasks). So in this section we want to provide a couple of transformations between various constraints. For example, for totally ordered domains we can transform the *before* condition on a set of sub-tasks into a *before* condition for one sub-task only:

$$T \rightarrow M, N, O[C], C = \{M \prec N, N \prec O, \textit{before(p,\{N,O\})}\} \tag{9}$$

As we know the order of tasks in set $\{N, O\}$, the before condition can be applied to sub-task N only: *before(p,N)*. There is a transformation available to move the *before* condition in front of the entire task N (to make it compatible with *HDDL*) instead of in front of a sub-task (see Fig. 4), but it requires adding a new task N'. This new task has to be added because there might be another method that decomposes into N that does not require this precondition. The upper limit of new tasks created is $min(c, t)$, c is the number of *before* conditions and t is the number of sub-tasks. Because if there are more conditions than tasks then at least two conditions relate to the same sub-task (using the first procedure of transforming a *before* condition from multiple sub-tasks to one).

We can also transform a *between* condition into a series of *before* conditions in totally ordered domains (see (10)).

$$\begin{aligned} &T \rightarrow M, N, O, P[C] \\ &C = \{M \prec N, N \prec O, O \prec P, \textit{between(p,\{N,P\})}\} \end{aligned} \tag{10}$$

The condition must be true between sub-tasks N and P, this means it must be true before sub-tasks O and P. We can therefore transform it into two *before* conditions: *before(p,O)* and *before(p,P)*. We could then also transform these to be before the whole task and not before sub-tasks using the process described previously.

In *totally ordered* domains the *after* condition on a single sub-task can also be transformed into a *before* condition on the following sub-task if there is a following sub-task. So *after(p,N)* would be transformed into *before(p,O)*. This cannot be done on partially ordered domains due to interleaving. If the *after* condition is on a set of subtasks we can transform it to an *after* condition on one sub-task analogously to the technique described for *before* conditions (except we focus on the last sub-task and not the first).

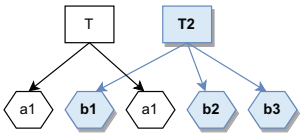

Fig. 3. Task interleaving

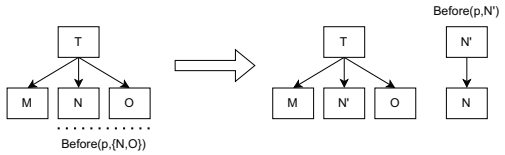

Fig. 4. Example of transformation of before condition

IX. Conclusion

In this short paper we proposed a novel semantics for hierarchical domain models with focus on how to handle empty methods while keeping extensive constraints. We then compared it with other formalisms used in hierarchical planning. Finally we listed some model modifications for different constraints. We think this is of special interest as not many of the constrains described are handled by current planners or verifiers.

X. Acknowledgments

Research is supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. S. Ondrčková is supported by the Charles University project GA UK number 280122 and by SVV project number 260 698.

References

[1] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *IROS 2011*. IEEE, 2011, pp. 1470–1477.

[2] P. Bercher, G. Behnke, M. Kraus, M. Schiller, D. Manstetten, M. Dambier, M. Dorna, W. Minker, B. Glimm, and S. Biundo, "Do it yourself, but not alone: *Companion*-technology for home improvement – bringing a planning-based interactive DIY assistant to life," *Künstliche Intelligenz – Special Issue on NLP and Semantics*, vol. 35, pp. 367–375, 2021.

[3] F. Mohr, M. Wever, and E. Hüllermeier, "ML-plan: Automated machine learning via hierarchical planning," *Machine Learning*, vol. 107, no. 8, pp. 1495–1515, 2018.

[4] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice*. Elsevier, 2004.

[5] K. Erol, J. A. Hendler, and D. S. Nau, "Umcp a sound and complete procedure for hierarchical task-network planning," in *Aips*, 1994, pp. 249–254.

[6] 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 *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 06, 2020, pp. 9883–9891.

[7] T. Geier and P. Bercher, "On the decidability of htn planning with task insertion," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, no. 3, 2011, p. 1955.

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

[9] K. Erol, J. A. Hendler, and D. S. Nau, "Complexity Results for HTN Planning," *Annals of Mathematics and AI*, vol. 18, no. 1, pp. 69–93, 1996.

[10] P. Bercher, R. Alford, and D. Höller, "A survey on hierarchical planning – one abstract idea, many concrete realizations," in *IJCAI 2019*. IJCAI, 2019, pp. 6267–6275.

[11] R. Barták, A. Maillard, and R. C. Cardoso, "Validation of hierarchical plans via parsing of attribute grammars," in *ICAPS*. AAAI Press, 2018, pp. 11–19.

[12] S. Ondrčková, R. Barták, P. Bercher, and G. Behnke, "On heuristics for parsing-based verification of hierarchical plans with a goal task," in *Proceedings of the 35th International Florida Artificial Intelligence Research Society Conference (FLAIRS 2022)*, 2022.