33rd International Conference on
Automated Planning and Scheduling

July 8–13, 2023, Prague, Czech Republic

HPlan 2023

Proceedings of the 6th ICAPS Workshop on

Hierarchical Planning

Program Committee

Ron Alford	The MITRE Corporation
Gregor Behnke	University of Amsterdam
Pascal Bercher	the Australian National University
Susanne Biundo	Ulm University
Kutluhan Erol	Izmir University of Economics
Robert Goldman	Smart Information Flow Technologies (SIFT)
Daniel Höller	Saarland University
Jane Jean Kiam	Universität der Bundeswehr München
Ugur Kuter	Smart Information Flow Technologies (SIFT)
Pascal Lauer	Saarland University
Rouxi Li	University of Maryland, College Park
Songtuan Lin	the Australian National University
Felipe Meneguzzi	University of Aberdeen
Conny Olz	Ulm University
Simona Ondrčková	Charles University
Sunandita Patra	IIT Kharagpur
Damien Pellier	Laboratoire d'Informatique de Grenoble
Xing Tan	Lakehead University
Julia Wichlacz	Saarland University

Organizing Committee

Pascal Bercher	The Australian National University
Daniel Höller	Saarland University
Julia Wichlacz	Saarland University
Ron Alford	The MITRE Corporation

Preface

The motivation for using hierarchical planning formalisms is manifold. It ranges from an explicit and predefined guidance of the plan generation process and the ability to represent complex problem solving and behavior patterns to the option of having different abstraction layers when communicating with a human user or when planning cooperatively. This led to numerous hierarchical formalisms and systems. Hierarchies induce fundamental differences from classical, non-hierarchical planning, creating distinct computational properties and requiring separate algorithms for plan generation, plan verification, plan repair, and practical applications. Many techniques required to tackle these – or further – problems in hierarchical planning are still unexplored.
With this workshop, we bring together scientists working on many aspects of hierarchical planning to exchange ideas and foster cooperation.

This year, in the 6th edition of the HPlan workshop, we accepted 8 papers, one of which was also accepted at SoCS 2023. As in all previous editions, we have a very high-quality review process, with 3 reviewers for most papers, and 4 for a few. We again had one conditional accept (that was accepted in the end) with another round of reviewing after the demanded changes were done.

Two papers are challenge papers, whereas all others are regular scientific papers. One of the challenge papers proposes an HTN formalism with time, with the aim at developing HDDL 2.1, an extension of the (still young) HDDL standard. The other explores the potential of Hierarchical Task Networks (HTN) and Hierarchical Domain Definition Language (HDDL) in enhancing safety in single-pilot operations (SPOs) by encoding private pilots' maneuvers and addressing challenges in onboard companion technologies. The scientific papers again cover a large variety of topics. One paper is concerned with *learning of task parameters* in the context of learning HTNs. Several papers are concerned with *solving HTN planning tasks*. For HTN planning with time, one paper proposes a compilation technique for obtaining novel heuristics by making use of existing heuristics for non-temporal planning. For non-temporal HTN planning, one paper deploys deep learning for heuristic design and predicting effects of compound tasks. The last paper concerned with solving HTN problems is also making contributions to heuristic search. It also exploits effects (and preconditions) of compound tasks, but uses them for pruning dead-ends in a progression search rather than for heuristic computation. Another paper deals with the scenario where planning fails due to execution errors and proposes a *plan repair* technique to deal with them. Once a (potential) solution was found, one might want to verify that it is indeed a solution to the given problem. One paper investigates the *computational complexity* of this verification (and whether it's an optimal solution), both for HTN and classical planning.

Pascal, Daniel, Julia, Ron
HPlan Workshop Organizers,
July 2023

Invited Talk

Each year so far we had one or two invited talks. This year, by *Héctor Munoz-Avila*.

Automated Learning of Hierarchical Knowledge for Planning and Acting

In this talk I will cover a variety of algorithms for learning hierarchical knowledge for planning developed together with collaborators over the years. This will include a variety of formalisms ranging from hierarchical task networks (HTNs) to hierarchical reinforcement learning. I will also present our studies on a variety of planning settings including deterministic planning, nondeterministic planning and planning and acting with a two-level architecture of feedforward neural networks. I will show results of empirical evaluation versus various baselines as well as theoretical analysis regarding the expressiveness of the resulting algorithms and architectures. This work has been published in AAAI, IJCAI, AAMAS and ACS conferences, among others.

Bio

Dr. Munoz-Avila is a Program Director at NSF's Information and Intelligent Systems (IIS) Division, where he is cluster lead for the Information Integration and Informatics (III) program. Prior to joining NSF, Dr. Muñoz-Avila was a (tenured) professor of Computer Science and Engineering and of Cognitive Science at Lehigh University. He was founding co-director of Lehigh's Institute for Data, Intelligent Systems, and Computation (I-DISC). Dr. Muñoz-Avila is recipient of a National Science Foundation (NSF) CAREER and held a Lehigh Class of 1961 Professorship. He has been chair for various international scientific meetings including the Sixth International Conference on Case-Based Reasoning (ICCBR-05) and the twenty-fifth Innovative Applications of AI Conference (IAAI-13). He was funded by the Office of Naval Research (ONR), the National Science Foundation (NSF), the Defense Advanced Research Projects Agency (DARPA), the Naval Research Laboratory (NRL) and the Air Force Research Laboratory (AFRL).

Table of Contents

Scientific Papers

Challenge Papers

Extracting Hierarchical Task Networks Parameters from Demonstrations

Philippe Hérail, Arthur Bit-Monnot

LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France

Abstract

Hierarchical Task Networks (HTNs) are a common formalism in automated planning. However, HTN models are mostly designed by hand by expert users. While many of the state-of-the-art approaches for learning HTN try and learn the structure and its parameterization in a single step, other focus specifically on learning the structure of the model.

Many of these structure-focused approaches, however, learn models with non-parameterized actions, task and methods, which limits their generalization capabilities. In this paper, we propose a constraint satisfaction-based approach for extracting parameters for a given HTN structure using a set of demonstration traces.

Introduction & Motivation

Hierarchical Task Networks (HTNs) are a common planning formalism that hierarchically decomposes abstract tasks until they are refined into a sequence of executable primitive actions. However, these models are difficult to design for non-expert users, and several approaches have been developed to learn HTNs instead of handcrafting them.

Several of the current approaches to HTN learning (Hogg, Muñoz-Avila, and Kuter 2008; Zhuo, Muñoz-Avila, and Yang 2014) do not generate new abstract tasks as part of the learning process but instead rely on annotated tasks to provide intermediate tasks in the hierarchical structure. Some systems that try and extract new intermediate tasks automatically learn models that are non-parameterized, which limits their generalization capabilities (Li et al. 2014; Chen et al. 2021). Segura-Muros, Pérez, and Fernández-Olivares (2017) propagate the arguments upwards in the hierarchy, but it is unclear how their technique would work with recursive tassk or how it would scale to more complex hierarchies. In another research area, that of program synthesis (Manna and Waldinger 1971), Programming By Example (PBE) approaches (Raza and Gulwani 2018; Dong et al. 2022), also need to map parameters to examples. However, they rely on a Domain Specific Language (DSL) that constrains the set of arguments in the structure of the possible programs and do not have to determine the structure of the model parameters, relying instead on human input.

We argue that an HTN learning system able to infer new abstract tasks should be able to automatically parameterize the resulting structure to generalize to new environments with similar characteristics. We therefore take the view that HTN learning may be split in three steps: structure, parameters and preconditions learning. To tackle the second step, namely learning parameters of a fixed structure, we propose a MAX-SMT (Nieuwenhuis and Oliveras 2006) approach that exploits set of demonstration traces for a given top level task t_{demo} and the decomposition trees of t_{demo} into the traces.

Learning Problem

Hierarchical Task Networks

In this paper, we consider HTNs as a tuple $H = (\mathcal{T}, \mathcal{A}, \mathcal{M})$ where $\mathcal{T}$ is a set of abstract tasks, $\mathcal{A}$ a set of primitive actions and $\mathcal{M}$ a set of possible methods decomposing the tasks $t \in \mathcal{T}$ into ordered subtasks $\{t_d \mid t_d \in \{\mathcal{T} \cup \mathcal{A}\}\}$.

A primitive task (or action) $a \in A$ models the basic acting capabilities of the agent, and represents a directly executable command. They are represented using an identifying symbol and a set of parameters, such as $a = \text{action\_name}(\text{arg}_1, \ldots, \text{arg}_n)$. Actions are associated with preconditions and effects that enable verifying the validity of a plan.

An abstract (or non-primitive) task $t \in \mathcal{T}$ is associated with a set of methods $\mathcal{M}_t$ that allow decomposing it. Similar to actions, they are represented using an identifying symbol and a set of arguments.

A method $m \in \mathcal{M}_t$ is associated with a symbol and a set of arguments, like abstract and primitive tasks. The method's preconditions are denoted as Pre_m. The method's subtasks, N_m, is a totally ordered sequence of subtasks in $\{\mathcal{T} \cup \mathcal{A}\}$, representing a possible decomposition of t. This totally-ordered task network represents a way to achieve the task t and is only applicable in the current state if its preconditions Pre_m hold.

Inputs & Objectives

We consider as input the structure of an HTN model, its primitive actions defined with their parameters, a set $\mathcal{T}_{dem}$ of known abstract tasks to be demonstrated and a set $\mathcal{D}$ of demonstration traces made of a sequence of ground actions. Each trace demonstrates a ground instance of a given task t_{dem}, for which the parameters are known and fixed. New abstract tasks $t, t \notin \mathcal{T}_{dem}$ will be called *synthetic* tasks.

Furthermore, we consider that a decomposition tree mapping each demonstrated task $t_{dem} \in \mathcal{T}_{dem}$ to each demonstration trace is available. Note that such decompositions can routinely be obtained using HTN planning, as used for plan verification (Höller et al. 2021), or through parsing techniques (Li et al. 2014).

For simplicity, we consider that the structure of the HTN model does not contain any of the demonstration tasks as a method's subtask, i.e., that the model is non-recursive through these demonstration tasks. A proposition to remove this assumption is presented at the end of this paper.

The goal of the parameter learner is to capture the relationship between the arguments of the subtasks of the hierarchy, both vertically (across levels) and horizontally (between siblings tasks in a single method). This in turn will enable a solver to efficiently use the model for planning, and may be used to extract useful preconditions.

Approach to Parameter Learning

In order to parameterize an HTN, our algorithm is decomposed into two main steps:

1. The identification of the set of candidate parameters for all non-parameterized abstract tasks and methods in the domain.
2. The simplification of this set of parameters. This is done by identifying, in the ground examples of the demonstrations, the usage patterns of the hierarchy in order to infer possible unifications of candidate parameters.

Identification of the possible parameters

To identify the superset of possible parameters, we defined properties for this set, listed below. These properties where based on our observation of domains from the International Planning Competition (IPC).

1. The parameters of a method m must allow to set the parameters for all its subtasks, and $|\operatorname{args}(m)|$ must be finite.
2. Each parameter of a synthetic task must be used in at least one of its methods.

We designed algorithm 1 following these properties, which propagates arguments from methods' subtasks to their parent tasks until no new argument can be extracted.

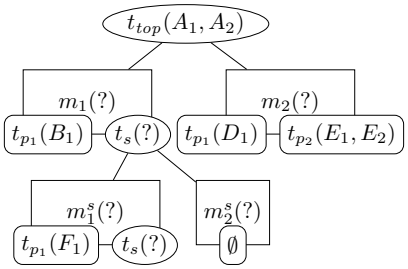
The propagation of arguments upwards would require defining a parent task for the whole hierarchy, which is difficult in the presence of recursive task definitions. To solve this issue, the function EXTRACT SUBHIERARCHIES (Alg. 1, line 2) takes each unique abstract task symbol t and convert it into a basic self-contained hierarchy containing only the task t_h, with symbol t as top level task, its methods $\mathcal{M}_t$ and the direct subtasks of each method $m \in \mathcal{M}_t$.

The decomposition into subhierarchies is illustrated in Figure 1, with t_{top} being a demonstrated top level task with fixed arguments, t_s an abstract task with unknown arguments and the t_{p_i} being primitive tasks. The ? symbol is here used to denote an unknown set of arguments for a given task or method.

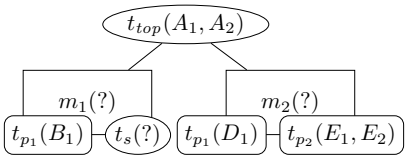
Algorithm 1: Parameter Superset Generation

```
1:  H ← HTN model structure
2:  H_subs ← EXTRACT SUBHIERARCHIES(H)
3:  repeat
4:    P ← args(H_subs)                          ▷ Existing parameters
5:    for all h_sub ∈ H_subs do
6:      h_sub ← PROPAGATE ARGS UPWARDS(h_sub)
7:    end for
8:    P_new ← args(H_subs) \ P                  ▷ New parameters
9:    for all h_sub ∈ H_subs do
10:     h_sub ← UPDATE SUBTASKS ARGS(h_sub, P_new)
11:   end for
12: until P_new = ∅
```

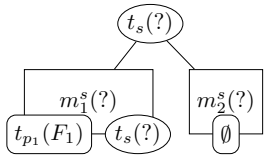
This decomposition into subhierarchies has the added benefit of permitting the parameter extraction process to work simultaneously with demonstrations for different top level tasks $t_{top} \in \mathcal{T}_{dem}$.



(a) Example task hierarchy, showing how t_{top} can be decomposed using the methods m_1 and m_2.



(b) Subhierarchy for t_{top}.



(c) Subhierarchy for t_s.

Figure 1: Example HTN structure and corresponding subhierarchies. Tasks in rounded rectangle boxes, such as t_{p_1}, represent primitive tasks while oval boxes, such as t_s represent abstract tasks. The $\emptyset$ as the only subtask in a method represent a method that rewrites to no subtask.

As recursive tasks could in theory propagate new arguments upwards infinitely, we need to enforce the termination of our algorithm in this case. To this end, we need to know the propagation history of an argument. We therefore augment each parameter p of a task or method in a subhierarchy with the set $\hat{\mathcal{M}}_p$ of methods through which it has been

propagated upwards, so that $\hat{p} = (p, \hat{\mathcal{M}}_p)$.

Algorithm 2: PROPAGATE ARGS UPWARDS(h_{sub})

```
1: for all m ∈ M_t do
2:   for all p̂ ∈ args(SUBTASKS(m)) do
3:     args(m) ← args(m) ∪ {p̂}
4:     if m ∉ M̂_p then
5:       p̂' ← (p, M̂_p ∪ {m})
6:       args(t_h) ← args(t_h) ∪ {p̂'}
7:     end if
8:   end for
9: end for
```

Algorithm 2 details the procedure used to propagate parameters from the subtasks to the methods and top level task of a given subhierarchy h_{sub}. Line 3 propagates all the arguments from the subtasks of m into the $\text{args}(m)$. Then, the condition on line 4 makes the methods act as filters, preventing parameters that already crossed this method's boundary from reaching the top level task once again. This guarantees termination in case of recursive task definitions. In such a case, if a task of symbol t is defined with n recursive instantiation of itself in its methods' subtasks, it will end up with $n+1$ sets $\mathcal{P}_i^t$ of potential parameters: $\mathcal{P}_0^t$ would contain the parameters induced by all the non-recursive subtasks, and the remaining sets $\mathcal{P}_i^t, i \in [1, n]$ will contain the parameters used in each recursive subtask instantiation. We argue that it is a reasonable limitation as an HTN planner can:

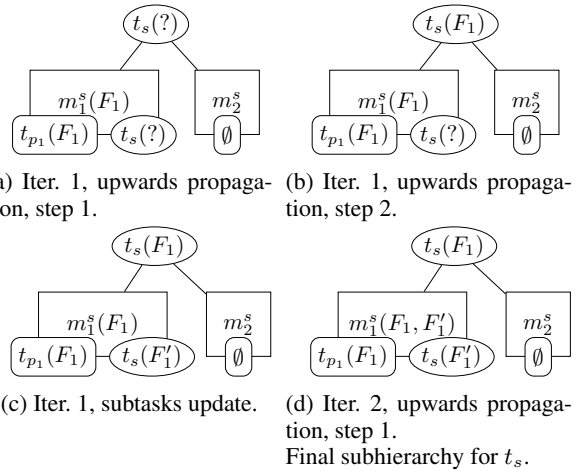
1. Parameterize all the non-recursive subtasks.
2. For each recursive subtask instantiation, choose whether the parameters are the same as the parent task or not.

Algorithm 3: UPDATE SUBTASKS ARGS($h_{sub}, \mathcal{P}_{new}$)

```
1: for all t_s ∈ SUBTASKS(h_sub) do
2:   t ← sym(t_s)
3:   P^t_new ← {p̂ | p̂ ∈ args(t) ∧ p̂ ∈ P_new}
4:   for all p̂ ∈ P^t_new do
5:     args(t_s) ← args(t_s) ∪ p̂
6:   end for
7: end for
```

Finally, Algorithm 3 presents the procedure to update the subtasks, called after each round of argument propagation. It is a straightforward procedure that is used to keep a consistent parameterization of every abstract task.

Figure 2 illustrates the parameter generation using Algorithm 1, focusing specifically on the subhierarchy for t_s from the example presented Figure 1, as it is independent of any other subhierarchies. Figures 2a and 2b shows the effect of the function PROPAGATE ARGS UPWARDS while Figure 2c shows the update of the subtasks. Note that due to the recursive nature of t_s, the added parameter during the subtasks update is F_1', as it may or may not be bound to F_1. This process is then repeated, as shown in Figure 2d. This time however, the filtering condition for the argument propagation (Alg. 2, line 4) is triggered by F_1', preventing it from being added to the parameters of t_s. As no new changes can be made to the subtasks of t_s (even considering the subhierarchy for t_{top}), all the possible arguments of this subhierarchy have been extracted.



(a) Iter. 1, upwards propagation, step 1.
(b) Iter. 1, upwards propagation, step 2.
(c) Iter. 1, subtasks update.
(d) Iter. 2, upwards propagation, step 1.
Final subhierarchy for t_s.

Figure 2: Example of argument superset generation for t_s.

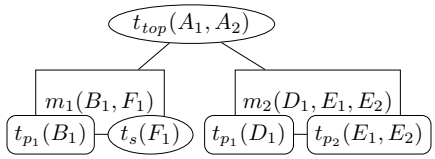
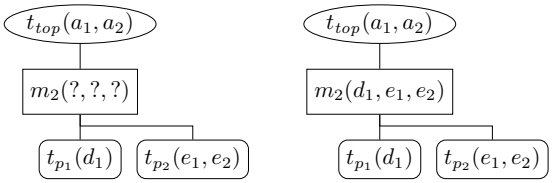


Figure 3: Extracted parameters for t_{top}

Figure 3 shows the resulting parameters for the task t_{top} after applying the same parameter extraction procedure.

From this parameterized HTN, we can easily extract parameterized decomposition trees by replacing argument instantiations in the primitive actions and the demonstrated top level tasks and propagating these substitutions throughout the tree. A basic example of decomposition tree is given in Figure 4, where a_1, a_2, d_1, e_1 and e_2 represent constants. These decomposition trees will be used to simplify the set of task and method parameters from the demonstration examples.



(a) Before substitutions.
(b) After substitutions.

Figure 4: Example of argument propagation in a decomposition tree for a demonstration of $t_{top}(a_1, a_2)$ as the sequence $t_{p_1}(d_1) \rightarrow t_{p_2}(e_1, e_2)$.

Recursive Task Definition: Specific Considerations The extraction and substitution procedure described in the previ-

ous section would actually lead to poor parameterization in the case of recursive tasks definitions, allowing the top level arguments to only refer to the first or second instantiation in the recursive chain.

A common usage of recursive task definitions is to encode the "do *something* until *condition*" pattern, which would be difficult to parameterize without considering the last step of the recursion. The ubiquitous $goto(L_1, L_d)$ pattern, presented Figure 5, is an example of such a pattern used in many planning domains, used to move an agent from a location L_1 to a location L_d. This is done recursively by chaining $move$ actions through intermediate locations until the agent arrives at L_d, mainly to obey location connection preconditions. As can be seen in this example, the L_i parameter is used to constrain the next instantiation of $goto$ and the L_d parameter constrains all of them.

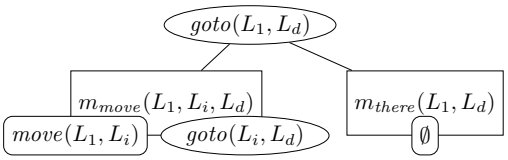
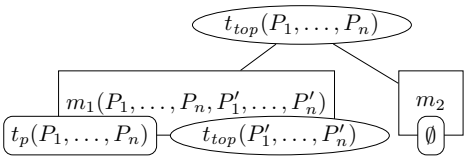
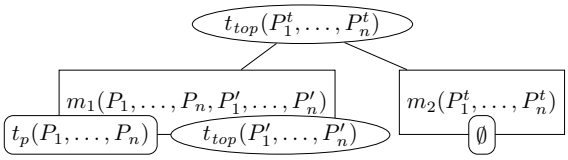


Figure 5: Subhierarchy for a $goto$ pattern. Preconditions omitted for clarity.

To solve this issue, we propose a small modification of the extracted parameters for recursive subhierarchies, presented Figure 6, as well as a preprocessing step leveraging the demonstrations, before extracting the full parameterized decomposition trees presented earlier. This process will be illustrated using a simple subhierachy structure, but could be easily generalized to any task with a single recursive subtask. While this covers many of the use cases, more work is needed for this to work on arbitrary task hierarchies. The main idea is to be able to map parameters of the top task of a given recursive subhierarchy to parameters from the demonstrations' primitive actions, while considering that recursive tasks should be able to refer to parameters at the end of a recursion.



(a) Basic recursive hierarchy.



(b) Modified recursive hierarchy.

Figure 6: Parameters modification for recursive tasks.

We first modify the parameters to keep track of the non-recursive parameters from which the recursive one has been generated, modifying the extracted structures from the previously extracted one, presented Figure 6a, into the one presented in Figure 6b. In this example the P^t_i parameters shows that this parameter originated from the P_i parameter of the task t_p, but we do not know whether it should refer to the immediate instantiation of P_i or if it needs to refer to its last instantiation. The P'_i are the instantiation of the parameters of the task t_{top} in the recursion chain, generated in the same way as in the example Figure 2.

We then can substitute the ground parameters in the non-recursive subtasks in each recursion chain, as presented in Figure 7 where all the p^j_i represent constants.

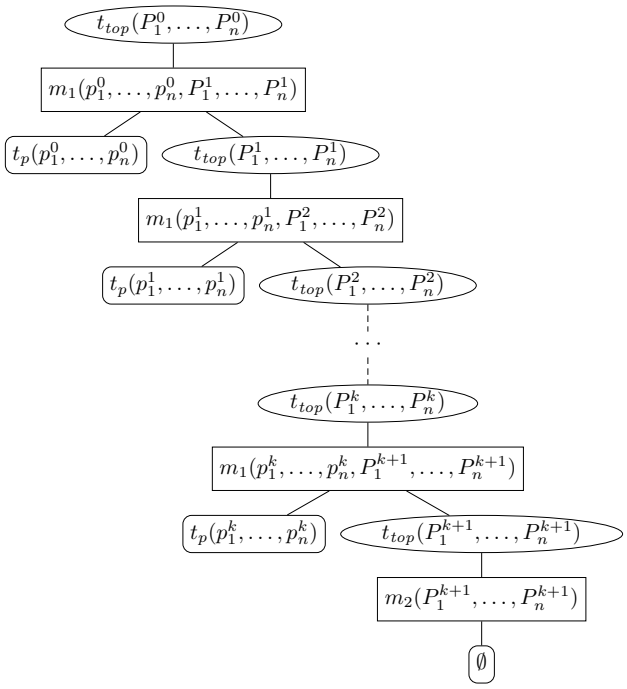


Figure 7: Generic decomposition tree for a recursive hierarchy.

Applying this process to a $goto$ task for which we want to learn the parameters, we obtain the subhierarchy presented in Figure 8. A decomposition tree for a given example demonstration trace is given in Figure 9. This task will be used as a running example to illustrate the remainder of this section.

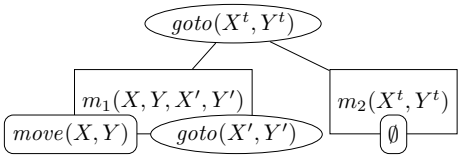
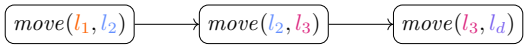
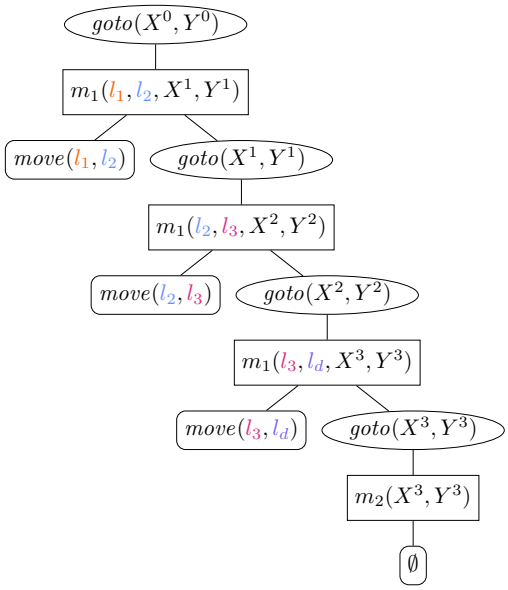


Figure 8: Extracted subhierarchy for a $goto$ task before recursion processing.

Allowed Structures We then need to determine, $\forall i \in [1, n]$, if P^t_i is bound to P_i or P'_i, or to the parameters of the



(a) Demonstration trace.



(b) Decomposition tree.

Figure 9: Possible example trace and corresponding decomposition tree example for the *goto* task. Colors are used to highlight identical constants.

last step of the chain, noted P_i^L. Furthermore, we want to know if some parameters are bound together in the method, transferring information from one step of the chain to the next. We note P_i^+ the instantiation of the parameter P_i in the next step and $\mathcal{P}$ the set of all arguments in the example and the subhierarchy. $\mathcal{P}_{\text{Gnd}} \subset \mathcal{P}$ represents the set of ground arguments, $\mathcal{P}_m \subset \mathcal{P}$ represents the set of lifted arguments of a method m in the considered subhierarchy and $\mathcal{P}_{\text{top}} \subset \mathcal{P}$ the set of lifted arguments of the top level task.

Leveraging the structure of the subhierarchies and the demonstrations, we cast the problem of grouping parameters together as a MAX-SMT problem with the goal of optimizing the size of the groupings of method parameters and the number of top level task parameters bound to their last instantiation in a recursion chain.

From the presented model in Figure 6b, we can extract the following structural constraints where hard constraints represent properties that must hold for the model to be consistent:

$$\forall i \in [1, n]$$

$$\textsc{Hard}(P_i^t = P_i \vee P_i^t = P_i') \tag{1a}$$

$$\textsc{Hard}(P_i^t = P_i' \Rightarrow (P_i^t = P_i^L \vee P_i' = P_i^+)) \tag{1b}$$

$$\textsc{Hard}(P_i^t = P_i^L \Leftrightarrow P_i' = P_i^L) \tag{1c}$$

$$\textsc{Hard}(P_i^t = P_i \Leftrightarrow P_i' = P_i^+) \tag{1d}$$

$$\textsc{Soft}(P_i^t = P_i^L) \tag{1e}$$

Constraints 1a and 1b are used to enforce consistency with the origin of a top task parameter P_i^t. Constraint 1a enforces the fact that the top level task argument either comes from the non-recursive subtasks (left) or the recursive instantiation (right) while constraint 1b enforces the fact that a top level parameter may only propagate information towards the next step in a recursion or towards the last one.

The constraints 1c and 1d are used to enforce consistency within the model generated by the constraint satisfaction solver.

The soft constraint 1e encodes the desirable, but not necessary, property that an argument is always passed recursively which avoids the need of the planner to non-deterministically choose its value.

Compatibility with Examples Equations 1*, are the constraints that must hold to ensure that parameter passing is sensible in a hierarchy.

We can also extract the constraints defined in equations 2$*$ from each example with a structure as presented in Figure 7. We note $\forall i \in [1, n], \forall s, s' \in [0, k] \cup \{L\}, p_i^{s',s}$ the argument $p_i^{s'}$ considered at step s, in order to allow parameters to refer to independent constants at each step.

$$\forall i \in [1, n]$$

$$\textsc{Hard}(P_i^L = p_i^{k,L}) \tag{2a}$$

$$\forall s \in [0, k[, \textsc{Hard}(P_i^+ = p_i^{s+1,s}) \tag{2b}$$

$$\forall s \in [0, k]$$

$$\textsc{Hard}(P_i = p_i^{s,s}) \tag{2c}$$

$$\textsc{Hard}\left(\begin{array}{l}\forall s' \in [0, k] \cup \{L\}, \forall j \in [1, n], \\ \text{sym}(p_i^{k,L}) \neq \text{sym}(p_j^{s,s'}) \Rightarrow p_j^{k,L} \neq p_j^{s,s'}\end{array}\right) \tag{2d}$$

$$\textsc{Hard}\left(\begin{array}{l}\forall s', s'' \in [0, k], \forall j \in [1, n], \\ \text{sym}(p_i^{s',s}) \neq \text{sym}(p_j^{s'',s}) \Rightarrow p_i^{s',s} \neq p_j^{s'',s}\end{array}\right) \tag{2e}$$

When considering our *goto* task, some possible constraints that ensure consistency with the example presented in Figure 9 are presented in the next paragraphs.

Equation 2a defines the binding for the last instantiation of each top task parameter as presented in the following equation:

$$\left\{X^L = l_3^L \quad Y^L = l_d^L\right\} \tag{3}$$

Equation 4 shows the bindings from steps 0 and 1 in the decomposition tree, showcasing the effect of the equations 2b and 2c.

$$\left\{\begin{array}{ll} X = l_1^0 & X' = l_2^0 \\ Y = l_2^0 & Y' = l_3^0 \end{array}\right\}_0 \quad \left\{\begin{array}{ll} X = l_2^1 & X' = l_3^1 \\ Y = l_3^1 & Y' = l_d^1 \end{array}\right\}_1 \tag{4}$$

Finally, equation 5 shows the action of constraint 2d, preventing unsound unifications involving the last instantiation of a given task parameter.

$$\left\{l_3^L \neq l_1^0 \quad l_3^L \neq l_2^0 \quad l_3^L \neq l_2^1 \quad l_3^L \neq l_d^1 \quad l_3^L \neq l_d^L\right\} \tag{5}$$

Minimizing Method Parameters through Unification To determine which parameters are bound together during the optimization process, we define a set $\mathcal{G}$ of potential groups for each $p \in \mathcal{P}$ and a function PGROUP (equation 6a) which maps each unique parameter to a single group (equation 6c). We also define a function NOTCOUNTG (equation 6b) which will be used in the definition of the optimization objectives and is defined through the constraint presented in equation 6d.

$$\text{PGROUP} : \mathcal{P} \to \mathcal{G} \tag{6a}$$

$$\text{NOTCOUNTGROUP} : \mathcal{G} \to \{0, 1\} \tag{6b}$$

HARD

$$\begin{pmatrix} \forall p_1, p_2 \in \mathcal{P} \\ \text{PGROUP}(p_1) = \text{PGROUP}(p_2) \Rightarrow p_1 = p_2 \end{pmatrix} \tag{6c}$$

HARD

$$\begin{pmatrix} \forall g \in \mathcal{G}, \text{NOTCOUNTG}(g) \\ \quad \Leftrightarrow \begin{cases} \nexists p \in \mathcal{P}_{m_1}, \text{PGROUP}(p) = g \\ \vee\, \exists p \in \mathcal{P}_{\text{top}}, \text{PGROUP}(p) = g \end{cases} \end{pmatrix} \tag{6d}$$

We define the objectives for our optimization problem in equation 7 with $\mathcal{C}_{\text{Soft}}$ designating the set of soft constraints. These objectives are considered in lexicographic order.

The first optimization objective (eq. 7a) is used to satisfy two of our objectives: i) grouping method arguments together, to allow transferring information from one step of the recursion to the next and ii) binding subtask arguments to top level tasks arguments.

The second optimization objective (eq. 7b) is used to satisfy the constraints binding top level arguments to the instantiation of arguments in the last step of recursion (eq. 1e) in order to transfer information throughout the recursion.

$$\max \sum_{p \in \mathcal{P}_{m_1}} \text{NOTCOUNTG}(\text{PGROUP}(p)) \tag{7a}$$

$$\max \sum_{c \in \mathcal{C}_{\text{Soft}}} \text{SATISFIED}(c) \tag{7b}$$

Solving this problem will generate a set of equivalence classes. We then replace each of these classes in the modified subhierarchy with a single new parameter, unifiying parameters with their right instantiation.

Considering again our *goto* task example, solving the associated MAX-SMT problem will produce the equivalence classes presented equation 8. Replacing each equivalence class in the subhierarchy Figure 8 with a parameter using the naming scheme shown under the classes will yield the same structure as presented in Figure 5, which is the expected result.

$$\underbrace{\{X, X^t\}}_{L_1} \quad \underbrace{\{Y, X'\}}_{L_i} \quad \underbrace{\{Y', Y^t, Y^L\}}_{L_d} \quad \{X^L\} \tag{8}$$

Parameter Simplification

Now that we have described a way to extract a set of possible parameters for a given HTN, we need to identify how parameters are passed from to its methods and from a method to its subtasks. This is done in a simplification step where we unify parameters from distinct sources.

We wish for the set of parameters to be general enough to be able to cover all the examples (and hopefully generalize well to new instances) while still restricting the decomposition possibilities to limit the search effort required of the solver. We propose to achieve this simplification through two main procedures:

1. Parameter unification, where parameters are unified with one another according to the examples.
2. Parameter removal, where parameters that will not help the solver will be dropped.

Parameter Unification We want to unify as many parameters as possible from the examples given as input. This is motivated by the fact that it will *(i)* reduce the number of parameters to instantiate in the model and *(ii)* constrain the parameters of the subtask of a given method, allowing them to refer to the same constant for the whole method without requiring the planner to infer that this is the best parameterization.

To achieve this unification, we frame the problem as MAX-SMT with the theory of equality and uninterpreted functions. We define $\text{args}(x)$ as the function that returns the ordered set of arguments of x, where x may be a method, a task, a subhierarchy or a set of subhierarchies and $\text{arg}_i(x)$ the function that returns the ith argument of x. We also define $\text{gnd}_d(P)$ as the function that returns the set of possible ground instantiation of a parameter P in the demonstration d, $d \in \mathcal{D}$.

As information may be propagated both upwards (from the primitive tasks in the demonstrations) and downwards (from a high level abstract task down to lower level ones), both cases need to be considered. While it is feasible to express this as a single set of constraint, the formulation is more complex and the practical performanc is worse, which is why we decided to separate these two propagations into two distinct steps. Even though the resulting parameterization may not be optimal, preliminary results show that the extracted parameters are consistent with the principles defined earlier, while the set of constraints remains easy to specify.

The constraints used to express the upwards propagation of information are both extracted from the examples as well as structural, and are defined in the following equations:

$$\begin{aligned} &\forall h_{sub} \in \mathcal{H}_{subs}, \forall P_1, P_2 \in \text{args}(h_{sub}), P_1 \neq P_2 \\ &\quad \text{For any ground instantiation of } h_{sub} \text{ in } d \in \mathcal{D} \text{ and} \\ &\quad \text{associated grounding } p_1 \text{ of } P_1 \text{ (resp. } p_2 \text{ of } P_2\text{):} \\ &\quad \begin{cases} \text{SOFT}(P_1 = P_2) & \text{if } p_1 = p_2 \\ \text{HARD}(P_1 \neq P_2) & \text{if } p_1 \neq p_2 \end{cases} \end{aligned} \tag{9a}$$

$$\begin{aligned} &\forall h_{sub} \in \mathcal{H}_{subs}, \forall P \in \text{args}(h_{sub}) \\ &\quad \nexists d \in \mathcal{D}, \text{gnd}_d(P) \neq \emptyset \\ &\qquad \Rightarrow \text{HARD}(\forall P' \in \text{args}(\mathcal{H}_{subs}) \setminus P, P \neq P') \end{aligned} \tag{9b}$$

$$\forall h_{sub} \in \mathcal{H}_{subs},$$
$$T_s = \{t_s \in \textsc{Subtasks}(\mathcal{H}_{subs}), \mathrm{sym}(t_s) = \mathrm{sym}(t_h)\}$$
$$\textsc{Hard}\left(\begin{array}{c}\forall (i,j) \in |\arg(t_h)|^2, \arg_i(t_h) = \arg_j(t_h) \\ \Rightarrow \forall t_s \in T_s, \arg_i(t_s) = \arg_j(t_s)\end{array}\right) \tag{9c}$$

Equation 9a simply translates the fact that two parameters can be unified if there is a positive example (soft constraint) but must never be unified if there is a negative example (hard constraint). Equation 9b is similar, enforcing that if a parameter has never been encountered in any example, then we have no reason to unify it with any other one. Finally, equation 9c enforces that if two parameters of an abstract task of symbol t have been unified in its "reference" definition (as the root t_h of the corresponding subhierarchy), then every instantiation of t as a subtask must unify these parameters as well to remain consistent.

After solving the constraint satisfaction problem, equivalence classes can be obtained, which are treated similarly to what has been shown for the recursive task preprocessing: for each equivalence class, only one single parameter is kept for abstract tasks and method parameters.

To address the downward propagation of the information, we try and unify parameters that are bound to higher level tasks' parameters, and therefore simply use a constraint similar to the one in equation 9a, as defined below:

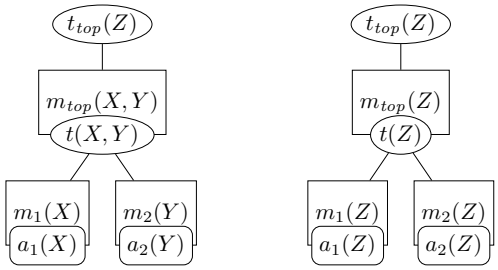
$$\forall h_{sub} \in \mathcal{H}_{subs}, \forall t_s \in \textsc{Subtask}(h_{sub}),$$
$$\forall (P_s, P_h) = \arg(t_s) \times \arg(t_h)$$
For any ground instantiation of h_{sub} in $d \in \mathcal{D}$ and associated grounding p_s of P_s (resp. p_h of P_h):
$$\begin{cases}\textsc{Soft}(P_s = P_h) & \text{if } p_s = p_h \\ \textsc{Hard}(P_s \neq P_h) & \text{if } p_s \neq p_h\end{cases} \tag{10}$$

Analogously to the previous part, we then extract equivalence classes from the solver result, and unify together arguments according to these classes.

Figures 10 and 11 show an example of this process. Figure 10a shows an example of subhierarchy where we may unify parameters as shown in Figure 10b, assuming we have the decomposition trees as shown in Figure 11. The lower level of methods and subtasks in Figure 10 is only included for clarity when presenting the decomposition trees, the subhierarchies considered in equation 10 actually have the same structure as presented earlier in the paper.

Parameter Removal Once the unification process has taken place, the HTN model may still contain abstract tasks with a large number of parameters, leading to methods with many parameters which will be difficult to instantiate for the solver. Therefore, we propose to remove the parameters that will hinder a solver's performance rather than improve it. We propose to define "useful" parameters as:

1. Parameters enabling early decision in the hierarchy, propagating information downwards.
2. Parameters enabling parameter unification across sibling subtasks.



(a) Before unification. (b) After unification.

Figure 10: Example of extended subhierachy used for downward information propagation.

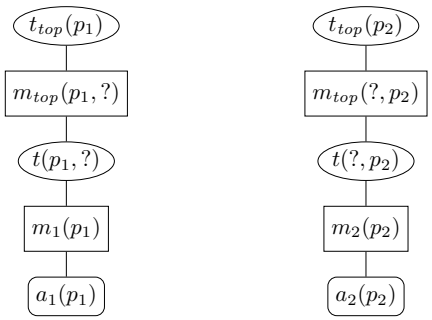


Figure 11: Example decomposition trees for downward propagation.

While we chose to focus on these two criteria first to define useful parameters, other possibilities are discussed at the end of this paper.

To determine which arguments to remove, we define two functions: $\textsc{Parents}(p)$ and $\textsc{Has Siblings}(p)$. $\textsc{Parents}(p)$ returns the set of parameters that are used as parents of a given parameter p in method's subtask, allowing to implement rule 1. $\textsc{Has Siblings}(p)$ returns $\textsc{True}$ or $\textsc{False}$ depending on whether p is used to unify parameters of multiple sibling tasks, allowing to implement rule 2.

Algorithm 4: Parameter Removal

repeat
  $\mathcal{P}_{drop} \leftarrow \emptyset$
  for all $p \in \arg(\mathcal{H}_{subs})$ **do**
    if $\textsc{Parents}(p) = \emptyset \wedge \neg\textsc{Has Siblings}(p)$ **then**
      $\mathcal{P}_{drop} \leftarrow \mathcal{P}_{drop} \cup \{p\}$
    end if
  end for
  $\arg(\mathcal{H}_{subs}) \leftarrow \arg(\mathcal{H}_{subs}) \setminus \mathcal{P}_{drop}$
until $\mathcal{P}_{drop} = \emptyset$

We then apply the procedure described in Algorithm 4, where parameters are removed from the tasks and methods until a fixed point is reached.

Evaluation

We here present preliminary results indicating that our approach is able to extract reasonable parameters. The constraint solver used for evaluation was Z3 (de Moura and Bjørner 2008).

Table 1 shows some results with regard to parameter extraction on domains from the 2020 IPC competition[1]. All the demonstrations were generated using the LILOTANE (Schreiber 2021) planner for instances in the IPC repository using a short 10 seconds timeout. The "IPC" column shows the number of parameters (in method and tasks) for a reference domain from the competition, while the "Superset" and "Simplified" columns respectively show the number of arguments after the first generation phase and after the simplification phase.

We can observe that the number of extracted arguments is close to the one from the reference model and that the simplification procedure is required for extracting models of reasonable size. Results marked with the † symbol contains recursive tasks, and do not make use of the procedure detailed earlier, as it was not fully integrated into the parameter extractor at the time of writing this paper. However, even though the extracted parameters are not as relevant in this version, the true number of extracted parameters should actually be lower, as the specific procedure for recursions should allow more unifications of arguments in a given recursive method.

Qualitatively, the extracted parameters are in line with what we could expect a human user to design. The main exception is with regard to recursive methods, as expected. We conjecture that some supplementary arguments, compared to hand-designed domains, are caused by a lack of unifications across methods in some cases. This issue is discussed in a later section.

	Parameter Count		
Domain	IPC	Superset	Simplified
CHILDSNACK	12	31	14
TOWERS†	34	158	54
HIKING†	62	721	76
SATELLITE†	29	96	22
ROVER†	58	252	72

Table 1: Argument extraction results on IPC domains.

Finally, a basic version of the parameter extractor was used in a full learner, extending the work from Hérail and Bit-Monnot (2022). This learner has shown similarly performing models to hand-designed ones, for simple domains. Figure 12 shows planning performance on a variant of the TRANSPORT domain for learned models with this system, compared to the reference IPC model. The different learned models only varied in their structure, not in the parameter extraction procedure. These model learning parameters are not detailed here as they are not relevant for this work. While this does not evaluate the performance of the argument extraction alone, it shows that it can be successfully integrated in a more complete system.

[1]Available at https://github.com/panda-planner-dev/ipc2020-domains

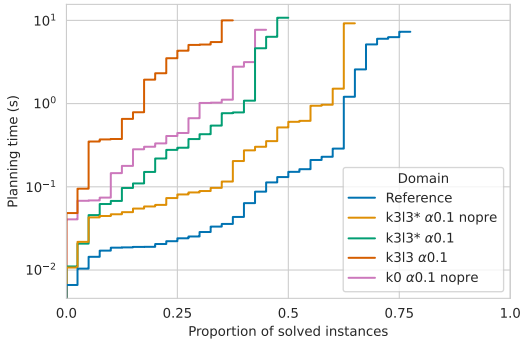


Figure 12: Planning time for models learned the proposed approach for parameter learning using the LILOTANE planner.

Discussion

Parameter Generation Assumptions In the definition of the assumptions surrounding the parameter superset generation, we considered that parameters of the preconditions of a method are a subset of the parameters of the method's subtasks. While this assumption appears valid in most of the IPC domains, if some domains require relaxing it, techniques such as the use of deictic references (Pasula, Zettlemoyer, and Kaelbling 2007) could be used to this end.

Parameter Removal While the presented approach shows acceptable performance, both in terms of extraction speed and model quality, the rules used to remove parameters could easily be extended by taking into account pre- and post-states.

Indeed, some IPC domains, such as the HIKING domain, have methods that have constraints that rely on parameters that are not passed down from a parent task nor are used to unify sibling methods. Therefore, it may be interesting to try and determine potential preconditions before the argument removal step and keep arguments that participate in the specification of method preconditions, as it should lead to a more efficient planning process from the solver. If effects of abstract tasks (Olz, Biundo, and Bercher 2021) are to be extracted, a similar argument can be made to keep parameters potentially participating in the definition of these effects.

Unification of Parameters Across Methods Additionally, while we are able to propagate unification information upwards from the demonstration, downwards from top level tasks and sidewards across subtasks of a method, we cannot unify multiple sibling methods' parameters, without relying on higher level tasks, as in the example presented Figure 10.

While we do not propose a solution in the case of demonstrations taken in isolation, in the context of demonstrations given with the explicit goal of teaching an agent, it does not

seem unreasonable to assume that multiple demonstrations can be given in the same ground context. The learner could then use this common context to infer additional parameter unifications. Furthermore, if the demonstrations are given as a form of curriculum, then it is reasonable to assume multiple high level tasks with user-defined arguments will be given, building up the hierarchy incrementally starting with lower-level abstract tasks, alleviating the impact of this issue in practice.

Fixed Parameter Tasks Finally, the presented constraints consider that the tasks with fixed arguments only appear as the root of a decomposition tree for extracting unification constraints. However, if they were used as subtasks, the mapping between their parameters and the primitive action parameters would be undefined, as can be seen in the subhierarchy in Figure 3: here, if t_{top} was used as part of an arbitrary subhierarchy, this undefined mapping would break the propagation of the arguments in the decomposition tree.

A possible solution would be to consider giving the expected effects of the demonstrated tasks along with the demonstrations. This would allow extracting possible mappings for the task's parameters, which could then be integrated into the system of constraints used during parameter simplification.

Conclusion

We presented a procedure to extract the parameters for a given HTN structure. This procedure may be used to allow HTN learning algorithms to focus on the model structure to extract relevant parameters, but it could equally be used in a system allowing a user to sketch the hierarchical structure of a domain and simply give some demonstrations of the expected behavior, leaving the system to complete the model. It may also be integrated into a larger system to automatically extract preconditions or even effects for task and methods in the learned HTN models, leading to systems that may compete with state-of-the-art learners while reducing the burden of annotation placed on the user.

While the evaluation is still preliminary, the results show that the approach extracts reasonable parameters for HTNs from the IPC. However, a more thorough evaluation would be necessary to identify the most pressing shortcomings and promising improvements.

References

Chen, K.; Srikanth, N. S.; Kent, D.; Ravichandar, H.; and Chernova, S. 2021. Learning Hierarchical Task Networks with Preferences from Unannotated Demonstrations. In *Proceedings of the 2020 Conference on Robot Learning*, 1572–1581. PMLR.

de Moura, L.; and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In Ramakrishnan, C. R.; and Rehof, J., eds., *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, 337–340. Berlin, Heidelberg: Springer. ISBN 978-3-540-78800-3.

Dong, R.; Huang, Z.; Lam, I. I.; Chen, Y.; and Wang, X. 2022. WebRobot: Web Robotic Process Automation Using Interactive Programming-by-Demonstration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 152–167. San Diego CA USA: ACM. ISBN 978-1-4503-9265-5.

Hérail, P.; and Bit-Monnot, A. 2022. Learning Operational Models from Demonstrations: Parameterization and Model Quality Evaluation. In *ICAPS Hierarchical Planning Workshop (HPlan)*. Singapore (virtual), Singapore.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, 950–956. Chicago, Illinois: AAAI Press. ISBN 978-1-57735-368-3.

Höller, D.; Wichlacz, J.; Bercher, P.; and Behnke, G. 2021. Compiling HTN Plan Verification Problems into HTN Planning Problems. In *Proceedings of the 4th ICAPS Workshop on Hierarchical Planning (HPlan 2021)*, 8–15.

Li, N.; Cushing, W.; Kambhampati, S.; and Yoon, S. 2014. Learning Probabilistic Hierarchical Task Networks as Probabilistic Context-Free Grammars to Capture User Preferences. *ACM Transactions on Intelligent Systems and Technology*, 5(2): 32.

Manna, Z.; and Waldinger, R. J. 1971. Toward Automatic Program Synthesis. *Communications of the ACM*, 14(3): 151–165.

Nieuwenhuis, R.; and Oliveras, A. 2006. On SAT Modulo Theories and Optimization Problems. In Biere, A.; and Gomes, C. P., eds., *Theory and Applications of Satisfiability Testing - SAT 2006*, Lecture Notes in Computer Science, 156–169. Berlin, Heidelberg: Springer. ISBN 978-3-540-37207-3.

Olz, C.; Biundo, S.; and Bercher, P. 2021. Revealing Hidden Preconditions and Effects of Compound HTN Planning Tasks – A Complexity Analysis. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(13): 11903–11912.

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2007. Learning Symbolic Models of Stochastic Domains. *Journal of Artificial Intelligence Research*, 29: 309–352.

Raza, M.; and Gulwani, S. 2018. Disjunctive Program Synthesis: A Robust Approach to Programming by Example. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

Schreiber, D. 2021. Lilotane: A Lifted SAT-based Approach to Hierarchical Planning. *Journal of Artificial Intelligence Research*, 70: 1117–1181.

Segura-Muros, J. A.; Pérez, R.; and Fernández-Olivares, J. 2017. Learning HTN Domains Using Process Mining and Data Mining Techniques. In *ICAPS Workshop on Generalized Planning*. Pittsburgh, United States.

Zhuo, H. H.; Muñoz-Avila, H.; and Yang, Q. 2014. Learning Hierarchical Task Network Domains from Partially Observed Plan Traces. *Artificial Intelligence*, 212: 134–157.

Implicit Dependency Detection for HTN Plan Repair

Paul Zaidins[1,3], Mark Roberts[3], Dana Nau[1,2]

[1]Dept. of Computer Science and [2]Institute for Systems Research, University of Maryland, College Park, MD, USA
[3]Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC, USA
[1]{pzaidins, nau}@umd.edu [3]{paul.zaidins,mark.roberts}@nrl.navy.mil

Abstract

Two recent approaches to HTN replanning, IPyHOP and SHOPFIXER, replan by adapting the previously planned solution when an action fails. IPyHOP replans the entire solution tree after the failure, while SHOPFIXER uses pre-calculated dependency graphs to replace portions of the tree; neither uses forward simulation of the plan to predict where future failures might occur.

This paper describes IPyHOPPER, which improves IPyHOP by retaining more of the information provided by the hierarchy and using forward simulation to repair minimal subtrees that contain future failures. Our experimental comparisons show that in domains where errors are not rare, IPyHOPPER is both faster and uses fewer iterations to repair than IPyHOP's repair mechanism. IPyHOPPER's repair speedups are similar to those of SHOPFIXER when given a probabilistic error model with nontrivial error rates.

1 Introduction

Given some level of domain expertise, a Hierarchical Task Network (HTN) can be leveraged to solve complex problems quickly. Hierarchy is perhaps the most powerful feature of HTN planners. For planning in static, known environments using this hierarchy is straightforward. However, because the world is often dynamic and full of uncertainty, acting must be properly coupled with *online* planning to operate well in such conditions. So the speed and efficiency of correcting errors in plans is important.

When a failure is encountered the most obvious solution is to replan using the current state as the new initial state. Even though it may be incorrect after a failure, the original plan provides useful information regarding the hierarchy and implicit action restrictions. Reusing portions of the original plan may also promote plan stability (Fox et al. 2006).

Two recent approaches to plan repair involve updating original solution tree: SHOPFIXER (Goldman, Kuter, and Freedman 2020) and the Lazy-Refineahead repair algorithm in the IPyHOP paper (Bansod et al. 2022). Both approaches use portions of the original plan when facing disruptions during execution, but differ in the following respects.

- Lazy-Refineahead uses the structure of the original plan up to the point of failure, but discards the original plan hierarchy past the failed action. In contrast, SHOPFIXER pre-calculates dependency graphs, so that when action ends in failure, the plan repair process can skip subtrees that are not dependent on the action that failed. This allows for minimal fixes, increasing plan stability and reducing repair time.
- SHOPFIXER uses actions and methods in the SHOP 3 format (Lisp code with similar structure to PDDL), so programmers must understand how to implement methods in that format. IPyHOP's methods and operators are Python functions, making them more accessible to those unfamiliar with automated planning formalisms. This makes the explicit dependency graph calculation, as SHOPFIXER does, impossible due to the lack of well-defined preconditions or effects.

This paper describes IPyHOPPER, which extends IPyHOP[1] to use plan repair techniques inspired by those in SHOPFIXER. IPyHOPPER improves IPyHOP's repair efficiency by retaining more of the information provided by the hierarchy and using forward simulation. Rather than cutting the solution tree at the failed node, it selectively prunes future solution branches using forward simulation to repair minimal subtrees containing future failures. This avoids future failures that may arise from the plan alterations from repairing the minimal subtree of the immediate failed action and alterations brought from prior preemptive repairs. Our contributions include:

1. A new algorithm, IPyHOPPER, that performs a mix of in-place repair with forward simulation;
2. Revised benchmarks to support evaluation of IPyHOPPER against IPyHOP and SHOPFIXER; and
3. Experiments showing that IPyHOPPER reduces planner iterations and execution time for the 5 tested domains while keeping similar, or sometimes better, plan costs.

The rest of the paper gives a brief overview of related work, presents our IPyHOPPER algorithm for minimal repair with IPyHOP, and describes a comparison of the performance of IPyHOPPER, Lazy-Refineahead, and SHOPFIXER.

[1]IPyHOP and IPyHOPPER can plan for both tasks and goals, but for brevity we will refer to both of them as HTN planners. Section 2.5 provides further justification for this terminology.

2 Related Work

Our work is based on the implementation of IPyHOP and we compare to SHOPFIXER, so we will discuss each of these before pointing out other related work.

2.1 IPyHOP

IPyHOP (Bansod et al. 2022), is an iterative, domain-independent, totally-ordered, Goal Task Network planner written in the Python programming language. Goals and tasks are ordered tuples while methods and actions are arbitrary Python code blocks. Input goals and tasks are repeatedly decomposed until only actions remain. This results in a solution tree from which the plan can be read as the preorder traversal of actions (i.e., leaves) of the tree.

IPyHOP's repair algorithm, Lazy-Refineahead, works by finding the failed action in the solution tree, removing all nodes to the right of the failed action in the preorder traversal, and then marking nodes that lost children. From there the plan is repaired by iterating through the tree and redoing decomposition at those marked nodes with the current state used for the preorder traversal's leftmost new nodes. It should also be noted that method preconditions are only checked during planning and during plan repair for methods touched during repair.

2.2 SHOPFIXER

SHOPFIXER (Goldman, Kuter, and Freedman 2020) is a plan repair system for the SHOP 3 (Goldman and Kuter 2019) HTN planner. Unlike IPyHOP, SHOP 3 uses highly structured Lisp code for methods and actions. Variables are instantiated primarily through unification. This allows for the computation of causal links in the solution tree. SHOPFIXER uses these causal links to isolate plan repair to only the sections of the solution tree that are relevant. This served as part of the inspiration for IPyHOPPER, the principal difference being that our methodology does not impose any structural restrictions on the decomposition methods and actions, as dependencies are found implicitly through simulation rather than explicitly computed. Thus, IPyHOPPER can still speed up plan repair even when a dependency graph is not known.

2.3 Plan Repair by Domain Modification

Given a planning domain D, task t and initial state s_0, suppose an HTN planner returns a solution plan $\pi = (a_1, \ldots, a_i, \ldots, a_n)$. While executing π, suppose an execution error occurs at a_i, producing a state s_i' rather than the predicted state s_i. Höller et al. (2020) define a modified planning domain D' in which (i) the predicted outcome of a_i is s_i' and (ii) given D', t, and s_0, the HTN planner returns a solution plan π' such that the first i actions are the same as in π. Thus if no further errors occur, t can be accomplished by executing the part of π' that starts at the action after a_i.

Unlike IPyHOPPER, π' preserves none of the unexecuted part of π. We think the actions in π' after a_i are the same as Lazy-Refineahead would have produced. Technically this can be viewed as a hybrid approach between plan repair and replanning.

2.4 Other Approaches

There has been work speeding up replanning by reusing plan solution trees (Soemers and Winands 2016). This differs from standard plan repair in that existing plans are modified for new tasks as opposed to altering an existing plan for the same task given an interruption. RepairSHOP (Warfield et al. 2007) uses a directed, dependency graph know as a goal graph to track alternative decisions (decompositions and instantiations). When plan repair is needed, the planner is reset to the planning state of the first applicable alternative found. HOTRiDE (Ayan et al. 2007) takes a similar approach with its task-dependency graph.

2.5 Evolution of HTN Terminology

Some of the best-known early formulations of HTN planning included both goals and tasks (Currie and Tate 1991; Kambhampati and Hendler 1992; Erol, Hendler, and Nau 1994). However, SHOP and its successors used a simplified HTN formulation that omitted goals (Nau et al. 1999, 2001; Goldman and Kuter 2019). Their popularity led researchers to lose track of goals as a part of HTN planning, and HGNs were subsequently conceived as separate from HTNs (Shivashankar et al. 2012, 2013). In this paper, we return to the earlier concept that HTN planning includes both goals and tasks.

3 Repairing Minimally with IPyHOPPER

Before we formally describe IPyHOPPER, we will contrast it with an example from the IPyHOP paper. Figure 1(a) copies Figure 1 by Bansod et al. (2022). It represents a notional hierarchical plan for tasks $t1$, $t2$, and $t3$. Hexagonal nodes indicate method instances m_i for a task t_j, and rectangular nodes indicate an operator instance o_i. The resulting plan, $\pi = \langle o1, o2, ..., o11, o12 \rangle$ is produced using a Depth First Search (DFS) tree preorder traversal. Moreover, $o7$ produces effects on which $o11$ relies, shown as a red dashed line; this will be an important detail in the following comparison.

While executing π, $o7$ nondeterministically fails. The Lazy-Refineahead algorithm discards the plan structure for the parent of the failed node as well as the nodes to the right of the failed node in preorder traversal, which includes $m1\_t4$, $m1\_t5$, and $m1\_t3$. This results in nine nodes removed from the tree. But it might be the case that only a few of these need to be changed to repair the plan.

Instead, IPyHOPPER preserves as much of the tree as possible to minimize computation and maximize stability (Fox et al. 2006). It does this by combining forward simulation (i.e., action application to each state) with localized repair. As simulation progresses forward, each action is checked for applicability. If the simulation succeeds for all actions in the remaining plan, then the repair was localized to the failed node. When the simulation fails for an *existing* action in a plan, that is treated as a potential future failure, resulting in further repair.

To make this concrete, consider Figure 1(b), where a dashed green line indicates starting point of simulation with, $t4$, the parent of the failed action having been unexpanded. IPyHOPPER removes only the children of $t4$ to find a new

decomposition the parent of the failed action using the current state, resulting in two new actions.

IPyHOPPER simulates forward from $o13$ to check for any future problems from the repair. Forward simulation reveals that actions $o8$, $o9$, and $o10$ will succeed so the tree supporting these actions remain in the plan. The result is shown in Figure 1(c) between the green and orange dashed lines. At this point, suppose $o11$ fails, perhaps because its precondition from $o7$ was not met by the new actions $o13$ and $o14$. Figure 1(d) shows that IPyHOPPER makes another repair to rightmost instance of $t4$, resulting in a new action $o15$. From the original plan, actions $o8$, $o9$, and $o10$ remained. We include Figure 2 to demonstrate the difference in the repair process with Lazy-Refineahead. Note how the parent and all pre-order succeeding nodes are unexpanded regardless whether the actions could still have been performed.

To summarize, IPyHOPPER repairs a plan by removing a failed node's immediate subtree then repairing while simulating possible future failures. Using the new observed state, it fixes the tree at the point of failure, traversing upward as needed while simulating forward. If the simulation yields errors, it fixes the next point of failure and continues simulating forward. This repeats until it reaches an unrecoverable state or the end of the simulated plan without a failure. A repair-simulate cycle might introduce further failures. This occurs when failure points have a common ancestor and the rightmost failure point has no applicable repair given the previous failure point repair. In such a case, it backtracks to the previous failure point. Backtracking to the root node of our tree indicates no plan is possible. Otherwise, it returns the repaired plan.

3.1 The Repair Algorithm

Algorithm 1 formalizes the above example. We assume here that the root of the full tree exists as the parent of all tasks provided to the planner as in IPyHOP.

Lines 2 and 3 place the parent of the failed action and current state into the stack and enters the repair-simulate loop. This loop continues until either simulation results in a successful plan or the algorithm reaches an unrecoverable state.

Lines 5 and 6 read the top node and state from the stack and replace the top node with its parent. This results in either moving up the solution tree when repair fails for the current subtree or moving to the previous repair when arriving at the root.

Line 7 removes the current decomposition of f to allow for a new decomposition. This begins the minimal repair, from which Lines 8 to 12 will check to see if any decomposition methods are relevant and if so will attempt to repair the subtree of w rooted at p.

The expansion is performed using a modified IPYHOP planning mechanism. The most important changes are preventing planning to progress to the top node of a subtree (thus allowing targeted subtree repair) and altering IPyHOP methods to find all potential variable bindings rather than a single binding (thus allowing functionality similar to unification). There are two cases to consider: (1) If there are relevant methods, but no applicable expansions, return to the top loop at Line 4. (2) If no relevant actions exist, check if p is a common ancestor of any other node in the stack and, if so, remove p and its associated state from the stack as the next loop will clobber previous repairs. Either case results in a return to the top of the loop.

A successful repair of the subtree of w results in advancing the simulation at Lines 17 to 25. Simulation proceeds by reading the suffix of π starting from w and following it to completion from our current state. At this point, there are two cases to consider. (1) If π has another failure, s' becomes the state at failure, the parent of the action that failed is placed on the stack, and the loop returns to the top. (2) If the simulation completes the rest of π without failure we terminate the loop and return π as the repaired plan to begin executing. We note here that the stack thus functions as our backtracking mechanism as the stack consists of the parents of our repaired nodes. We exhaust all possible repairs for our current node before returning to the immediate prior repair point to look for a solution If the stack should become empty (i.e., the algorithm arrive at the root), then tree must have been stripped down to the root and π will be empty. When this occurs, an empty plan indicates failure of the algorithm to produce a viable repair.

Algorithm 1: IPyHOPPER plan repair algorithm.

```
1  Def IPyHOPPER( state: s, decomposition tree: w, failed
     action node: f ):
2      p ← parent of f in w;
3      stack ← [(p, s)];
4      while stack not empty do
5          f, s ← pop(stack);
6          p ← parent of f in w;
7          unexpand f;
8          if f has possible decomposition then
9              subTree ← subtree of w with root f;
10             attempt expansion of subTree from s;
11             if subTree cannot be fully expanded then
12                 continue;
13         else
14             if if the parent p is an ancestor of a previous
                 node then
15                 pop(stack);
16             continue;
17         π ← plan from w;
18         simulate( s, π);
19         if simulation failed then
20             s' ← input state for failed action;
21             p_a ← parent of failed action node;
22             push( stack, (p_a, s'));
23             continue;
24         else
25             break;
26     π ← plan from w;
27     return π;
```

4 Experiments

We incorporated Algorithm 1 into Run-Lazy-Refineahead with no other changes, resulting in IPyHOPPER. We compared IPyHOPPER to IPyHOP as well as to SHOPFIXER.

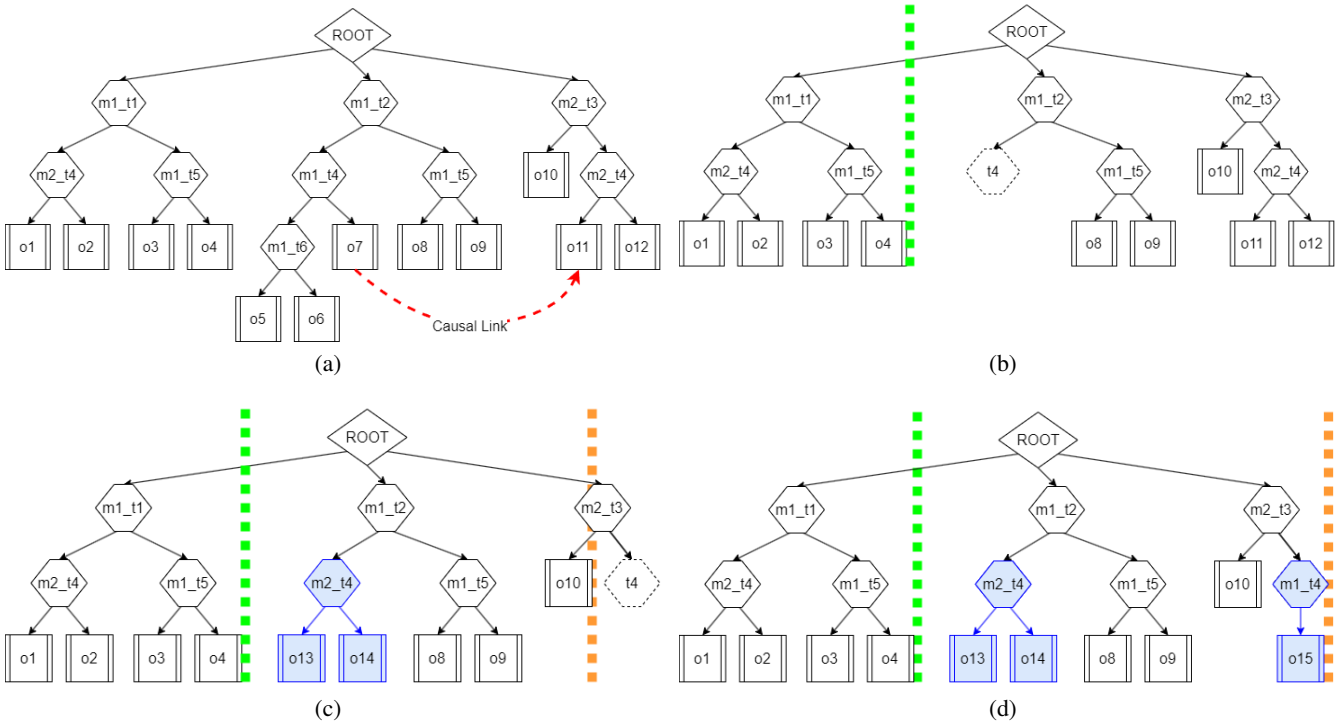


Figure 1: An example IPyHOPPER repair using Figure 1 by Bansod et al. (2022). See prose for a detailed description.

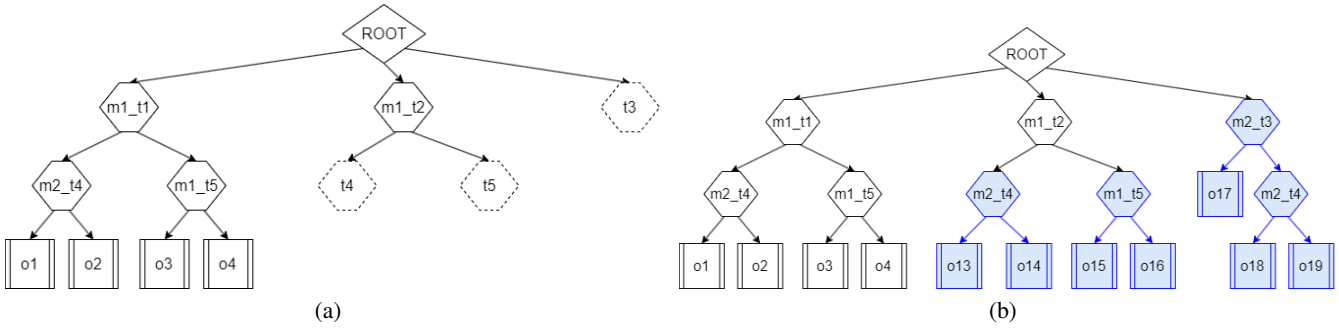


Figure 2: With Lazy-Refineahead, 1(b), 1(c), and 1(d) would have been the above.

4.1 IPyHOP Comparison

To evaluate how well IPyHOPPER improves the performance of IPyHOP, we repeat the study on two domains from the IPyHOP paper (Bansod et al. 2022).

The robobsub domain comes from the RoboSub 2019 competition. An autonomous submersible is expected to complete a course consisting of several tasks. Not all tasks must be completed for an attempt at the course to be considered complete, but these optional tasks yield additional points. For such tasks non-empty decompositions are attempted first by the internal ordering of the methods and if no non-empty decomposition is applicable an empty decomposition is returned rather than concluding no plan is possible.

The rescue domain consists of several unmanned air and ground vehicles attempting to perform search and rescue operations following some disaster. One important difference between these domains is that the rescue domain has states from which recovery is impossible.

The experimental setup is identical except for more runs; the starting seeds increased from 1,000 to 10,000 with number of trials per seed increased from 11 to 100. We evaluate using two metrics from Bansod et al. (2022). *Total decompositions* assesses the total number of nodes expanded, thus tasks and actions are both counted, while *total action cost* is the sum of the costs of all actions attempted, including failed and successful actions.

For the Rescue domain, we found that of our 10,000 seeds, 412 were unsolvable. Of the remaining 9,588 configurations, 82 of them had at least one trial end in an unsolvable state when using the Lazy-Refineahead algorithm. IPyHOPPER had no such partial seed failures.

Rescue IPyHOPPER produces plans with lower total action cost using fewer node expansions. Figure 3(a) shows that IPyHOPPER produces a lower total action cost for most problems, resulting in an averaged reduction of 2.28 ± 0.02 with 95% confidence. This is most likely a domain-specific effect, as the algorithm does not impose any cost-related conditions. Most plan costs occur in discrete, relatively narrow bands while IPyHOP produces a much less coherent pattern. This is likely a consequence of the stability that the new algorithm imposes through minimal plan changes.

Figure 3(b) shows that IPyHOPPER expands significantly fewer nodes than Lazy-Refineahead for all configurations (28.77 ± 0.03 with 95% confidence). This is in line with expectations, as the failures in this domain are generally fixable by repairing only the immediate parent of the failed node, while Lazy-Refineahead must replan all of the unexecuted plan.

Robosub IPyHOPPER produces plans with higher total action cost using fewer node expansions. Figure 3(c) shows that IPyHOPPER produces plans with a significantly higher total action cost for all problems (33.80 ± 0.02 with 95% confidence). This is unexpected, but perhaps a consequence of the unique feature of most actions being skippable in this domain. Optional objectives are rewarded with additional points, but are not required. The method of last resort to decompose such tasks is often simply to return nothing.

Table 1: Mean percentile change in paired sample difference for CPU time and mean iteration count by domain. A negative value indicates that IPyHOPPER is outperforming Lazy-Refineahead.

Domain	Change in mean CPU time (%)	Change in mean iteration count (%)
Openstacks	-34.5	-16.1
Rovers	-40.7	-49.3
Satellites	-34.0	-49.3

Figure 3(d) shows that IPyHOPPER not only expands significantly fewer nodes ($59.45 \pm 0.07\%$ with 95% confidence), but also reveals a nearly constant number of node expansions. This suggests IPyHOPPER is exceptionally stable for this domain.

4.2 SHOPFIXER Comparison

To compare IPyHOPPER to SHOPFIXER, we compare the difference between Lazy-Refineahead and IPyHOPPER on the SHOPFIXER domains explored by Goldman and Kuter (2020): openstacks, rovers, and satellites.

Our first task was to adapt the SHOP 3 methods as faithfully as possible. This is somewhat challenging because the exact format of the state is more "Pythonic" and IPyHOP does not use unification. We also needed to enable IPyHOP methods to return multiple different instantiations based on ground arguments. Otherwise, we replicated the spirit of the SHOP 3 methods, actions, and deviations.

We ran each problem 1000 times with a nominal error rate of 10%. To replicate the potential error distribution from the SHOPFIXER experiments while allowing for multiple errors in a single experiment, we scaled each action's likelihood of error linearly in proportion to the number of potential errors for that action and the current state.

Errors were introduced randomly and are uniformly randomly selected from all potential errors for that action and the current state. In general this meant that the action-state pair with the largest number of potential errors (as calculated by a running max) would have some error occur with probability equal to the nominal probability. Time spent calculating execution errors was not included in time measures for the planner. All domains here had only recoverable failures, so we did not need to consider the case of failed planning.

We recorded three metrics: action count, CPU time, and iteration count. *Action count* is the number of all actions attempted, thus both failed and successful actions are counted. *CPU time* is the elapsed process time from immediately prior to the initial call to the planner for a plan to immediately after the successful completion of the last action. *Iteration count* is the total number of iterations the planner runs for both in initial planning and every call to the plan repair algorithm. Iteration count includes iterations spent traversing through the solution tree without expanding nodes, and is slightly different than node expansions.

Openstacks IPyHOPPER has similar performance in terms of action counts but generally uses less CPU time and fewer

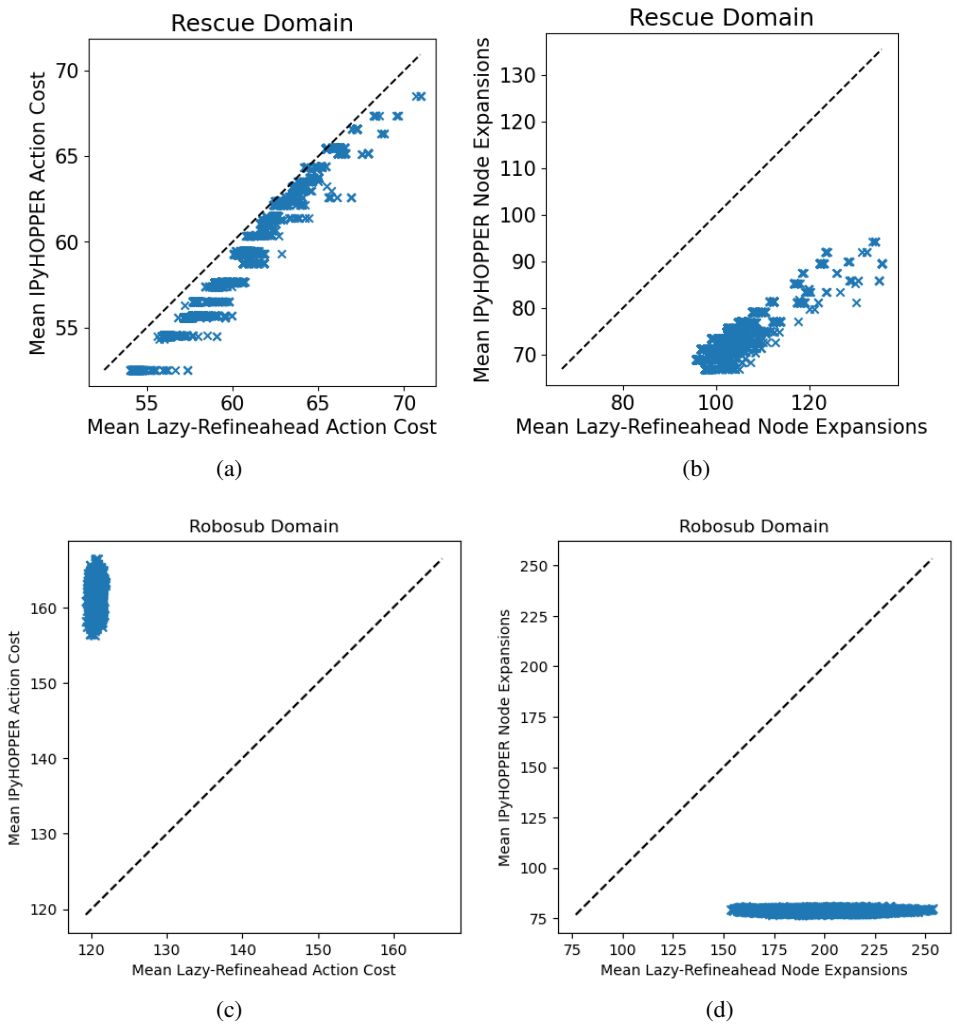


Figure 3: Scatter plots showing the mean action cost and node expansions for the Lazy-Refineahead and IPyHOPPER algorithms, in the Rescue and Robosub domains. For the metrics shown, points below the dashed line represent IPyHOPPER performing better than Lazy-Refineahead.

iterations. Figure 4 shows the spread for IPyHOPPER is no larger than SHOPFIXER and, for the larger problems, is significantly smaller in iteration count and CPU time. This comes from the increased stability of the new algorithm. There is a notable decrease in the medians of the iteration count and CPU time distributions when looking across all problems. However, this varies across problems. Action counts are essentially identical between the algorithms.

Rovers IPyHOPPER produces more varied plan costs but does so with lower CPU time and fewer iterations. Figure 5 shows a large improvement in iteration count and CPU time for most problems, especially for the largest problems. There is still a reduction in spread for iteration count and CPU time. There is definitely some distribution shift in the action count, but appears to highly dependent on problem.

Satellites IPyHOPPER produces results similar to Rovers with sometimes large improvements in action count using reduced iteration count or CPU time. Figure 6 shows less of a shift in the action count distributions. Both the rovers and satellites domain involve multiple heterogeneous agents with occasionally redundant capabilities fulfilling goals with essentially no ordering constraint. This sort of problem is amenable to the IPyHOPPER because repairs of a single subtask are localized in the solution tree.

Summary In Table 1 we summarize the performance gains of IPyHOPPER over Lazy-Refineahead in the SHOPFIXER domains. We find that for all three domains there is a significant improvement in both CPU time and iteration count.

5 Conclusions and Future Work

IPyHOPPER offers substantial improvement in iteration count and computation time across all 5 tested domains. Given the diverse nature of these domains, we speculate that

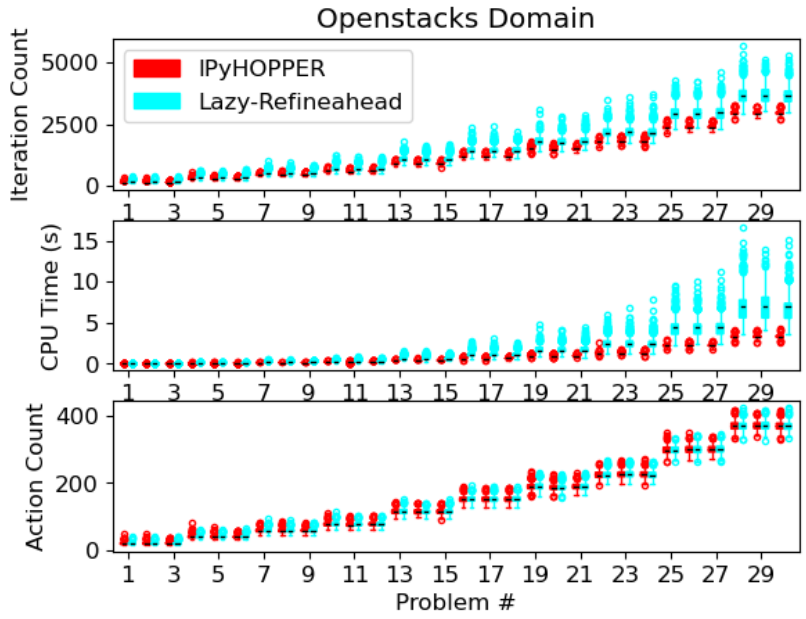


Figure 4: Key metric distributions for openstacks domain.

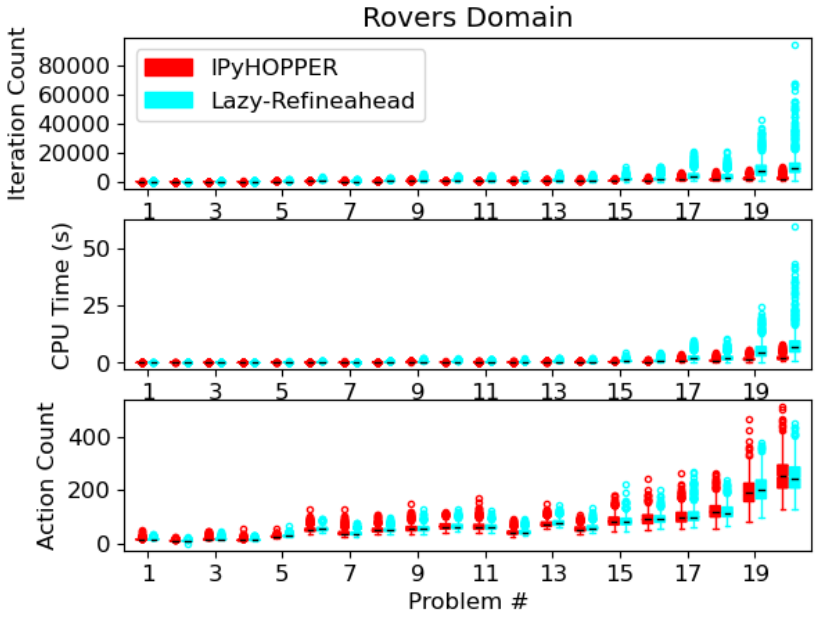


Figure 5: Key metric distributions for rovers domain.

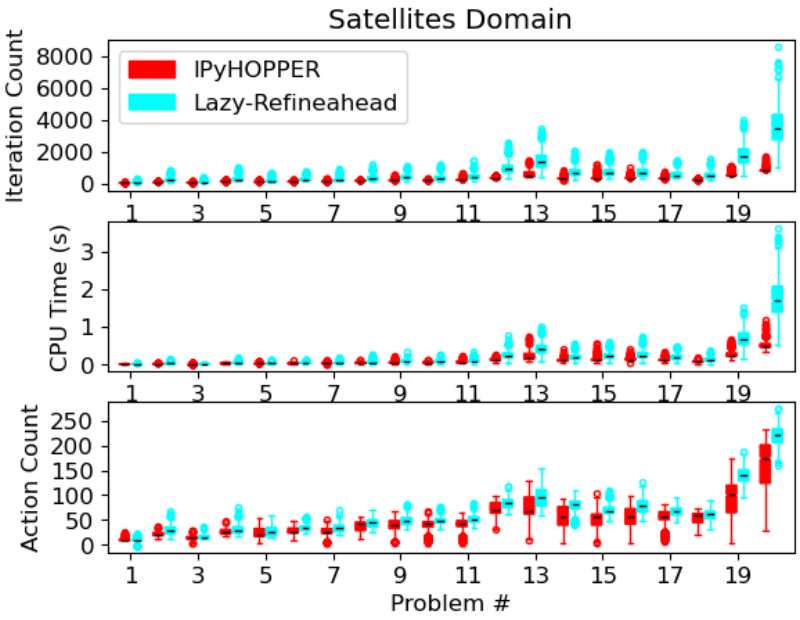


Figure 6: Key metric distributions for satellites domain.

this approach could yield similar benefits for many domains of interest. However, this approach cannot work as-is for all planning domains, because certain conditions regarding the definition of actions, methods, and the nature of errors must be met for plans to be guaranteed to be correct. These conditions are not currently well defined and this could be a subject of future work.

There seems to be a general increase in the relative benefit of the new algorithm with problem size. However, this needs to be investigated further because our suspicion is that the tested domains may be relatively sparse in terms of how subtasks are related. A better understanding of this quality of separability may yield valuable insights into planning. As the requirement for explicit preconditions and effects is removed in IPyHOPPER, it may offer some transferable lessons for refinement and hybrid plan repair.

Our experiments were in domains expressed in the well known PDDL. This does not make use of the full expressitivity possible with IPyHOPPER. Given IPyHOP should be capable of planning in domains not well expressed in PDDL form, we would like to measure potential gains in such domains.

Acknowledgements

This work has been supported for UMD in part by ONR grant N000142012257 and AFRL contract FA8750-23-C-0515. MR thanks ONR and NRL for funding this research. The views expressed are those of the authors and do not reflect the official policy or position of the funders.

References

Ayan, N.; Kuter, U.; Yaman, F.; and Goldman, R. 2007. HOTRiDE: Hierarchical ordered task replanning in dynamic environments.

Bansod, Y.; Patra, S.; Nau, D.; and Roberts, M. 2022. HTN Replanning from the Middle. *The International FLAIRS Conference Proceedings*, 35.

Currie, K.; and Tate, A. 1991. O-Plan: The Open Planning Architecture. *Artificial Intelligence*, 52(1): 49–86.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. In *AIPS*, 249–254.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: replanning versus plan repair. In *Proceedings of the Sixteenth International Conference on International Conference on Automated Planning and Scheduling*, 212–221. AAAI Press. ISBN 978-1-57735-270-9.

Goldman, R. P.; and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: the case of SHOP3. In Neuss, N., ed., *Proceedings of the 12th European Lisp Symposium (ELS 2019), Genova, Italy, April 1-2, 2019*, 73–80. ELSAA. ISBN 978-2-9557474-3-8.

Goldman, R. P.; Kuter, U.; and Freedman, R. G. 2020. Stable Plan Repair for State-Space HTN Planning. In *ICAPS Workshop on Hierarchical Planning (HPlan)*.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020. HTN Plan Repair via Model Transformation. In *KI 2020: Advances in Artificial Intelligence: 43rd German Conference on AI, Bamberg, Germany, September 21–25, 2020, Proceedings*, 88–101. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-030-58284-5.

Kambhampati, S.; and Hendler, J. A. 1992. A Validation-Structure-Based Theory of Plan Modification and Reuse. *Artificial Intelligence*, 55: 193–258.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple Hierarchical Ordered Planner. In *IJCAI*, 968–973.

Nau, D. S.; Muñoz-Avila, H.; Cao, Y.; Lotem, A.; and Mitchell, S. 2001. Total-Order Planning with Partially Ordered Subtasks. In *IJCAI*, 425–430. Seattle.

Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. 2013. The GoDeL Planning System: A More Perfect Union of Domain-Independent and Hierarchical Planning. In *IJCAI*, 2380–2386.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A Hierarchical Goal-Based Formalism and Algorithm for Single-Agent Planning. In *AAMAS*, 981–988.

Soemers, D. J. N. J.; and Winands, M. H. M. 2016. Hierarchical Task Network Plan Reuse for video games. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8.

Warfield, I.; Hogg, C.; Lee-Urban, S.; and Muñoz-Avila, H. 2007. Adaptation of Hierarchical Task Network Plans. In Wilson, D.; and Sutcliffe, G., eds., *Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference, May 7-9, 2007, Key West, Florida, USA*, 429–434. AAAI Press.

Integrating Deep Learning Techniques into Hierarchical Task Planning for Effect and Heuristic Predictions in 2D Domains

Michael Staud

Ulm University, Institute of Artifical Intelligence, D-89069 Ulm, Germany
michael.staud@uni-ulm.de

Abstract

In this paper, we present a novel approach that combines *Hierarchical Task Planning* (HTN) with deep learning techniques to address the challenges of scalability and efficiency in large-scale planning problems. Building upon the Hierarchical World State Planning (HWSP) algorithm, our method utilizes a multi-layered world state representation, which allows for planning at abstract levels without the need to consider lower-level details. We propose a deep learning method for predicting the effects of abstract tasks, which opens the door to enhancements in both planning performance and plan quality. Additionally, we employ the same approach to create a domain-dependent planning heuristic. Our contributions demonstrate the potential of integrating HTN planning with deep learning techniques, paving the way for future research in various application domains such as robotics, logistics, and urban planning. The proposed approach employs standard deep learning techniques, ensuring adaptability as the state of the art advances.

1 Introduction

Automatic planning is a fundamental problem in *artificial intelligence* with numerous practical applications, such as robotics and logistics. One approach to tackling complex problems is total-order *Hierarchical Task Planning* (HTN), which involves devising an abstract plan and gradually decomposing it into a concrete plan (Ghallab, Nau, and Traverso 2004, 238). However, existing planning methods often struggle with scalability and efficiency, particularly for large, complex problems. In this paper, we propose a novel approach that combines HTN planning with deep learning techniques to overcome these limitations.

Recent progress in *deep reinforcement learning* (Arulkumaran et al. 2017) has demonstrated the efficiency of this approach, with the hierarchical policies improving reinforcement agent performance and expediting the learning process (Vezhnevets et al. 2017; Huang 2020). By combining classical planning with reinforcement learning (Rivlin, Hazan, and Karpas 2020), it is possible to efficiently solve significantly larger planning instances. While not directly comparable, since these results involve policy generation, they offer insight into the potential benefits of deep learning for hierarchical planning problems. Additionally, reinforcement learning has been utilized in HTN planning to improve plan quality (Hogg, Kuter, and Munoz-Avila 2010). The Refinement Acting Engine (RAE), which also uses a hierarchical task structure like HTN planning but is an online algorithm, can also be successfully combined with a learning algorithm (Patra et al. 2020).

And in classical planning, deep learning has been used to learn planning heuristics. For instance, ASNets (Toyer et al. 2020) employ a clever approach to create domain-dependent heuristics by encoding action schemas in a neural network, allowing weight sharing across different problems. These can also be combined with *Monte-Carlo Tree Search* (Shen et al. 2019). However, this method requires problem grounding, which is inefficient for large problems like urban simulation (Staud 2022). Alternatively, domain-independent heuristics can be generated by providing a fixed set of features to a neural network (Gomoluch et al. 2017; Trunda and Barták 2020), or through the use of hypergraph networks (Shen, Trevizan, and Thiébaux 2020). In contrast, we aim to develop a domain-dependent heuristic designed to exploit the unique features and structure of a domain.

Symbolic Networks (Garg, Bajpai et al. 2020) achieve *state-of-the-art* performance on some planning benchmarks but are less relevant to our approach due to their focus on relational MDPs rather than HTN planning.

In this paper, we build upon the *Hierarchical World State Planning* (HWSP) algorithm (Staud 2022). This algorithm enhances total-ordered HTN planning by using a multi-layered world state representation, where the higher layers offer a more abstract perspective on the world state than the lower layers. By planning at a higher layer, there is no need to consider the details of the lower layers. The domain designer defines the effects of so-called separable abstract tasks, which can be applied directly without decomposition on the higher layers.

Our proposed deep learning method for predicting the effects of separable abstract tasks within the HWSP algorithm not only has the potential to improve planning performance but also to enhance plan quality, enabling more effective solutions for large-scale planning problems. We employ standard deep learning techniques (Géron 2022), allowing for easy adaptation as the state of the art changes. The use of specialized DNNs, such as ASNets, would anchor us to a specific architecture, limiting this flexibility. Similarly, our decision to convert the state into a 2D representation, in-

stead of directly feeding the state into the DNNs, stems from our goal of retaining compatibility with planning algorithms that use state representations beyond a set of atoms. This approach also facilitates the use of diverse DNN architectures, further broadening the applicability of our work. Furthermore, we utilize the same approach to estimate the number of steps to the goal, thereby creating a domain-dependent heuristic.

Our contributions are:

- A deep learning method for predicting the effects of abstract tasks within the *Hierarchical World State Planning* algorithm (Staud 2022).
- A deep learning method capable of heuristic learning. The heuristic can then be used by the HTN planning algorithm (Höller et al. 2019).

In the subsequent sections, we will first provide an overview of the Hierarchical World State Planning algorithm. Next, we will explain the process of inputting the world state into a neural network. We will then discuss the prediction mechanism and elaborate on the neural network architectures employed in our study. Lastly, we will present and analyze the results obtained from our experiments.

2 Hierarchical World State Planning

This section describes the Hierarchical World State Planning (HWSP) (Staud 2022) algorithm, which extends the hierarchical task network (HTN) planning paradigm. It introduces an innovative type of abstract tasks, referred to as *separable abstract tasks*, devised to enhance planning performance. This enhancement is achieved by partitioning the planning process into smaller, manageable sub-processes functioning across multiple layers of abstraction.

The set of all constants, variables, and atoms are designated as C, V, and A, respectively. A literal is an atom or its negation, while an atom is a predicate applied to a tuple of terms. Here, a term could be a constant or a variable. Every predicate p belongs to the set P.

2.1 Hierarchical Task Planning Domain

A hierarchical task planning domain is denoted as $D = (T_a, T_p, M)$, comprising three finite sets. Here, T_a is the set of abstract tasks, T_p represents primitive tasks, and M consists of methods. Both primitive and abstract tasks are tuples in the form $t(\overline{\tau}) = \langle prec_t(\overline{\tau}), \mathit{eff}_t(\overline{\tau})\rangle$. Each task is characterized by a precondition $prec_t(\overline{\tau})$, an effect $\mathit{eff}_t(\overline{\tau})$, and a set of parameters $\overline{\tau}$. The effects can modify atoms and fluents in the world state, while the precondition can examine if a particular atom is present in the world state or if a numerical equation is fulfilled.

A method is a tuple $m = \langle t_a(\overline{\tau}_m), P_m\rangle$, where $t_a(\overline{\tau}_m)$ is the abstract task it can decompose, and P_m is a set of plan steps, with $\overline{\tau}_m$ as the parameters of the method. An abstract task can be decomposed via methods into other tasks, where the plan steps P_m of method m replace the original abstract task in the plan.

A plan is a total ordered sequence of tasks. A problem is defined as a tuple $P = \langle init_P, goal_P, PS_P\rangle$, consisting of the initial plan PS_P, the initial state $init_P$, and the goal state $goal_P$. A solution is a plan where each task is an action (primitive task), satisfying its precondition at each step, thereby transforming the initial state into the goal state. A world state $w \in W$ is a set of atoms and a finite number of numerical fluents.

2.2 Multi-Layered World State

The HWSP algorithm divides the world state into n layers. The layer n encapsulates the actual world state containing factual information, whereas layer $l < n$ is more abstract than layer $l + 1$. Each predicate p and task t is associated with a specific layer $\hat{l}(p) = l$, and can only exist on that layer. The layer function is formulated as $\hat{l} : T_a \cup T_p \cup P \to \mathbb{N}$. Predicates at layer $l < n$ are derived predicates (Staud 2022; Edelkamp and Hoffmann 2004), while those at layer n are non-derived predicates. Derived predicates can only rely on predicates of layer $l+1$, facilitating information flow between layers. Consequently, the planning domain is then expressed as a tuple $D = (T_a, T_p, M, \hat{l})$.

2.3 Task Representation

Each task t is confined to having predicates in its effects that operate only on the same layer $\hat{l}(t) = l$. However, its preconditions can contain predicates p from layer $\hat{l}(p) < l$. Standard abstract tasks can only decompose into tasks on the same layer and lack effects. In contrast, separable abstract tasks must have effects, which must be defined by the domain designers. These effects should approximate the indirect influence the separable task have on the derived predicates at its layer.

Distinct from traditional abstract tasks that are instantaneously decomposed into a method, separable abstract tasks function as markers that signal the requirement for decomposition in a separate planning process (see section 2.6) that operates on layer $l+1$. These tasks are subsequently decomposed by a method akin to a standard abstract task in the new planning process.

It is important to note that unlike derived predicates in PDDL (Edelkamp and Hoffmann 2004), derived predicates can appear in effects within this planning algorithm. This is possible because they are treated as non-derived predicates in the child planning process.

2.4 Fluents and Functions

In contrast to the planning algorithm outlined by Staud (2022), our approach incorporates the support for numerical fluents as described by (Fox and Long 2003). These fluents are managed identically to predicates, and we make use of derived functions to handle these numerical fluents similarly to the way derived predicates are treated (Edelkamp and Hoffmann 2004).

The derived functions utilize numerical formulas rather than logical ones. These formulas support basic arithmetic operations such as addition, subtraction, multiplication, and division. Further, these derived functions provide the capability to aggregate the values of a set of numerical fluents using the sum operation.

The set over which the aggregation takes place is determined by a PDDL goal description (Kovacs 2011), in a manner akin to the forall operator used in PDDL. This implies that a derived function can apply a mathematical operation on a subset of numerical fluents defined by a PDDL goal description .

Notably, the derived functions can be modified via effects and can be involved in the preconditions of tasks in a child planning process. We also extend the capability to include operations that count the number of elements that fulfill a goal description. This allows for calculations such as averaging, which requires the count of the relevant fluents.

This makes derived functions a flexible and powerful tool in managing numerical fluents, expanding the range of possible planning tasks and scenarios that can be modeled.

2.5 Main Planning Algorithm

The main planning algorithm maintains a stack of planning processes, where only the top-most process is active. The algorithm comprises a main plan, which contains the final resultant plan, and the current world state, which is modified when a new primitive task is appended to the main plan (see figure 1). Each planning process is distinct, possessing its own unique plan and set of goals.

The initial planning process, which always operates on layer 1, uses the most abstract layer of the world state, the original problem goal, and the initial plan derived from the problem. The goal is restricted to predicates of layer 1. However, as these are derived predicates and depend recursively on predicates of layer n, any goal can be transformed into a form supported by the planner through the introduction of derived predicates.

The system is designed to support full backtracking, allowing it to continue across planning process boundaries (Russell et al. 2010). This feature ensures the completeness of the planning algorithm (Staud 2022).

2.6 Child Planning Algorithm

In each iteration, the main planning algorithm triggers the *child planning algorithm* within the top-most *planning process* on the stack. This process functions on a distinct layer l. The child planning algorithm subsequently returns a new task, which can either be a primitive task or a separable abstract task. If it is a primitive task, it is incorporated into the main plan, and the current world state is updated by applying the task's resultant effects. If the task's preconditions are unfulfilled or if no task is returned, the system reverts via backtracking. If the returned task is a separable abstract task, a novel planning process is set up to realize its effects, operating on layer $l + 1$. A planning process functions in the following manner:

Planning Process Initialization: The initiating plan of a process is comprised of tasks present in the method that was used to decompose the separable abstract task that created it. The goal of the planning process aligns with the effects of the task (Staud 2022).

Method Selection and Decomposition: The child planning process uses the Monte Carlo Tree Search (MCTS) algorithm (Kocsis and Szepesvári 2006), complemented by forward decomposition, to generate plans (with the H0 heuristic (Ghallab, Nau, and Traverso 2004)). Instead of producing a complete plan as might be expected, the child planning process returns a single task that has a high likelihood of guiding the main planning process towards the goal. The child planning process treats tasks differently: separable abstract tasks are treated as primitive tasks, with no need for further decomposition, whereas derived predicates are handled as non-derived predicates. However, standard abstract tasks are still decomposed through methods, as their decomposition is strictly restricted to tasks within the same layer.
This unique approach enables the child planning algorithm to formulate plans on the abstract layer as though dealing with a non-hierarchical planning problem. A significant reduction in the world state is achieved by exclusively utilizing the horizon (see section 2.7, thereby enhancing performance. It is crucial for the declared effects of the separable abstract tasks, as defined by the domain designer, to closely align with their actual effects when fully executed in the main planner. To increase precision, we employ neural networks in the method proposed in this paper. This approach leads to a decrease in the required amount of backtracking and an overall enhancement of plan quality

Planning Process Termination: Upon accomplishing its goal in the world state of the main planning process, a child planning process is discarded from the stack.

2.7 Horizon and Planning in the Planning Process

The horizon, composed of a set of atoms and fluents, plays a pivotal role in the child planning algorithm. It encapsulates only the portion of the world state pertinent to the child planning process. All atoms and fluents present within the task decomposition graph of the separable abstract graph, as outlined by (Bercher, Keen, and Biundo 2014), are included in this set. Importantly, the horizon is not a static construct, it is dynamically reconstructed from the world state of the main planning algorithm every time a new task is added the main plan (Staud 2022).

2.8 Completion of Planning

Planning processes are tasked with identifying subsequent actions for integration into the main plan. Upon the successful fulfillment of the original planning problem's goal, the main planning process is finished. Then, the main plan contains the result. If no plan could be found the algorithm returns an empty plan.

2.9 Illustrative Example

The concepts discussed in this paper can be best understood through an example scenario: a robot navigation challenge in a grid environment populated with obstacles. The robot's task is to navigate from its starting position to a designated goal location while circumventing these obstacles.

Layered Representation The world state can be represented in multiple layers:

Algorithm 1: The Main Planning Algorithm of the Hierarchical World State Planning (HWSP) (Staud 2022) algorithm.

```
1  Function MainPlanningProcess:
2   while not stack.isEmpty():
3    cp = stack.top
4    new_task = cp.getNextTask()
5    if (cp.finished())
6     stack.remove(cp)
7    if new_task is Primitive_Task:
8     add_to_plan(new_task)
9     update_world_state(new_task)
10   elif new_task is
         Separable_Abstract_Task:
11    newp = new PlanningProcess(new_task)
12    stack.push(newp)
13   else:
14    backtrack()
15   if goal_achieved():
16     return plan
17   return None
```

- Layer n: This represents the actual grid, complete with the robot's current location, the goal location, and the obstacles.
- Layer $l < n$: This is a simplified rendition of the grid, with regions consolidated into higher-layer zones (e.g., rooms within a building).

Planning Process The primary planning algorithm begins with the most abstract layer, where the objective is to maneuver the robot from one high-layer zone to another. As the algorithm advances, it may encounter separable abstract tasks, such as transitioning from one room to another. In such an instance, a fresh planning process will be established to operate on the subsequent layer ($l+1$), offering a more detailed view of the environment.

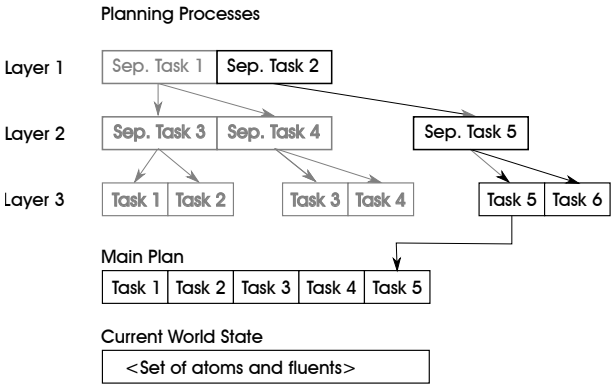


Figure 1: Adapted from the paper by Staud (2022), this figure provides an example of what the state of the main planning algorithm might look like in a domain with 3 layers. Completed processes are marked in grey. The current state of the stack would be: Sep. Task 1, Sep. Task 5.

3 Representation of the World State

Deep Learning usually takes a fixed-size representation as input (Géron 2022). To generate it out of the world state and enable predictions, a 2D representation is chosen due to its suitability for planning domains that are fundamentally two-dimensional, such as ware transport and robot steering. For domains with different dimensionalities, 1D or 3D representations can be used, which makes the algorithm, with modifications, also usable in this case. The usage of a 2D input image to guide planning is well explored in reinforcement learning (Mnih et al. 2013) or algorithm selection for planning (Sigurdson et al. 2019; Kaduri, Boyarski, and Stern 2020) and has been shown to be effective. Note that we will only project one layer of the multi-layer world state as the planning processes also have only access to one layer of the world state.

A 2D domain is a domain for which a projection $f_p : W \to I$ of its world state w into a 2D space I exists that has the following properties:

- $i \in I$ is a RGB pixels image. This means it is a matrix with dimensions $M \times N \times 3$.
- f_p is consistent: let $w_1 \in W$ and $w_2 \in W$ be two world states. When they are equivalent in regarding planning, then it holds $f_p(w_1) = f_p(w_2)$. When w_1 and w_2 are not equivalent it holds $f_p(w_1) \neq f_p(w_2)$.
- f_p is reversible: for all 2D images $i \in I$ that were generated by f_p we can construct a world state $w \in W$ such that $f(w) = i$. And this world state is equivalent to the original world state in terms of planning.
- The encoding of f_p should be in a way that allow a deep learning algorithm to recognize and learn patterns. So the spacial organization of the image is meaningful and informative.

The last property isn't fully formalized due to ongoing research about what specific neural networks can detect. Important is that the spatial locality is preserved and that the complexity in the output image is minimized while still preserving the information.

We propose a series of annotations to create the required domain-specific projection function. These annotations, added to a PDDL domain or problem file (Fox and Long 2003), can be used to automatically render a world state to an image. The annotation system allows the user to define position and shape information for every atom, object, predicate or function. Annotations are inspired by Java annotations (Gosling et al. 2005), and multiple annotations are possible.

- `;@position(vec2(x,y))`: Defines the position of an object, with x and y as floats in image coordinates.
- `;@position(arg)`: Sets the position of an atom or fluent equal to the object of the given argument. It is used as a shift in combination with `;@propagate`.
- `;@appearance(shape, size, zlayer, {<paint attributes>})`: Defines the appearance of an object, with the shape as either a box, ellipse, PNG, or a line; the size as a `vec2`; and the `zlayer` as the draw order. The paint attributes contain pen and brush style, with colors specified directly or derived from a fluent. Each object also has an internal instance

ID, which is automatically encoded via hashing into the final color of the object to differentiate between multiple instances in the final image.

- `;@propagate(source, target)`: Can only be applied to an atom. Propagates the position of the object in the source argument to the object in the target argument.

The line shape can only be applied to an atom or fluent and takes the position of two objects in the arguments to determine the orientation of the line. The objects on the same z-layer are not rendered on top of each other even if they have the same position. Instead, they will be arranged deterministically by a layouting algorithm to ensure that every atom, object, or fluent is visible. We support other features like, for example, a slot mechanism to give the user more control but we will only present the basic features in this paper.

The annotations are placed in the comments so that if a PDDL planner does not support them, they can be ignored. The resulting projection function can be used not only for rendering the world state but also for rendering the goal state.

3.1 Algorithm

The algorithm accepts as input a world state, consisting of a set of atoms and a finite number of numerical fluents. It comprises two integral functions:

The *draw* function initiates a set for object positions, defined directly via annotations in the problem file, and computes the appearance and rectangle information for each object. This function then proceeds to call the *propagate* function iteratively until no further propagation transpires. Post initial propagation, the function processes any residual objects that have not been assigned positions, bestowing default positions and appearance information. Another round of propagation is undertaken to update the positions of these newly handled objects, continuing until no additional propagation occurs. The *draw* function concludes by sorting objects according to their z-layer, thereby ensuring objects with lower z-layer values are rendered first. The object's attributes such as shape, size, z-layer, position, and paint are taken into consideration during this drawing phase. The function adeptly handles a variety of shapes, including boxes, ellipses and lines, adjusting the pen and brush styles and colors as required.

The *propagate* function accepts a set of determined object positions, the world state, and a map describing child-parent relationships as defined by the annotation. It traverses the atoms contained within the world state and applies propagation rules specified by the annotations to compute new object positions. If an object has a valid position, it propagates this position to its child objects. If the space of a child object is already occupied by another object on the same layer, thus violating the properties of the projection function, the child object will be displaced until a free space in the same z-layer is located. A straightforward rule is employed for this, shifting any object along the x-axis to the right until a free space is found in the same z-layer or until the right image border is reached.

Thus, objects are arranged in a deterministic manner. Note, it is quite straightforward for a PDDL domain designer to design an annotation that breaches the conditions of the projection function, such as insufficient space in the image, child-parent relationships not being visible in the image due to excessive object displacement, or predefined object positions overlapping. Under such circumstances, the system responds by returning an error message to the user.

3.2 Example

The proposed annotation system's utility is demonstrated via an example from the airport domain (Anders 2015). This domain encompasses three types of objects: airports, planes, and cargos, with the objective being to transport cargo from one airport to another utilizing planes.

Here is a simplified annotated PDDL domain for this problem:

```
(define (domain airport)
  (:types
    gobject - object
    ;@appearance(box, vec2(4,4), 0, {
        color = white})
    airport - gobject
    ;@appearance(box, vec2(2,2), 1, {
        color = magenta})
    cargotype - gobject
    ;@appearance(ellipse, vec2(3,3), 1,
        {color = red})
    planetype - gobject
  )
  (:predicates
    ;@propagate(?obj2, ?obj1) ;@position
        (vec2(2,2))
    (in ?obj1 - gobject ?obj2 - gobject)
    ;@propagate(?airport, ?obj1) ;
        @position(vec2(-2,2))
    (at ?obj1 - gobject ?airport -
        airport)
    (cargo ?obj1 - gobject)
    (plane ?obj1 - gobject)
    (airport ?obj1 - gobject)
  )
  ... ; Rest of the domain
)
```

In the provided PDDL file, different object types' appearances are defined using `;@appearance` annotations. For instance, an airport is denoted by a 4x4 pixel white box as specified by the `;@appearance(box, vec2(4,4), 0, color = white)` annotation.

The `;@position` and `;@propagate` annotations determine how object positions are established. For instance, the `;@propagate(?obj2, ?obj1)` annotation associated with the "in" predicate suggests that the position of `?obj1` mirrors that of `?obj2`, and the `;@position(vec2(2,2))` annotation stipulates that `?obj1` is displaced by `vec2(2,2)` from the position of `?obj2`. In the PDDL problem file, the specific positions of the airports are established.

This annotated PDDL domain can now be leveraged by the planning algorithm to generate a two-dimensional projection of the world state, which can serve as an input to a

deep learning algorithm.

4 Prediction for Planning

In this section, we discuss how we employ neural networks to predict various important aspects of planning, including the effects of separable abstract tasks and the number of actions required to reach a goal.

4.1 Effects of Abstract Tasks

We aim to predict two key factors for separable abstract tasks: simple effects and fluent effects.

Simple Effects Simple Effects (McDermott et al. 1998) add or remove atoms. Our goal is to predict whether these effects will be fulfilled when a separable abstract task is executed. We use a neural network that returns a probability corresponding to the likelihood that the effect will be applied. To train the network we are using the binary cross-entropy loss functions (Good 1952).

Fluent Effects Fluent effects (Fox and Long 2003) change fluents, which are numerical values. In this case, we do not predict a probability; instead, we directly predict the delta value (residual learning) of the fluent using a neural network. To train the network we are using mean absolute error (MAE) (Bishop 2006) as loss function. The network performs *deep regression* which is competitive to problem specific estimators (Lathuilière et al. 2019).

Input To predict these effects, we first encode both the current world state and the world state after the separable abstract task was applied with the default effects defined by the domain designer (Staud 2022). This process results in a rough approximation of the new world state. However, this approximation is sufficient for the neural network to identify which separable abstract task was applied. Note that when the prediction of the effects is not correct only the performance of the algorithm will decrease, it will not affect completeness (Staud 2022).

Both world states are then converted into two separate 2D representations, which are subsequently provided as input along with their delta to the neural network. In addition, we also provide the current values of the fluents to assist the neural network in predicting the delta changes of the fluents.

4.2 Heuristic Learning

Deep learning can be used to create planning heuristics, which estimate the number of planning steps required to reach a goal (Shen et al. 2019; Gomoluch et al. 2017; Shen, Trevizan, and Thiébaux 2020). We train a neural network using a 2D representation of the current world state and the goal state. The training data is generated from example domains and includes the states, and the actual number of steps required to reach the goal. We determine the number of steps using a planning system, which supports PDDL 2.1 (Fox and Long 2003) and can return the minimal amount of actions to reach a goal. As loss function we use MSE (Bishop 2006). The network performs *deep regression* which is competitive to problem specific estimator (Lathuilière et al. 2019).

It is not easy to generate the (full) goal state out of the initial state and the goals in the problem. In our tests, we determined the set of predicates which were stationary and simply combined them with the goal from the problem. This is mostly sufficient. If not the domain designer has to provide additional annotations so that the (full) goal state is generated correctly.

It's important to note that the resulting heuristic is likely to be non-admissible (Ghallab, Nau, and Traverso 2004), as we're approximating the number of steps. This heuristic can then be used in an HTN planner (Höller et al. 2019).

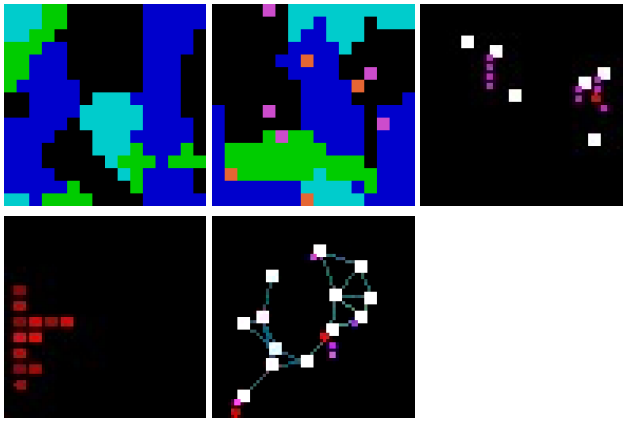


Figure 2: The images displayed, arranged from left to right and top to bottom, represent the following domains: City1, City2, Airport, Blocksworld, and Transport. These graphical outputs were generated by applying the projection functions, as defined by the annotation process detailed in section 3. They provide a visual representation of each domain's state.

4.3 Architecture of the Neural Network

We employ three distinct types of neural networks to evaluate our approach:

- **Conv**: This network consists of three convolution layers (LeCun et al. 1998) without max pooling, followed by a fully connected dense layer. Batch normalization (Ioffe and Szegedy 2015) is applied after each layer:
  - Conv2D(16, kernelsize=1), ReLU
  - Conv2D(32, kernelsize=3), ReLU
  - Conv2D(64, kernelsize=3), ReLU
  - Flatten()
  - Dense(n, activation=<see text>)

  The network has 115940 weights. It uses the convolution layers to first determine complex features in the input and then uses a highly connected dense layer for regression.
- **Conv Max**: Like *Conv*, but this network includes max pooling $(2, 2)$ after the convolutional layers (not including the first one). The network has 26340 weights, and this kind of architecture is usually used for image classification tasks (LeCun et al. 1998). Additionally, it can also be used for the analysis of non-visual 2D data, like a spectrogram of a word (Warden and Situnayake 2019).
- **ResNet**: This network is a truncated version of *ResNet-34*, which is used for image classification and showed

Domain Variable	Conv Fluents	Integer	Atoms	Conv Max Fluents	Integer	Atoms	ResNet Fluents	Integer	Atoms
City1 All	0.3523	0.3732	0.1295	0.3885	0.3756	0.2830	0.0075	0.0087	0.088
City1 Industrial	0.2547	0.2674	0.0299	0.2765	0.2832	0.2181	0.0177	0.0204	0.0109
City1 Residential	0.2531	0.2575	0.0324	0.2805	0.2932	0.2243	0.0199	0.0256	0.0021
City1 Commercial	0.2488	0.2763	0.0415	0.2751	0.2812	0.2256	0.0178	0.0254	0.0166
City2 All	0.1271	0.1343	0.2456	0.2812	0.3012	0.3042	0.0090	0.0120	0.0186
City2 Industrial	0.3083	0.3123	0.1087	0.2726	0.2823	0.2304	0.01328	0.0170	0.0162
City2 Residential	0.3158	0.3323	0.1021	0.2675	0.2742	0.2202	0.0204	0.0232	0.01256
City2 Commercial	0.3212	0.3202	0.1081	0.2695	0.2743	0.2248	0.0191	0.0205	0.0202
City2 Mine	0.2834	0.3004	0.0907	0.2434	0.2404	0.2193	0.0235	0.0223	0.0158

Table 1: Prediction of separable abstract task effects using three neural networks (Conv, Conv Max, and ResNet) across different city domains. The table shows the mean absolute error (MAE) (Bishop 2006) on normalized data and the binary cross-entropy loss for atoms. The networks were trained on 200 epochs, although fewer epochs may suffice for less stringent accuracy requirements. The dataset size was 16384, with 20% of the data used for validation (Bishop 2006). The size of each 2D representation was 16×16.

Domain	Conv	Conv Max	ResNet	ResNet +H0
Airport	1.4943	0.8883	0.8272	0.2899
Blocksworld	4.345	1.5293	1.5322	1.995
Transport	3.423	1.5075	1.6820	0.5117

Table 2: Heuristic learning results for various domains using Conv, Conv Max, and ResNet neural networks. Displayed is the unnormalized loss when predicting the number of stes to the goal. The input data size is larger (64×64), requiring two additional residual or convolutional units. ResNet+H0 represents a combination of the traditional H0 planning heuristic with a neural network, where the neural network outputs a delta value that is added to the H0 heuristic. It shows the best results.

Probability	City1	City2
0.01	6.2480	6.4520
0.1000	9.1330	12.8180
0.2000	15.2060	34.3300
0.3000	29.6240	100.1370
0.4000	59.2270	402.3630
0.5000	134.0850	2066.6430

Table 3: The table demonstrates the influence of prediction quality on the average number of planning processes generated by the primary planning algorithm. The "Probability" column indicates the frequency at which an atomic effect is inaccurately predicted. Such mispredictions necessitate backtracking in the overall system, which consequently increases the number of planning processes. A higher volume of planning processes typically leads to a decrease in overall system performance. Hence, the improvement of prediction quality can significantly enhance the planning algorithm's efficiency by reducing the need for backtracking and subsequently minimizing the number of planning process processes required.

very high performance in this task (He et al. 2016):
  - Conv2D(64, kernelsize=7, stride=2), BatchNormalization, Relu, MaxPooling2D($(3, 3)$, stride=2)
  - ResidualUnit(64, stride=1) $\times 3$
  - ResidualUnit(128, stride=2)
  - ResidualUnit(128, stride=1) $\times 3$
  - GlobalAvgPool2D(), Flatten()
  - Dense(n, activation=<see text>)

  The network has 1367524 weights. The ResidualUnit is composed of two convolution layers with ReLU activation and batch normalization, in addition to a skip connection (He et al. 2016).

In each neural network, the dense layer employs linear activation (Géron 2022) for predicting fluents and sigmoid activation (Rumelhart, Hinton, and Williams 1986) for atoms. To enable efficient training, we implement data normalization (Bishop 2006).

5 Results

In this section, we first present results from training the neural network. The domains were manually annotated domains in order to create a 2D representation of the world state.

- **Prediction of Abstract Task Effects** (see Table 1): We implemented a city domain like to the one used in the paper (Staud 2022) to test the prediction of abstract task effects which we will call *City1*. We also created a more sophisticated urban domain called *City2*, which will additionally simulate effects like an electric power grid, criminality, and fires. In these domains, we consider several separable abstract tasks (industrial, residential, commercial, mine). The training data set is created via planning and simulation of the effects of the abstract tasks.
  We used one neural network to predict the effects of all abstract task. Additionally we also measured what happens when we use a seperate neural network for each abstract task.
- **Heuristic Learning** (see Table 2): We test the heuristic learning approach on the airport domain (Anders 2015), as well as the blocksworld and transport domains, which

were presented as challenges in the International Planning Competition and are available on GitHub (Seipp, Torralba, and Hoffmann 2022).

The data presented in the tables 1 and 2 indicates that the impact of an abstract task and the number of steps to the goal can be approximated with a high degree of accuracy using neural networks. As anticipated, the ResNet variant outperformed the other two networks. Intriguingly, the general ResNet variant, which approximates all abstract tasks simultaneously, yielded better results than training separate neural networks for each abstract task. Conversely, the Convolution variants exhibited improved performance when individual networks were trained. The performance of the Conv Max network equals or surpasses that of the standard Conv network when predicting fluents; however, this superiority is not observed in the prediction of atoms. This suggests that the Max Pooling operation may inadvertently discard crucial information necessary for accurate atom prediction. On the other hand, Conv might perform poorly on large representations due to the dense network.

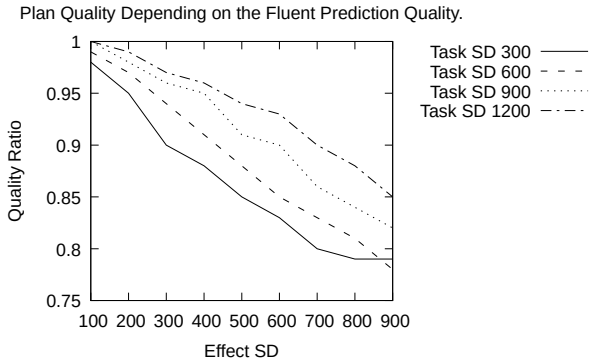


Figure 3: The graph illustrates how the quality of numerical fluent effect predictions impacts plan quality within a simplified 'City1' domain. This domain has steady income-building tasks (which make buildings) without secondary effects like crime or fire. The main plan quality metric is the five-year city treasury income. Ideally, the planner would select the highest income-generating building at each step, but prediction errors skew this process. The 'quality ratio' (y-axis) is the actual gain to maximum possible gain. We assumed a *Gaussian prediction* error distribution with a mean of 0, and the standard deviation denoted by 'Effect SD' (x-axis). The curves show the influence of different numerical fluent effect distribution in the tasks. Notably, diverse original numerical effects lessen the error's impact on plan quality, while similar fluent effects heighten the influence of prediction errors, decreasing plan quality.

It is also noteworthy that the measurements can be divided into two categories: those that predict the data with high accuracy and those that have a relatively low accuracy. This observation suggests that when the accuracy is low, the neural network might not have successfully fully generalized the underlying structure of the problem. In our experiments, we always employed separate networks to predict atoms and fluents because we saw a performance drop when using a single network for both tasks.

We conducted an evaluation to measure the influence of our method on planning performance and the quality of the plan produced. Table 3 provides a comparative analysis of the number of planning processes generated, depending on the probability of an incorrect prediction of an atom's effect. An erroneous prediction triggers backtracking. For this specific analysis, the planner employs the *City1* domain (Staud 2022). The *City2* domain will create more planning processes as its tasks have more effects and backtracking is necessary when a single one fails.

Furthermore, we evaluated the impact of inaccurately predicted numerical fluents on the quality of the plan. The objective within the *City1* domain is to optimize the quantity of money generated by the city. Graph 3 depicts the ratio of actual money generated to the optimal possible generation of money.

Upon reviewing the results, it appears that the most effective course of action for automatic planning is to utilize a ResNet, as this network consistently delivers the best outcomes. The primary drawback, however, is the substantial amount of data that must be processed each time the network is evaluated. As such, it would be prudent to primarily adopt this approach in complex domains like *City1* or *City2* where a traditional general heuristic would take too much time to evaluate.

6 Conclusions

In this paper, we have presented a practical and simple method for leveraging deep learning to improve the performance of automatic planning. Our approach requires no complex data structures and can be easily integrated into existing planners. Notably, the ResNet+H0 combination demonstrates a significant improvement over other tested networks. This finding hints at the potential of combining traditional planning heuristics with neural networks to achieve superior performance.

Looking forward, we plan to extend our approach to automatically generate abstraction hierarchies for hierarchical world state planners. This will allow us to transform non-hierarchical domains into hierarchical ones and enjoy the performance benefits which come with the hierarchical world state (Staud 2022). Overall, we believe that our research provides a promising direction for integrating deep learning with automated planning systems to achieve better performance and scalability.

References

Anders, A. 2015. Planning Exercises. Https://github.com/-arii/planning_exercises, Accessed: 20.3.2023.

Arulkumaran, K.; Deisenroth, M. P.; Brundage, M.; and Bharath, A. A. 2017. Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine*, 34(6): 26–38.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid Planning Heuristics Based on Task Decomposition Graphs. In *SoCS 2014*, 35–43. AAAI Press.

Bishop, C. M. 2006. *Pattern Recognition and Machine Learning*. Berlin, Heidelberg: Springer-Verlag.

Edelkamp, S.; and Hoffmann, J. 2004. PDDL 2.2: The Language for the Classical Part of IPC-4. In *Int. Planning Competition*.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*.

Garg, S.; Bajpai, A.; et al. 2020. Symbolic Network: Generalized Neural Policies for Relational MDPs. In *International Conference on Machine Learning*, 3397–3407.

Géron, A. 2022. *Hands-On Machine Learning With Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Inc.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Elsevier.

Gomoluch, P.; Alrajeh, D.; Russo, A.; and Bucchiarone, A. 2017. Towards Learning Domain-Independent Planning Heuristics. *CoRR*.

Good, I. J. 1952. Rational Decisions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 14(1): 107–114.

Gosling, J.; Joy, B.; Steele, G.; and Bracha, G. 2005. Java (TM) Language Specification (The 3rd Edition).

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 770–778.

Hogg, C.; Kuter, U.; and Munoz-Avila, H. 2010. Learning Methods to Generate Good Plans: Integrating HTN Learning and Reinforcement Learning. In *Proceedings of the AAAI Conference on AI*, volume 24, 1530–1535.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2019. On Guiding Search in HTN Planning with Classical Planning Heuristics. In *IJCAI*, 6171–6175.

Huang, Y. 2020. *Hierarchical Reinforcement Learning*, 317–333. Singapore: Springer Singapore.

Ioffe, S.; and Szegedy, C. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning*, 448–456. Proceedings of Machine Learning Research.

Kaduri, O.; Boyarski, E.; and Stern, R. 2020. Algorithm Selection for Optimal Multi-Agent Pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 161–165.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *ECML*, 282–293. Springer.

Kovacs, D. 2011. BNF Definition of PDDL3.1: Completely Corrected, Without Comments. *Unpublished Manuscript from the International Planning Competition Website*.

Lathuilière, S.; Mesejo, P.; Alameda-Pineda, X.; and Horaud, R. 2019. A Comprehensive Analysis of Deep Regression. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(9): 2065–2081.

LeCun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. 1998. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11): 2278–2324.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - The Planning Domain Definition Language. *AIPS-98*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with Deep Reinforcement Learning.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. 2020. Integrating Acting, Planning, and Learning in Hierarchical Operational Models. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 478–487.

Rivlin, O.; Hazan, T.; and Karpas, E. 2020. Generalized Planning With Deep Reinforcement Learning. *CoRR*.

Rumelhart, D. E.; Hinton, G. E.; and Williams, R. J. 1986. Learning Representations by Back-Propagating Errors. *Nature*, 323(6088): 533–536.

Russell, S. J.; Norvig, P.; Canny, J. F.; Malik, J. M.; and Edwards, D. D. 2010. *Artificial Intelligence: A Modern Approach*. Prentice Hall Upper Saddle River, 3 edition.

Seipp, J.; Torralba, Á.; and Hoffmann, J. 2022. PDDL Generators. . Https://github.com/AI-Planning/pddl-generators, Accessed: 20.3.2023.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 574–584.

Shen, W.; Trevizan, F.; Toyer, S.; Thiébaux, S.; and Xie, L. 2019. Guiding Search with Generalized Policies for Probabilistic Planning. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, 97–105.

Sigurdson, D.; Bulitko, V.; Koenig, S.; Hernandez, C.; and Yeoh, W. 2019. Automatic Algorithm Selection in Multi-Agent Pathfinding. *CoRR*.

Staud, M. 2022. Urban Modeling via Hierarchical Task Network Planning. *HPlan 2022*, 73.

Toyer, S.; Thiébaux, S.; Trevizan, F.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *Journal of Artificial Intelligence Research*, 68: 1–68.

Trunda, O.; and Barták, R. 2020. Deep Learning of Heuristics for Domain-independent Planning. In *ICAART*, 79–88.

Vezhnevets, A. S.; Osindero, S.; Schaul, T.; Heess, N.; Jaderberg, M.; Silver, D.; and Kavukcuoglu, K. 2017. Feudal Networks for Hierarchical Reinforcement Learning. In *International Conference on Machine Learning*, 3540–3549. Proceedings of Machine Learning Research.

Warden, P.; and Situnayake, D. 2019. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media.

On Guiding Search in HTN Temporal Planning with non Temporal Heuristics

Nicolas Cavrel, Damien Pellier, Humbert Fiorino

Univ. Grenoble Alpes - LIG
Grenoble, France
{nicolas.cavrel, damien.pellier, humber.fiorino}@univ-grenoble-alpes.fr

Abstract

The Hierarchical Task Network (HTN) formalism is used to express a wide variety of planning problems as task decompositions, and many techniques have been proposed to solve them. However, few works have been done on temporal HTN. This is partly due to the lack of a formal and consensual definition of what a temporal hierarchical planning problem is as well as the difficulty to develop heuristics in this context. In response to these inconveniences, we propose in this paper a new general POCL (Partial Order Causal Link) approach to represent and solve a temporal HTN problem by using existing heuristics developed to solve non temporal problems. We show experimentally that this approach is performant and can outperform the existing ones.

Introduction

Among planning formalisms, Hierarchical Task Network (HTN) planning is one of the most expressive. In addition to classical STRIPS actions, HTN allows to express complex abstract tasks in the form of decompositions into subtasks and their ordering constraints. HTN has been used in a wide variety of applications (Barreiro et al. 2012; Lallement, de Silva, and Alami 2018; Milot et al. 2021). However, despite the pioneering work of (Lemai 2004; Au et al. 2003; Goldman 2006), few approaches have been proposed to deal with time in HTN planning.

Temporal HTN approaches can be classified according to the expressiveness of the solution plans they can generate in the sense of Cushing's classification of temporal problems (Cushing 2007). This classification defines three categories of temporal planning problems. The first category contains the temporal problems whose solutions are solely sequential solution plans (non-concurrent solution plans). The second one contains the temporal problems whose solution plans can possibly be concurrent, but for which there exists a sequential solution plan (possibly concurrent solution). Finally, the third one includes the temporal problems for which the only existing solution plans are necessarily concurrent (necessary concurrent solution).

Most of the approaches are able to solve only temporal problems of the first two Cushing's categories. Among these approaches, we distinguish state-space approaches (Au et al. 2003; Asuncion et al. 2005). These approaches deal with temporal actions as classical ones by either preprocessing the temporal actions into a sequence of classical tasks (Au et al. 2003), or by compiling temporal actions into classical actions. The latter approach is also widely used in STRIPS planning (Fox and Long 2003; Celorrio, Jonsson, and Palacios 2015). Approaches converting temporal problems into classical ones can benefit from the classical search heuristics and algorithms. At the same time, several plan-space approaches have been proposed (Younes and Simmons 2003; Bechon et al. 2014; Bit-Monnot et al. 2020). They aim at refining an initial HTN into a solution by refining tasks and building causal relations between them, and use *STN* (Simple Temporal Networks) (Dechter, Meiri, and Pearl 1991) to deal with temporal constraints. These approaches are relatively efficient but suffer from their lack of flexibility, especially when dealing with real problems, where tasks are necessarily concurrent.

To our best knowledge, the only approach capable of producing expressive plans of the third Cushing's category is an algorithm in plan-spaces that handles constraints using *Chronicles* (Bit-Monnot et al. 2020). Chronicle approaches keep tracks of the value of each proposition with regards to time into a *timeline*, then try to find non conflicting timelines for every proposition. These approaches are very expressive but suffer from their lack of efficiency due to the lack of informative heuristics.

In this paper, we propose a new planning approach for hierarchical temporal planning able to solve planning problems for the two first Cushing's categories, called HTEP (*Hierarchical Temporal Event Planner*). HTEP is based on POCL (*Partial Ordered Causal Link*) (Bercher, Keen, and Biundo 2014). The particularity of our approach is to be both flexible (HTP produces partial temporal solution plans) and to be able to exploit the heuristics developed for non temporal HTN planning for plan and flaw selection. To do so, HTEP starts by compiling a temporal problem into a non temporal one by refining abstract tasks and durative tasks into instantaneous non temporal actions. Then, it tries to find a solution plan by relaxing the duration constraints on tasks and by checking only their consistency. The relaxed non-temporal problem is expressed as a partial plan in the POCL semantics. Hence classical POCL search algorithms and heuristics can be used to solve it. Finally, if a solution to the relaxed problem is found, HTEP tries to find timestamp assignments matching the temporal constraints by using a

simple CSP solver.

The paper is organized as follows. Section 1 introduces the temporal planning formalism. In the second section we present our planner, from the relaxed problem to the search algorithm solving it, with the heuristic used to guide it. Finally, the third section shows the performance results of HTEP.

Problem Statement

In this section we propose a formalization of a temporal HTN planning problem and its solution. The notations are based on (Höller et al. 2020) and (Abdulaziz and Koller 2022) to deal with time.

Action, Methods, Tasks and Plan

A key concept in HTN planning and a fortiori in temporal HTN planning is the concept of task. Each task is given by a name and a list of parameters. We distinguish three kinds of tasks: the snap actions, the durative actions and the abstract tasks. Unlike snap actions that do change the state of the world, *durative tasks* and *abstract tasks* do not. They are names referring to other tasks (either snap, durative or abstract) that must be achieved with respect to some constraints. We consider every task has a start and an end time point. We refer to the start and the end time points of a task t with temporal variables denoted respectively v_t^s and v_t^e. Since a snap action t is instantaneous, $v_t^s = v_t^e$, thus we will simply refer to the time point of the snap action t as v_t.

The durative actions and the abstract tasks can be refined respectively by applying *snap actions* and *methods* defined below.

A *snap action* a is nearly an action in the sense of classical planning, i.e., a tuple $(\textit{name}(a), \textit{precond}(a), \textit{effect}(a))$. $\textit{name}(a)$ is the name of a. The preconditions $\textit{precond}(a)$ and effects $\textit{effect}(a)$ are sets of ground predicates. Let v_a be the time point at which a is supposed to be executed. a is executable if $\textit{precond}(a)$ hold strictly before v_a. As in classical planning, the execution of a produces the effects $\textit{effect}(a)$ such that $\textit{effect}(a) = \textit{effect}^+(a) \cup \textit{effect}^-(a)$ and $\textit{effect}^+(a) \cap \textit{effect}^-(a) = \emptyset$ where $\textit{effect}^+(a)$ and $\textit{effect}^-(a)$ are conjunctions of predicates, respectively true and false after the execution of a. Finally, we say that a snap action a refines a task t if $t = \textit{name}(a)$.

A *durative action* a is a tuple $(\textit{name}(a), \textit{start}(a), \textit{end}(a), \textit{inv}(a), d)$: $\textit{name}(a)$ is the name of a, $\textit{start}(a)$ and $\textit{end}(a)$ are snap actions ; $\textit{inv}(a)$ is a set of ground predicates that must hold after the execution of $start(a)$ and until the beginning of $end(a)$, i.e., on the interval $]v_a^s, v_a^e[$ and $d = v_a^e - v_a^s$ is the duration of a. We assume as PDDL 2.1 (Fox and Long 2003) that $v_a^s < v_a^e$ is true. Therefore the duration of a is a strictly positive number. Similarly to a snap action, a durative action refines a task t if $t = \textit{name}(a)$.

An *abstract task* must be decomposed into durative actions in order to be performed. The several ways of decomposing an abstract task are described through *methods*. Even though we cannot set a duration for an abstract task in the general case, the start and end point of an abstract can still be subject to temporal constraints.

A *method* m is a tuple $(\textit{name}(m), \textit{task}(m), \textit{subtasks}(m), \alpha, \textit{constraints}(m))$, where $\textit{name}(m)$ is the name of the method, $\textit{task}(m)$ is the task refined by the method, $\textit{subtasks}(m)$ the set of tasks symbols (possibly empty) which refines $\textit{task}(m)$, $\alpha : \textit{subtasks}(m) \mapsto \mathcal{T}$ maps the task symbols to a set of task names and $\textit{constraints}(m)$ is a set of temporal ordering constraints over $\textit{subtasks}(m)$. Temporal ordering constraints are defined over the time variable start or the end of the subtasks $\textit{subtasks}(m)$ of m. The possible qualitative temporal ordering constraints are those from the classical point algebra (Broxvall and Jonsson 2003): $<$, $\leq$, $>$, $\geq$, $=$ and $\neq$. For instance, the temporal ordering constraint $v_{t_1}^s < v_{t_2}^e$ expresses that the start of the task t_1 must occur strictly before the end of t_2. A method m refines a task t if $t = task(m)$. Note, that consistency checking of such a set of constraints C can be refined by computing strongly connected components of the constraint graph associated in polynomial time $O(|C|)$.

A *partial temporal plan* π is a tuple $(T, \alpha, \mathcal{C}, \mathcal{L})$ where:

- T is a set of task symbols.
- $\alpha : T \mapsto \mathcal{T}$ a mapping from task symbols to task names.
- $\mathcal{C}$ is a set temporal ordering constraints over the tasks symbols in T. The constraints are like those used in methods.
- $\mathcal{L}$ is a set of causal links of the form $\langle t_i \xrightarrow{p} t_j \rangle$ with t_i and t_j two snap actions in T such as $(t_i < t_j) \in \mathcal{C}$ and $p \in \textit{effect}(\alpha(t_i))$ and $p \in \textit{precond}(\alpha(t_j))$ (classical causal link definition in POCL).

A snap task t_k in a partial temporal plan π is a *threat* on a causal link $\langle t_i \xrightarrow{p} t_j \rangle$ if and only if (1) t_k has an effect $\neg p$ and (2) the ordering constraints $(t_i < t_k)$ and $(t_k < t_j)$ are consistent with $\mathcal{C}$ if $t_i < t_j$.

A *flaw* in a partial temporal plan $\pi = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ is either (1) an open precondition, i.e., a precondition or a postcondition of task $t \in \mathcal{T}$ not supported by a causal link or (2) a threat, i.e., a task that may interfere with a causal link or (3) a task $t \in \mathcal{T}$ that is not a snap task.

Temporal HTN Planning Problem and Solution

A *temporal HTN planning problem* $\mathcal{P}$ is a tuple $(L, \mathcal{T}, \mathcal{A}, \mathcal{M}, s_0, \pi_0, g)$, where L is a finite set of logical propositions, $\mathcal{T}$ is a set of tasks, $\mathcal{A}$ is a set of durative actions and $\mathcal{M}$ is the set of methods, $s_0 \subseteq L$ is the initial state in the set of states $\mathcal{S}$, π_0 is the initial partial temporal plan, and $g \subseteq L$ is a of ground predicates describing the goal.

The solution of a temporal HTN planning problem is a partial temporal plan π obtained by refining an initial partial plan π_0 as in POCL planning built into snap tasks by applying methods and durative actions. Formally, a partial temporal plan π is solution of a planning problem $\mathcal{P} = (L, \mathcal{T}, \mathcal{A}, \mathcal{M}, s_0, \pi_0, g)$ if and only if:

1. π is a refinement of the initial partial temporal plan π_0: π_0 contains two special snap task: t_0 with no precondition but with s_0 as effects and t_∞ with the goal g as precondition but no effects and $v_{t_0}^e < v_{t_\infty}^s$ in $\mathcal{C}$.
2. π needs to be executable in the initial state s_0. Thus,

(a) all tasks in π are snap tasks,

(b) π has no flaws, i.e., no open precondition and no causal threats,

(c) for all t in π, v_t is assigned and satisfies the temporal constraints of π.

It remains to define how to refine a partial temporal plan into a plan containing only snap actions by using methods and durative actions.

First, consider the case of the method refinement. Let $m = (\mathit{name}(m), \mathit{task}(m), \mathit{subtasks}(m), \alpha, \mathit{constraints}(m))$ be a method that refines a task t and plan $\pi_1 = (T_1, \alpha_1, \mathcal{C}_1, \mathcal{L}_1)$ a plan such that $t \in \mathcal{T}_1$. Then, m refines π_1 into a plan $\pi_2 = (T_2, \alpha_2, \mathcal{C}_2, \mathcal{L}_2)$ and

$$
\begin{aligned}
T_2 &= (T_1 - \{t\}) \cup \mathit{subtasks}(m) \\
\alpha_2 &= \{(t', \alpha_1(t')) \mid t' \in T_1 \backslash \{t\}\} \cup \alpha \\
\mathcal{C}_2 &= \mathcal{C}_1 \cup \mathit{constraints}(m) \\
&\quad \cup \{c \mid \forall u \in \mathit{subtasks}(m)\ v_t^s \leq v_u^s\} \\
&\quad \cup \{c \mid \forall u \in \mathit{subtasks}(m)\ v_t^e \geq v_u^e\} \\
&\quad \cup \{v_t^s \leq v_t^e\} \\
\mathcal{L}_2 &= \mathcal{L}_1
\end{aligned}
$$

Consider now the case of the durative action refinement. Let $a = (\mathit{name}(a), \mathit{start}(a), \mathit{end}(a), \mathit{inv}(a), d)$ a durative action. To realize this refinement it is first necessary to slightly modify the definition of the two snap actions $\mathit{start}(a)$ and $\mathit{end}(a)$ to translate the invariant properties $\mathit{inv}(a)$ into the POCL logic. This modification consists in adding $\mathit{inv}(a)$ to the effects of $\mathit{start}(a)$ and to the preconditions of $\mathit{start}(a)$ and $\mathit{end}(a)$ (see Figure 1). It is now possible to express the invariant properties of a durative task by classical causal links between the effects of $\mathit{start}(a)$ and the preconditions of $\mathit{end}(a)$ in accordance with PDDL 2.1 semantics that constrains $\mathit{inv}(a)$ to be checked on the interval $]\mathit{start}(a), \mathit{end}(a)[$. More formally, suppose a refines a task t of a plan $\pi_1 = (T_1, \alpha_1, \mathcal{C}_1, \mathcal{L}_1)$ such that $t \in T_1$. Then, a refines π_1 into a plan $\pi_2 = (T_2, \alpha_2, \mathcal{C}_2, \mathcal{L}_2)$ and

$$
\begin{aligned}
T_2 &= (T_1 - \{t\}) \cup \{v_t^s, v_t^e\} \\
\alpha_2 &= \alpha_1 \cup \{(v_t^s, \mathit{start}(a)); (v_t^e, \mathit{end}(a))\} \\
\mathcal{C}_2 &= \mathcal{C}_1 \cup \{v_t^s < v_t^e, v_t^e - v_t^s = d\} \\
\mathcal{L}_2 &= \mathcal{L}_1 \cup \{l \mid \forall p \in \mathit{inv}(a)\ l = \langle \mathit{start}(a) \xrightarrow{p} \mathit{end}(a) \rangle\}
\end{aligned}
$$

Hierarchical Temporal Planning Event

The particularity of our approach is that it works by interleaving two steps to take advantage of the heuristics developed for non-temporal hierarchical planning. The first step is to compile a temporal problem in a non temporal one. To that end, we refine abstract tasks and durative actions into instantaneous non temporal actions called *snap actions*, and we search for a solution plan by guiding this search with non temporal POCL HTN heuristics, and by *checking* constraint consistency. The second step is to *find* time assignments matching the temporal constraints by using a simple CSP solver.

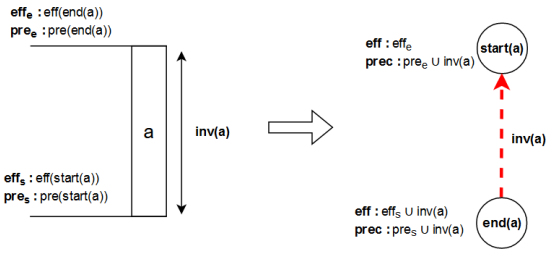


Figure 1: Our compilation of durative actions into snap actions. The invariant conditions $\mathit{inv}(a)$ are added to the preconditions of $\mathit{start}(a)$ and $\mathit{end}(a)$ and a causal link protecting $\mathit{inv}(a)$ (represented as a red dashed line) is added between the two snap actions.

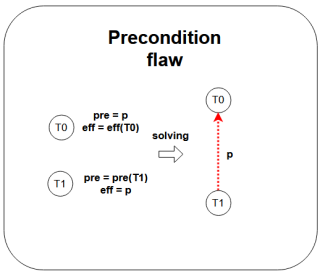
Algorithm 1: HTEP$(\mathcal{S}, \mathcal{T}, \mathcal{A}, \mathcal{M}, s_0, \pi_0, g)$

```
1  open ← {π0}
2  while open ≠ ∅ do
3      π ← non-deterministically select plan in open
4      flaws ← the set of flaws of π
5      if flaws = ∅ then
6          V ← search of time variable assignment of π
7          if V ≠ ∅ then return π and V
8      φ ← deterministically select a flaw in flaws
9      open ← open ∪ solveFlaw(ω, φ)
10 return Failure;
```

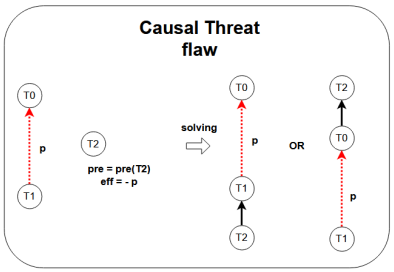
In this section, we present the search procedure implemented in our approach, called HTEP (Hierarchical Temporal Event Planner). We first give an overview of its search procedure based on hybrid planning (Bercher, Keen, and Biundo 2014), then we detail the particular way of handling specific temporal flaws of our approach, and we terminate by presenting the implemented heuristics.

Search Procedure

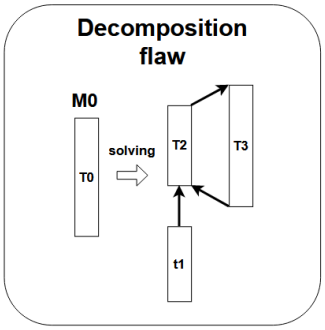
The HTEP search procedure is given in Algorithm 1. It takes as input a hierarchical temporal planning problem and returns a temporal partial plan. The procedure starts by compiling the tasks of the initial plan π_0 into snap actions (line 1). Recall that π_0 contains two special snap actions: t_0 with no precondition but with s_0 as effects and t_∞ with the goal g as precondition but no effects and $v_{t_0}^e < v_{t_\infty}^s$ in $\mathcal{C}$. Then, π_0 is added to the pending list of partial plans to explore *open* (line 2) and the main loop starts (line 3). At each iteration a plan π is non-deterministically selected in *open* (line 4). Then, the flaws of π are computed (line 5). If the plan has no flaws (line 6), HTEP searches an assignment of the time variables of π by using a CSP that matches its constraints. If such assignment exists (line 8), π is solution. Otherwise, one flaw is deterministically selected (line 9). Solving this flaw, generates a new set of partial temporal plans, which are added to the *open* list (line 10).



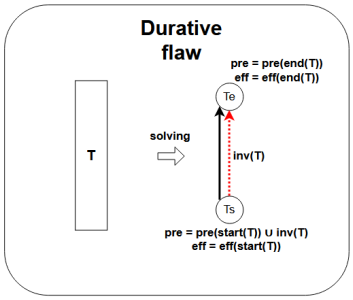
(a) A Precondition flaw is solved by adding a *causal link* from a provider event and the needer event.



(b) A Causal Threat flaw is solved by ordering the threatening event either before or after the causally linked events.



(c) Decomposition flaws are solved by applying a method, temporal constraints are applied over start and end point of tasks.



(d) Durative flaws are solved by compiling a durative task into snap actions.

Figure 2: The four types of flaws encountered by our planner and how to solve them. On every figure, black arrows represent ordering constraints and red dashed arrows the causal links.

Temporal Flaws

In addition to the classic flaws in POCL (precondition and causal threat flaws), HTEP requires the management of two specific types of flaws: the decomposition flaws that are repaired by applying a method and by decomposing an abstract temporal task in primitive temporal tasks, and the durative flaws that can be repaired by decomposing a durative action into snap actions. We detail each flaw encountered in the following:

- **Precondition flaw:** This flaw occurs for each precondition introduced in the partial plan. It is resolved by adding a *causal link* between a preceding snap action having as effect the required precondition. We denote a causal link c by : $t_1 \xrightarrow{p} t_2$. The resolution of this flaw is represented on Figure 2a.
- **Causal threat flaw:** There is a causal threat flaw for each causal link $c : t_1 \xrightarrow{p} t_2$ and each snap action a (with $p \in del(a)$) that is neither ordered before t_1 nor ordered after t_2. A causal threat is resolved either by adding a constraint: $a < t_1$ or $t_2 < a$ as represented on Figure 2b.
- **Decomposition flaw:** As in hybrid planning, decomposition flaw occurs for each abstract task still in the partial plan. They are solved by decomposing the abstract task with a method. Note that the decomposed tasks are still considered as durative and still need to be compiled into snap actions. This flaw is represented on Figure 2c.
- **Durative flaw:** A durative flaw occurs for each durative action still in the plan. In order to solve this flaw, the durative action is decomposed into two snap actions representing the start and the end points of the task. Finally, a causal link protecting the invariant preconditions is added between the two snap actions. These invariant preconditions are then added as preconditions of the start snap action.

Partial Order Planning heuristics

The HTEP search procedure relies on two selection functions. The first one performs a non deterministic choice over the set of partial temporal plans in the *open* list, and decides which partial plan to explore first (line 4). This function is called *plan selection heuristic* and greatly impact both the search performances (e.g. the time required to find a solution plan) but also the quality of the returned plan (e.g. the cost of the plan according to some optimization function). The second selection function, called *flaw selection heuristic*, selects (line 9) the flaw to be solved in the current partial plan to explore. Note that every flaw in the partial plan will eventually have to be solved in order to find a solution network. Hence, the *admissibility* of a POCL procedure only depends on the *plan selection* heuristic. However, the *order* in which the flaws are resolved, defined by the flaw selection heuristic, greatly impacts the search performances of the procedure. In the following we will present the plan selection and flaw heuristics implemented in HTEP.

Plan selection heuristics In the literature, there are two main categories of plan selection heuristics. The first type of heuristics has a POCL related approach which analyzes the flaws of a partial plan to infer a heuristic value. A well-known heuristics has been proposed in (Nguyen and Kambhampati 2001) and simply counts the number of open conditions to satisfy in the partial plan. This idea has been further refined in PANDA (Bercher, Keen, and Biundo 2014) where the use of TDG (Task Decomposition Graphs) allows to also estimate the number of open conditions that will be introduced by refining the plan. This led to two plan selection heuristics. The first one denoted h_{TC} computes the cardinality of the mandatory tasks that will appear in the partial plan decomposition. It is usually summed with the number of flaws remaining in the plan forming a heuristics denoted h_{F+TC}. The second TDG heuristic is denoted h_{MME} and estimates the number of modifications required to refine the partial plan into a solution one. The h_{MME} has been further refined to take into account the number of causal links already introduced into the plan by a heuristic denoted h_{TDGm}. Finally, FAPE (Bit-Monnot et al. 2020) has proposed a plan selection heuristics in their chronicle planner. This heuristics is also an estimation of the remaining effort required to obtain a solution from a partial plan. In the following we will denote this heuristic h_{FAPE}.

The second type of heuristics adapt heuristics from non hierarchical planning. The two most notable approaches have been proposed in (Nguyen and Kambhampati 2001) where an adaptation of the Fast Forward heuristic (Hoffmann and Nebel 2011) has been proposed. In addition, the ADD heuristics (Baier, Bacchus, and Mcilraith 2009) has been adapted in the VHPOP planner (Younes and Simmons 2003). These heuristics are very well suited for non-hierachical POCL planning with task insertion, which is not considered here.

In that regard, we decided to use the two TDG plan selection heuristics h_{F+TC} and h_{TDGm} used in PANDA and the plan selection heuristic presented in FAPE for our experimentation.

Flaw selection heuristics When it comes to flaw selection heuristics, the known strategies aim at reducing the branching factor of the search space by resolving the flaws with the fewest number of resolvers first. It is usually expressed as a priority list to follow. To our knowledge, there is no recent comparison between the current flaw selection heuristics. In that regard, we have chosen to implement in HTEP the flaw selection heuristics of PANDA (Bercher, Keen, and Biundo 2014) (which is the LCFR heuristics presented in (Joslin and Pollack 1994)) and the priority list presented in FAPE (Bit-Monnot et al. 2020). Note that the heuristic used in FAPE is a refinement of the LCFR heuristic: while LCFR prioritizes the flaws with the fewest resolvers, the FAPE flaw selection heuristics also prioritizes unrefined tasks and preconditions first.

Experimentation

In this section, we will compare our *Temporal Event* approach with a *Chronicle* approach. Both approaches have been been coded and tested on the same device. We have used as reference the chronicle planner described in (Bit-Monnot et al. 2020) as it is the current state-of-the-art of hierarchical temporal planning. As both algorithms use a CSP solver in their procedures, we will use the same CSP solver as well. All the benchmarks and code used to produce these results will be freely available for result reproduction.

Experimental Setup

We will compare the results on three indicators:

- **Solving time**: it represents the time spent to solve the problem (from instantiation to solution)
- **Makespan of the solution**: it represents the overall length of the plan, meaning the time between the initial state and the final task end point.

The results will be presented by scoring these two indicators with the IPC scoring metric.

In this paper we consider four configurations, which are presented below:

- $HTEP+h_{TDGm}$**:** in this first configuration, HTEP is used with the h_{TDGm} heuristics described in (Bercher et al. 2017) adapted to our temporal event representation.
- $HTEP + h_{F+PC}$**:** in this configuration, the Temporal Event Planner is used with the first TDG heuristics proposed in (Bercher, Keen, and Biundo 2014) associated with h_{F+PC} heuristics.
- $TE + h_{FAPE}$**:** in this configuration, HTEP is used with the plan selection heuristics used by the FAPE planner (Bit-Monnot et al. 2020).
- $Chr + h_{FAPE}$**:** this is the chronicle planner encoded to mimic the behavior of FAPE (Bit-Monnot et al. 2020). It uses both the plan selection and flaw selection heuristics described in (Bit-Monnot et al. 2020) and uses a chronicle representation.

Note that all configurations use the same Flaw Selection heuristics described in (Bit-Monnot et al. 2020).

We chose four temporal planning domains for our tests. These domains require different levels of concurrency to be solved. We categorized these problems according to the three Cushing concurrency categories (Cushing 2007):

- **Gripper:** this domain is a sequential one, it is the simplest domain with no temporal concurrency required. These problems are members of the first Cushing (Cushing 2007) category where all solutions are sequential.
- **Satellite:** this domain is also a sequential one but optimization over the *makespan* of the solution is possible (e.g. there are several sequential plans possible with different makespans). These problems are also members of the first Cushing category.
- **Rover:** in this domain, concurrency is possible although not required. The planner can either choose to find a simple non concurrent plan or a more efficient concurrent one. These problems are members of the second Cushing category where solutions can either be sequential or concurrent.

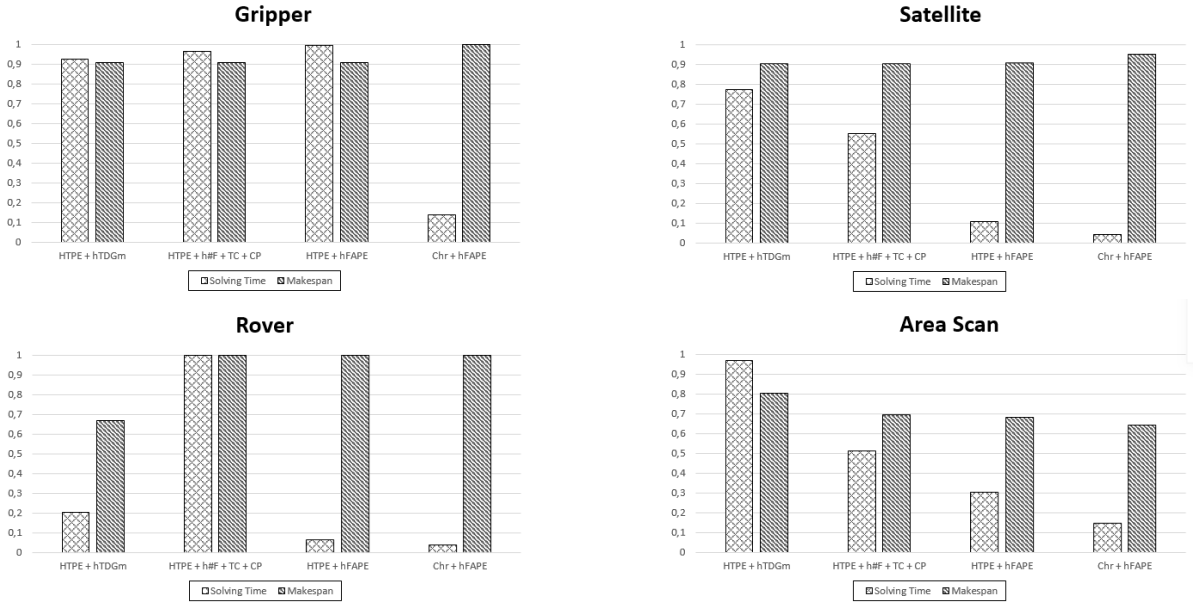


Figure 3: Results for the five configurations, with the IPC metric. Each diagram represents the results on one domain.

- **Area scan:** this domain models a set of heterogeneous devices aiming at cooperating in order to scan areas. The devices can either cooperate to reach their destination faster or act by their own, resulting in less effective plans. It presents both sequential and concurrent solution plans, with numerous makespan optimization possible.

We expressed all domains and problems in HDDL 2.1 language proposed by (D. Pellier and Bailon-Ruiz. 2023). All experiments were run on a single core of a Intel Core i7-9850H CPU, with a limit of 8GB of RAM over 600 seconds. The code and benchmarks will be made available if this paper is accepted.

Results

The results are presented on Figure 3.

We can see that there are some tendencies over all domains. On all domains and for both heuristics, the Temporal Event planner outperforms its chronicle counterpart with regards to the *Solving time* metric. HTEP handles non temporal constraints and flaws which leads to simpler plan refinements and flaws compared to a Chronicle planner. In our opinion, this is what explains the discrepancies between the $HTEP + h_{FAPE}$ and $Chr + h_{FAPE}$ configurations. In addition, the HTEP approach benefits from the use of classical HTN heuristics: the $HTEP + h_{TDGm}$ and $HTEP + h_{F+PC}$ configurations are the best performing ones on all domains. We can notice that the h_{F+PC} seems to be the best performing one on the Rover domain while h_{TDGm} is more efficient on the other domains. As explained in (Bercher, Keen, and Biundo 2014), the performance of each heuristics depends on the domain definition of the problem and a heuristics can be more or less informative depending on the domain. As a Chronicle approach should handles a larger set of constraints it should be able to eliminate partially refined plan sooner in the refinement process than HTEP. However this case does not happen often as many HTN domains describe the necessary temporal ordering through methods and task decomposition. The main sources of partial plan elimination are unsolvable non temporal flaws. Overall, it seems that the $HTEP+h_{TDGm}$ configuration is the most efficient one when it comes to the *Solving Time* metric.

Concerning the *Makespan*, we can see that the $Chr + h_{FAPE}$ configuration is the most efficient one. As this configuration handles the full temporal constraints, it can prioritize the best one in terms of makespan when two have equal heuristic value. This leads to higher quality plans in most domains. On the other hands, the HTEP configurations remain competitive with the Chronicle approach. The exception to this is displayed on the Rover domain where the $HTEP + h_{TDGm}$ configuration is the lowest one makespan wise. This heuristics aims at finding the solution with the fewest number of refinement required, regardless of the solution plan makespan. Note that if h_{TDG} is the best performing one on the Area Scan domain is due to the fact that some instances have not been solved by others configurations, thus increasing its score.

Overall, the configuration combining HTEP and the classical HTN heuristics seems to outperform the *Chronicle* approaches. This formalism allows for simpler constraints representation and managements. It also benefits from the hybrid planning heuristics and the HTN formalism which often provides the necessary ordering constraints to the planner.

Discussion

All along this paper, we used a compilation of temporal actions into snap actions by representing the invariant condition of a temporal action through a POCL causal link. The invariant are treated as precondition of the start snap action

and protected until the end snap task through this causal link (see Figure 1). This compilation implies a that HTEP can not solve temporal in the third Cushing's category: in temporal planning, invariant condition can not be seen as precondition for the whole temporal action. Instead, they should be seen as *postconditions* of the start snap action, meaning that invariant condition should be verified right *after* the execution of the start. This subtlety actually adds a new layer of complexity in planning as it allows to define *necessary concurrent actions*. This has been demonstrated and explained by Cushing in his three temporal hierarchical problem classes definition (Cushing 2007).

Conclusion

In this paper, we have presented an approach to represent and solve Temporal HTN problems by using Temporal Events. This approach relaxes the temporal problem in a simpler one, which allows to apply classical HTN search heuristics to it. We have compared this approach with the Chronicle one, which is the current state of the art in hierarchical temporal planning. We have shown that the Temporal Event approach outperforms it in terms of time spent to find a solution and is comparable to it when it comes to the quality of the solution plans. HTEP can still be improved by applying other classical HTN search techniques. In addition, we want to improve the compilation made in HTEP in order to allow it to solve temporal problems in the third Cushing category.

References

Abdulaziz, M.; and Koller, L. 2022. Formal Semantics and Formally Verified Validation for Temporal Planning. In *AAAI Conference on Artificial Intelligence*, 9635–9643.

Asuncion, M.; Castillo, L.; Fdez-Olivares, J.; Garcia-Perez, O.; Gonzalez-Munoz, A.; and Palao, F. 2005. SIADEX: An interactive knowledge-based planner for decision support in forest fire fighting. *AI Commun.*, 18: 257–268.

Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Nau, D. S.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *J. of Artif. Intell. Res.*, 20: 379–404.

Baier, J.; Bacchus, F.; and Mcilraith, S. 2009. A Heuristic Search Approach to Planning with Temporally Extended Preferences. *Artif. Intell.*, 173: 593–618.

Barreiro, J.; Boyce, M.; Do, M.; Frank, J.; Iatauro, M.; Kichkaylo, T.; Morris, P.; Ong, J.; Remolina, E.; Smith, T.; et al. 2012. EUROPA: A platform for AI planning, scheduling, constraint programming, and optimization. *4th International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*.

Bechon, P.; Barbier, M.; Infantes, G.; Lesire, C.; and Vidal, V. 2014. HiPOP: Hierarchical Partial-Order Planning. In *Starting AI Researchers' Symposium*.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In *IJCAI*, 480–488.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid Planning Heuristics Based on Task Decomposition Graphs. *Proceedings of the International Symposium on Combinatorial Search*.

Bit-Monnot, A.; Ghallab, M.; Ingrand, F.; and Smith, D. E. 2020. FAPE: a Constraint-based Planner for Generative and Hierarchical Temporal Planning. *CoRR*, 2010.13121.

Broxvall, M.; and Jonsson, P. 2003. Point algebras for temporal reasoning: Algorithms and complexity. *Artif. Intell.*, 149(2): 179–220.

Celorrio, S. J.; Jonsson, A.; and Palacios, H. 2015. Temporal Planning With Required Concurrency Using Classical Planning. In *ICAPS*, 129–137.

Cushing, W. 2007. Evaluating Temporal Planning Domains. *ICAPS*, 105–112.

D. Pellier, H. F., A. Albore; and Bailon-Ruiz., R. 2023. HDDL 2.1: Towards Defining an HTN Formalism with Time. In *6th ICAPS Workshop on Hierarchical Planning (HPlan 2023)*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal Constraint Networks. *Artif. Intell.*, 49(1-3): 61–95.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. of Artif. Intell. Res.*, 20: 61–124.

Goldman, R. 2006. Durative Planning in HTNs. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 382–385.

Hoffmann, J.; and Nebel, B. 2011. The FF Planning System: Fast Plan Generation Through Heuristic Search. *J. of Artif. Intell. Res.*, 14.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *AAAI Conference on Artificial Intelligence*, 9883–9891.

Joslin, D.; and Pollack, M. E. 1994. Least-Cost Flaw Repair: A Plan Refinement Strategy for Partial-Order Planning. In Hayes-Roth, B.; and Korf, R. E., eds., *NCAI*, 1004–1009.

Lallement, R.; de Silva, L.; and Alami, R. 2018. HATP: Hierarchical Agent-Based Task Planner. In *AAMAS*, 1823–1825.

Lemai, S. 2004. IXTET-EXEC: planning, plan repair and execution control with time and resource management.

Milot, A.; Chauveau, E.; Lacroix, S.; and Lesire, C. 2021. Solving Hierarchical Auctions with HTN Planning. In *ICAPS workshop on Hierarchical Planning*.

Nguyen, X.; and Kambhampati, S. 2001. Reviving Partial Order Planning. In *IJCAI*, 459–464.

Younes, H. L. S.; and Simmons, R. G. 2003. VHPOP: Versatile Heuristic Partial Order Planner. *J. of Artif. Intell. Res.*, 20: 405–430.

On the Computational Complexity of Plan Verification, (Bounded) Plan-Optimality Verification, and Bounded Plan Existence

Songtuan Lin[1], Conny Olz[2], Malte Helmert[3], Pascal Bercher[1]

[1] School of Computing, The Australian National University
[2] Institute of Artificial Intelligence, Ulm University
[3] Department of Mathematics and Computer Science, University of Basel
{songtuan.lin, pascal.bercher}@anu.edu.au, conny.olz@uni-ulm.de, malte.helmert@unibas.ch

Abstract

In this paper we study the computational complexity of several reasoning tasks centered at the bounded plan existence problem. We do this for standard classical planning and hierarchical task network (HTN) planning and each for the grounded and the lifted representation. Whereas bounded plan existence complexity is known for classical planning, it has not been studied yet for HTN planning. For plan verification, results were available for both formalisms except the lifted representation of HTN planning. We will thus present the lower bound and the upper bound of the complexity of plan verification in lifted HTN planning and provide novel insights into its grounded counterpart, in which we show that verification is not just **NP**-complete in the general case, but already for a severely restricted special case. Finally, we show the computational complexity concerning the optimality of a given plan, i.e., answering the question whether such a plan is optimal, and discuss its connection to the bounded plan existence problem.

Introduction

Automated planning is the task of finding a course of actions called a plan which achieves a certain goal. An immense effort has been devoted to studying the computational complexity of the plan existence problem in the context of both non-hierarchical (classical) planning (Erol, Nau, and Subrahmanian 1991; Bylander 1994; Helmert 2006; Bäckström and Jonsson 2011) and hierarchical planning (Erol, Hendler, and Nau 1996; Geier and Bercher 2011; Alford et al. 2014; Alford, Bercher, and Aha 2015) which is to decide whether a planning problem has a solution. In contrast, the number of research endeavors on the complexity of finding an *optimal* plan is relatively small. Despite that many approaches for finding optimal plans have been developed for both classical planning (Karpas and Domshlak 2009; Pommerening et al. 2014) and hierarchical planning (Bercher et al. 2017; Behnke, Höller, and Biundo 2019; Behnke and Speck 2021), the complexity results only exist in the classical setting but not in the hierarchical one.

We will discuss the complexity of several problems centered at the *bounded* plan existence problem (which is a standard way of framing the problem of finding an optimal solution as a decision problem). Our discussion starts with the plan verification problem, which serves as the basis for the investigation of the bounded plan existence problem, and ends up with the plan optimality verification problem and its extension, the bounded plan optimality verification problem. Plan optimality verification is to verify whether a plan is an optimal solution to a planning problem, and its bounded version is to check whether the length of a given plan is not far away from the length of an optimal one by some bound.

We will investigate some general properties of these problems and discuss their complexity results in the specific context of classical planning (Ghallab, Nau, and Traverso 2004) and *Hierarchical Task Network* (HTN) planning (Bercher, Alford, and Höller 2019), which is the most commonly used hierarchical planning (Ghallab, Nau, and Traverso 2004; Bercher, Alford, and Höller 2019) formalism. One important reason for discussing all these results, which are summarized in Tab. 1, is that they can serve as a reference for future research endeavors in related disciplines.

Concretely, for plan verification, although the complexity is well-developed for classical planning and *grounded* HTN planning (Behnke, Höller, and Biundo 2015), no investigations have been done for *lifted* HTN planning. Here, we will present the lower bound and the upper bound of the complexity of lifted HTN plan verification, which turns out to be significantly harder compared to its grounded counterpart.

For the bounded plan existence problem, we will discuss its complexity in terms of both the *encoding size* and the *magnitude* of the bound. For this, we follow the methodology by Bäckström and Jonsson (2011) which encodes the bound in *binary* and in *unary*, respectively. Lastly, we will discuss the connection between the bounded plan existence problem and the plan optimality verification problem and present the complexity results for the latter.

Background

We start by presenting the notations that will be used throughout the paper together with the planning formalisms on which the complexity results are developed.

Size of Objects Given an *arbitrary* object x, e.g., x can be a number, a problem instance, etc., we say that the size of x, written $\|x\|$, is the length of a binary string which encodes the object x. When studying the complexity of a problem, accounting for the size of the problem is crucial because the runtime of a certain algorithm (operation) for the problem

		Plan Verification	k-length Plan Existence		Plan Optimality Verification	Bounded Plan Optimality Verification	
			k in binary	k in unary		plan given	only plan length given
Classical	*ground*	In **P**	**PSPACE**-complete (Bylander 1994)	**NP**-complete (Bäckström and Jonsson 2011)	**coNP**-complete Prop. 3	**coNP**-complete Prop. 4	**PSPACE**-complete Prop. 5
	lifted	In **P**	**NEXPTIME**-complete (Erol, Nau, and Subrahmanian 1991)	**NP**-complete Thm. 4	**coNP**-complete Prop. 3	**coNP**-complete Prop. 4	**coNEXPTIME**-complete Prop. 5
Hierarchical	*ground*	**NP**-complete Prop. 2	**NEXPTIME**-complete Thm. 3	**NP**-complete Thm. 5	**coNP**-complete Prop. 3	**coNP**-complete Prop. 4	**coNEXPTIME**-complete Prop. 5
	lifted	**PSPACE**-hard Thm. 1, Cor. 1 In **NEXPTIME** Thm. 2	**NEXPTIME**-complete Thm. 3	**PSPACE**-hard Thm. 6 In **NEXPTIME** Thm. 6	**PSPACE**-hard Prop. 3 In **coNEXPTIME** Prop. 3	**PSPACE**-hard Prop. 4 In **coNEXPTIME** Prop. 4	**coNEXPTIME**-complete Prop. 5

Table 1: Summary of the complexity results and the respective theorems. Note that we demand here that a solution to an HTN planning problem is an *action sequence*, which is different from the standard definition of solutions where a solution is a primitive task network. The plan optimality verification problem and the bounded plan optimality verification problem with the plan given explicitly are semantically equivalent, and they are the *complement* of the bounded plan existence problem with the bound given in unary. The bounded plan optimality verification problem with only the plan length given is the complement of the bounded plan existence problem. As a special case, plan optimality verification with only the length of a plan being given is equivalent to setting the bound to zero in the bounded plan optimality verification (where the plan is also not given explicitly), and hence it is also the complement of the bounded plan existence problem (cf. Prop. 6). **NP**-completeness of plan verification in grounded HTN planning was proved (Behnke, Höller, and Biundo 2015). We show that it holds even in a severely restricted case.

is measured with respect to the problem size. Notably, the size of an object varies in how it is encoded, e.g., a number can be encoded either in binary or in unary, which can affect the complexity of the problem. As an example, the unary encoding of 5 is "11111", and its binary encoding is "101".

Grounded Classical Planning A *grounded* classical planning problem is a tuple $\Pi = (\mathcal{D}, s_I, g)$ where $\mathcal{D} = (F, \mathcal{A}, \alpha)$ is called the *domain* of Π. F is a (finite) set of *propositions*, $\mathcal{A}$ is a (finite) set of *action names* (or actions for short), and $\alpha : \mathcal{A} \to 2^F \times 2^F \times 2^F$ is a function mapping each action $a \in \mathcal{A}$ to its precondition, add list, and delete list, written $\alpha(a) = (\mathit{prec}(a), \mathit{add}(a), \mathit{del}(a))$. $s_I \in 2^F$ is the *initial state* of Π and $g \subseteq F$ the goal description.

Generally speaking, the objective of (grounded) classical planning is to find an action sequence which turns the *initial state* into another *state* where the goal description is satisfied. Formally, a state s in classical planning is a set of propositions, i.e., $s \in 2^F$. Applying an action $a \in \mathcal{A}$ in a state s will result in a new state s' with $s' = (s \setminus \mathit{del}(a)) \cup \mathit{add}(a)$. An action a is *applicable* in a state s if $\mathit{prec}(a) \subseteq s$. In other words, the precondition of a is satisfied in s. For convenience, we write $s \to_a s'$ to indicate that the action a is applicable in the state s, and the state s' is obtained by applying a in s. Further, given a state s and an action sequence $\pi = \langle a_1 \cdots a_n \rangle$ ($n \in \mathbb{N}$), we write $s \to_\pi^* s'$ for some state s' to indicate that s' is obtained by applying π in s, that is, there exists a state sequence $\langle s_0 \cdots s_n \rangle$ such that $s_0 = s$, $s_n = s'$, and for each $1 \leq i \leq n$, $s_{i-1} \to_{a_i} s_i$. Consequently, a solution to a (grounded) classical planning problem is an action sequence π such that $s_I \to_\pi^* s'$ for some state s' and $g \subseteq s'$.

Lifted Classical Planning The *lifted* classical planning formalism is an extension of the grounded one and is defined on the *alphabet of a first-order language* $\Sigma = (\mathcal{V}, \mathcal{O}, \mathcal{R})$ where $\mathcal{V}$ is a set of *variables*, $\mathcal{O}$ a set of *objects*, and $\mathcal{R}$ a set of *predicates*. A predicate $\mathbf{p} \in \mathcal{R}$ is of the form $\mathbf{p} = P(v_1, \cdots, v_n)$ for some $n \in \mathbb{N}$ where P is called the predicate's name, and $v_i \in \mathcal{V}$ for each $1 \leq i \leq n$. Substituting every variable in a predicate with an object is called *grounding* the predicate, and it is characterized by a *variable substitution function* $\varrho : \mathcal{V} \to \mathcal{O}$. More concretely, given a variable substitution function ϱ, grounding the predicate $\mathbf{p}$ according to ϱ results in the *grounded* predicate, written $\mathbf{p}[\![\varrho]\!]$, with $\mathbf{p}[\![\varrho]\!] = P(\varrho(v_1), \cdots, \varrho(v_n))$. In particular, a grounded predicated is equivalent to a proposition in the grounded classical planning formalism.

A lifted planning problem is again a tuple $\mathbf{\Pi} = (\mathcal{D}, s_I, g)$ with $\mathcal{D} = (\Sigma, \mathcal{A}, \alpha)$ being its domain. In the lifted setting, $\mathcal{A}$ is a set of *action schemas*. An action schema, $\mathbf{a} \in \mathcal{A}$, also consists of an action name and a tuple of variables, written $A(v_1, \cdots, v_n)$ ($n \in \mathbb{N}$) with A being the action name. α maps an action schema to its precondition, add list, and delete list, written $\alpha(\mathbf{a}) = (\mathit{prec}(\mathbf{a}), \mathit{add}(\mathbf{a}), \mathit{del}(\mathbf{a}))$, each of which is a set of *predicates* $P(v_{i_1}, \cdots, v_{i_j})$ such that $v_{i_r} \in \{v_1, \cdots, v_n\}$ for each $r \in \{1, \cdots, j\}$.

An action schema $\mathbf{a}$ can also be grounded into an *action* a in the grounded setting by a variable substitution function ϱ, written $a = \mathbf{a}[\![\varrho]\!]$. Notably, when grounding an action schema, all predicates in its precondition, add list, and delete list are grounded simultaneously by the same variable substitution function.

Lastly, s_I and g are two sets of *grounded* predicates (i.e., propositions) which are the initial state and the goal description of $\mathbf{\Pi}$, respectively. A solution to $\mathbf{\Pi}$ is an *action* sequence $\pi = \langle a_1 \cdots a_n \rangle$ such that $s_I \to_\pi^* s'$ for some state s' with $g \subseteq s'$, and for each a_i with $1 \leq i \leq n$, there exist an action schema $\mathbf{a} \in \mathcal{A}$ and a variable substitution function ϱ such that $a_i = \mathbf{a}[\![\varrho]\!]$.

Notably, one can obtain a *grounded* planning problem Π from a lifted one $\mathbf{\Pi}$ by grounding *every* predicate and action schema with *all* possible variable substitution functions, and the problem Π produced in such a way has the same solution set as $\mathbf{\Pi}$. One important remark is that $\|\Pi\|$ is exponential in $\|\mathbf{\Pi}\|$, that is, $\|\Pi\| = O(2^{\|\mathbf{\Pi}\|^q})$ for some constant $q \in \mathbb{N}$.

Grounded HTN Planning We now reproduce the formalism of the *grounded* Hierarchical Task Network (HTN)

planning (Bercher, Alford, and Höller 2019). A grounded HTN planning problem Π is a tuple $(\mathcal{D}, s_I, tn_I, g)$ with $\mathcal{D} = (F, \mathcal{A}, \mathcal{C}, \mathcal{M}, \alpha)$ being its domain. A grounded HTN planning problem is an extension of a grounded classical one in the sense that F, $\mathcal{A}$, α, s_I, and g are defined in the same way as their counterparts in the classical setting. An action $a \in \mathcal{A}$ in HTN planning is also called a *primitive task*. Two components, $\mathcal{C}$ and $\mathcal{M}$, which are not in the classical formalism, are the set of *compound tasks* and of *methods*, respectively. A method $(c, tn) \in \mathcal{M}$ *decomposes* a compound task $c \in \mathcal{C}$ into a so-called *task network* tn, which is essentially a partial order *multiset* of primitive and compound tasks. Formally, a task network tn is a triple $(T, \prec, \gamma)$ where T is a set of *identifiers*, $\prec \subseteq T \times T$ is a partial order defined over T, and $\gamma : T \to \mathcal{A} \cup \mathcal{C}$ is a function that maps each identifier to a task. Two task networks, $tn = (T, \prec, \gamma)$ and $tn' = (T', \prec', \gamma')$, are said to be *isomorphic*, written $tn \cong tn'$, if there exists a *bijective* mapping $\varphi : T \to T'$ such that $\gamma(t) = \gamma'(\varphi(t))$ for any $t \in T$, and for any $t, t' \in T$, $(t, t') \in \prec$ *iff* $(\varphi(t), \varphi(t')) \in \prec'$. The last component tn_I in Π is the initial task network.

The notion of decomposing a compound task can also be extended to decomposing a task network. A task network $tn = (T, \prec, \gamma)$ is decomposed into another one $tn' = (T', \prec', \gamma')$ by some method $m = (c, tn^\dagger)$, written $tn \Rightarrow_m tn'$, if there exists an identifier $t \in T$ and a task network $tn^* = (T^*, \prec^*, \gamma^*)$ with $tn^* \cong tn^\dagger$ such that 1) $T^* \cap T = \emptyset$, 2) $\gamma(t) = c$, 3) $T' = (T \setminus \{t\}) \cup T^*$, 4) $\gamma' = (\gamma \setminus \{(t, c)\}) \cup \gamma^*$, and 5) $\prec' = (\prec \setminus \prec_t) \cup \prec^* \cup \prec_\delta$ with $\prec_t = \{(t', t) \mid (t', t) \in \prec\} \cup \{(t, t') \mid (t, t') \in \prec\}$, i.e., $\prec_t$ is the set of all ordering constraints in tn that are associated with t, and $\prec_\delta = \{(t_1, t_2) \mid t_2 \in T^*, (t_1, t) \in \prec\} \cup \{(t_2, t_1) \mid t_2 \in T^*, (t, t_1) \in \prec\}$, i.e., $\prec_\delta$ specifies the position of tn^* in tn' with respect to the task t replaced by it. Further, let tn and tn' be two task networks and $\overline{m}$ a sequence of methods. We use $tn \Rightarrow^*_{\overline{m}} tn'$ to indicate that tn' is obtained from tn by applying $\overline{m}$.

Like classical planning, (grounded) HTN planning is also to find an action sequence (i.e., a plan) which turns s_I into a state satisfying g. However, in HTN planning, such a plan must be obtained from the initial task network by decompositions. Concretely, a plan π is a solution to an HTN planning problem Π if $s_I \Rightarrow^*_\pi s$ with $g \subseteq s$ for some state s, and there exists a task network $tn = (T, \prec, \gamma)$ such that $tn_I \Rightarrow^*_{\overline{m}} tn$ for some method sequence $\overline{m}$, and tn has a *linearization* $\overline{tn}$ that forms π. A linearization $\overline{tn} = \langle t_1 \cdots t_{|T|} \rangle$ of tn is a total order of T which respects $\prec$, and by $\overline{tn}$ forming π, we mean that $\pi = \langle \gamma(t_1) \cdots \gamma(t_{|T|}) \rangle$. For convenience, we use $\gamma(\overline{tn})$ to denote the task sequence formed by $\overline{tn}$. Please note that there is a minor difference compared to standard HTN literature (Bercher, Alford, and Höller 2019; Erol, Hendler, and Nau 1996) in our solution definition. In our definition, a solution is an *action sequence*, which we argue makes the most sense. In standard literature, a solution is a primitive *task network* having an executable linearization.

Lifted HTN Planning A lifted HTN planning problem is a tuple $\mathbf{\Pi} = (\mathcal{D}, s_I, tn_I, g)$ with $\mathcal{D} = (\Sigma, \mathcal{A}, \mathcal{C}, \mathcal{M}, \alpha)$ being its domain where $\Sigma = (\mathcal{V}, \mathcal{O}, \mathcal{R})$, $\mathcal{A}$, and α are defined in the same way as that in lifted classical planning. Every action schema is also called a *primitive task schema*. $\mathcal{C}$ is now a set of *compound task schemas* and $\mathcal{M}$ a set of *method schemas*. A compound task schema $\mathbf{c} \in \mathcal{C}$ is simply a compound task name together with a tuple of variables. A method schema $\mathbf{m}$ is a tuple $(\mathbf{c}, \mathbf{tn})$ where $\mathbf{c}$ is a compound task schema and $\mathbf{tn}$ a *task network schema*. A task network schema is again a tuple $(T, \prec, \gamma)$ where T and $\prec$ are identical to those in a *grounded* task network, and γ maps each identifier to a *task schema*.

A task, task network, or method schema $\mathbf{x}$ can again be grounded by some variable substitution function $\varrho : \mathcal{V} \to \mathcal{O}$, written $\mathbf{x}[\![\varrho]\!]$. When grounding a task network schema $\mathbf{tn}$ with a substitution function ϱ, all task schemas in $\mathbf{tn}$ are grounded simultaneously by ϱ, and for any method schema $\mathbf{m} = (\mathbf{c}, \mathbf{tn})$ with $\mathbf{m}[\![\varrho]\!] = (\mathbf{c}[\![\varrho]\!], \mathbf{tn}[\![\varrho]\!])$. A grounded task schema and a grounded method schema are equivalent to a task and a method in the grounded setting, respectively.

s_I and g are again the initial state and the goal description consisting of *propositions*, and tn_I is the *grounded* initial task network. An action sequence π is a solution to a lifted HTN planning problem if $s_I \to^*_\pi s$ for some state s with $g \subseteq s$, and there exists a *grounded* method sequence $\overline{m} = \langle m_1 \cdots m_n \rangle$, $n \in \mathbb{N}$, such that for each $1 \leq i \leq n$, there exists a method schema $\mathbf{m} \in \mathcal{M}$ with $\mathbf{m}[\![\varrho]\!] = m_i$ for some ϱ, and $tn_I \Rightarrow^*_{\overline{m}} tn$ for some *primitive grounded* task network tn which possesses a linearization forming π.

Similar to lifted classical planning, one could also ground a lifted HTN planning problem without changing its solution set, and the size of the grounded problem is again exponential in that of the lifted one.

Proposition 1. *Let $\mathbf{\Pi}$ be a lifted (classical or hierarchical) planning problem and Π its grounded counterpart. Then it holds that $\|\Pi\| = O(2^{\|\mathbf{\Pi}\|^q})$ for some constant $q \in \mathbb{N}$.*

Plan Verification

Having presented all planning formalisms involved in this paper, we move on to discuss the complexity results for the plan verification problem, which is to decide, given a planning problem and a plan, whether the plan is a solution to the planning problem.

The complexity results for classical planning are obvious. In the grounded setting, a plan can clearly be validated in polynomial time by checking whether it is executable and satisfies all goals. This is well-known and exploited by verifiers like VAL (Howey, Long, and Fox 2004). Given a ground plan but a lifted problem description, the problem gets *slightly* more complicated because for each action in the plan we need to check whether it can be created by grounding some lifted action schema. This can easily be checked in polynomial time (just match constants to the respective variables). As a result, the plan verification problem for both grounded and lifted classical planning is in **P**.

In contrast, the plan verification problem in HTN planning is more computationally expensive. Previous works have already shown that it is already **NP**-complete in the *grounded* setting (Behnke, Höller, and Biundo 2015; Bercher, Lin, and Alford 2022). Those investigations rely on the standard def-

inition of solutions being task networks that possess some executable linearization, whereas we define such a linearization as the solution itself. In fact, even with our solution criteria, HTN plan verification is still **NP**-complete in grounded HTN planning (Behnke, Höller, and Biundo 2015, Thm. 2).

The existing hardness proof (Behnke, Höller, and Biundo 2015) for the verification problem (in the context of our definition of solutions) relies on finding a decomposition hierarchy, i.e., hardness of the problem adheres to decomposition that results in the given plan. We can further improve this result by showing that **NP**-hardness already holds even if the initial task network is *primitive*. This follows trivially from the fact that it is **NP**-complete to decide whether an action sequence is a linearization of a partial order task network (Lin and Bercher 2023).

Proposition 2. *The grounded HTN plan verification problem is **NP**-complete. This holds even in the special case where the initial task network of the given planning problem is primitive.*

Notably, as a special case, the plan verification problem in the context of *total order* (TO) HTN planning is polytime decidable (Behnke, Höller, and Biundo 2015). A TOHTN planning problem is such that the initial task network is totally ordered, and every method decomposes a compound task into a total order task network as well. Solving a TOHTN planning problem is computationally cheaper than solving a partial order one, and many theoretical investigations into properties of TOHTN planning have been made which have great potential to be utilized to solve TO problems more efficiently (Olz, Biundo, and Bercher 2021). The poly-time decidability of TOHTN plan verification holds because a (grounded) TOHTN planning problem is essentially equivalent to a context-free grammar (CFG) (Höller et al. 2014), and hence, the plan verification problem is equivalent to the parsing problem in TOHTN planning. Bearing this connection, many efficient TOHTN plan verifiers (Barták et al. 2021; Lin et al. 2023) have been developed by exploiting CFG parsers.

Now we extend our investigation from the grounded setting to the lifted one. Note that in the lifted setting, the plan to be verified is still grounded, but the planning problem is represented in the lifted way. Unlike the case in classical planning, hardness of the plan verification problem increases dramatically in lifted HTN planning. Concretely, we will show that plan verification is already **PSPACE**-hard even for lifted TOHTN planning.

Theorem 1. *The plan verification problem in lifted TOHTN planning is **PSPACE**-hard.*

Proof. We reduce from the plan existence problem in *grounded* classical planning. Suppose $\Pi = (\mathcal{D}, s_I, g)$ with $\mathcal{D} = (F, \mathcal{A}, \alpha)$ is a grounded classical planning problem. For convenience, we assume that, without loss of generality, $F = \{p_1, \cdots, p_n\}$ with $n \in \mathbb{N}$ and $g = \{p_{i_1}, \cdots, p_{i_j}\}$. The lifted HTN planning problem we are to construct has only two objects, namely, 0 and 1. At the central of the reduction is the compound task schema $\mathbf{c}$ which is of the form

$$\mathbf{c} = \texttt{State}(x_1, \cdots, x_n, v_0, v_1)$$

with $n = |F|$. Each x_i with $1 \leq i \leq n$ represents the corresponding $p_i \in F$. Thus, a *grounded* version of the schema $\mathbf{c}$ encodes a *state*. Our construction will ensure that v_0 is *always* grounded to 0 and v_1 to 1 in decomposition, which is useful for simulating state transitions (and which can be done by the construction of the initial task network). More concretely, we will construct method schemas that decompose $\mathbf{c}$ to encode actions (in the given classical problem). For each action $a \in \mathcal{A}$, we construct a method schema $\mathbf{m}_a$ which decomposes the task schema

$$\texttt{State}(x'_1, \cdots, x'_n, v_0, v_1)$$

into another one $\texttt{State}(x^*_1, \cdots, x^*_n, v_0, v_1)$ such that for all $1 \leq i \leq n$, $x'_i = v_1$ if $p_i \in \mathit{prec}(a)$, $x^*_i = v_0$ if $p_i \in \mathit{del}(a)$, $x^*_i = v_1$ if $p_i \in \mathit{add}(a)$, and $x'_i = x^*_i$ if none of the previous holds. Intuitively, all x'_i's with $x'_i = v_1$ together enforce that the precondition of a must hold (because we will ensure that v_1 can only be grounded to 1), and similarly, those x^*_i's with $x^*_i = v_1$ (*resp.* $x^*_i = v_0$) enforce that the respective propositions should be added (*resp.* deleted).

As an example for the construction, consider a grounded classical problem which has three propositions $\{p_1, p_2, p_3\}$ and three actions $\{a_1, a_2, a_3\}$. The precondition and effects of each action are depicted in the most left column of Fig. 1 (inside the box labeled with construction). Those on the left side of an action are preconditions, and those on the right are effects. Each effect with a negation symbol in front of it (e.g., $\neg p_1$ in the action a_1) is in the delete list of the respective action, otherwise it is in the add list. On the right side of each action is the corresponding method schema that encodes it. For instance, the method schema with respect to a_1 decomposes the task schema

$$\texttt{State}(v_1, x_2, x_3, v_0, v_1)$$

into $\texttt{State}(v_0, x_2, v_1, v_0, v_1)$. The first v_1 in the decomposed task schema is changed to v_0 because the proposition p_1 is in the delete list of a_1, and x_3 becomes v_1 because p_3 is in the add list. x_2 is unchanged because the execution of a_2 will not affect p_2.

Having simulated each action, we now encode the initial state s_I of the classical problem and enforce that the parameters v_0 and v_1 of $\mathbf{c}$ can only be grounded to 0 and 1 in decomposition, respectively. This is done by the construction of the initial task network of the HTN problem, which consists solely of one *grounded* compound task:

$$\texttt{State}(y_1, \cdots, y_n, 0, 1)$$

where for each i with $1 \leq i \leq n$, $y_i = 1$ if the respective p_i is in s_I, otherwise, $y_i = 0$. By letting $v_0 = 0$ and $v_1 = 1$ in the initial task, we enforce that the values of those two variables cannot be changed in decomposition, because in each method schema we construct, the variables v_0 and v_1 are always inherited down from the task schema to be decomposed to the subtask schema. Hence, the correctness of our construction for simulating executions of actions follows.

Recall the classical problem presented in Fig. 1, the right side of the figure illustrates how the decomposition hierarchy simulates the actions' executions, using our construction of method schemas. Concretely, assume that the initial state

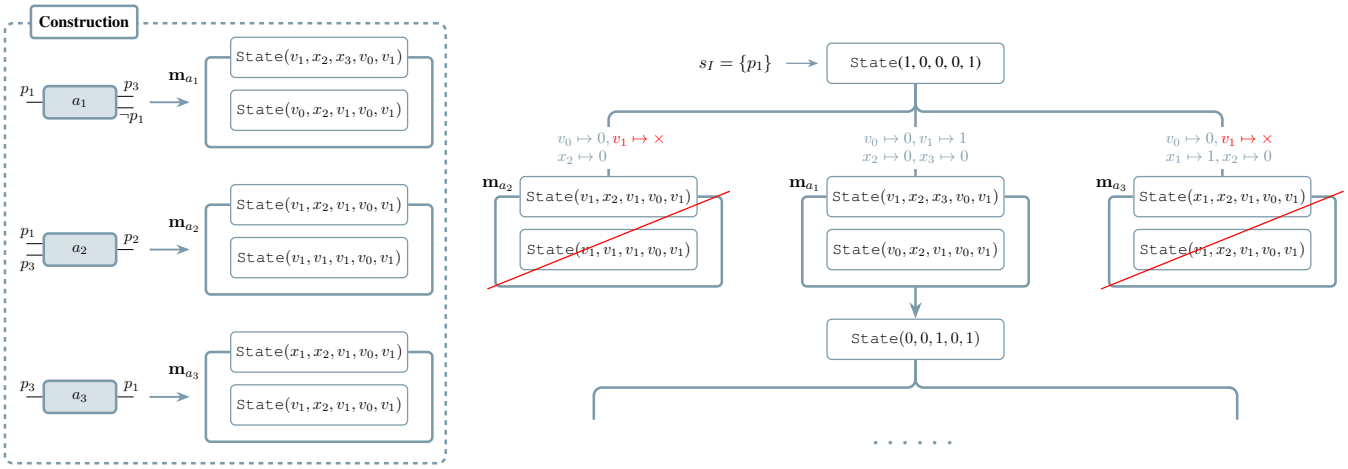


Figure 1: An example of using a decomposition hierarchy to simulate actions' executions in a classical planning problem. The left side shows how each action is encoded by a method schema, and the right side shows the decomposition.

of the classical problem is $\{p_1\}$. This results in the *grounded* initial compound task:

$$\texttt{State}(1, 0, 0, 0, 1)$$

because only p_1 is true in the state. a_1 is the only action that is applicable in the initial state. As a consequence, only the method schema $\mathbf{m}_{a_1}$ has a corresponding grounded version that can decompose the initial compound task. Specifically, for the action a_2, since it requires p_3 which is not in the initial state, it leads to a contradiction that v_1 should be grounded to both 0 and 1 simultaneously. The similar situation also happens to a_3. The decomposition of the initial compound task results in a new compound task $\texttt{State}(0, 0, 1, 0, 1)$ which encodes the state obtained by applying a_1 in the initial state.

Lastly, we will encode the goal description and the criterion that the goal must be satisfied. To this end, we first construct a method schema which decomposes the task schema $\texttt{State}(x_1, \cdots, x_n, v_0, v_1)$ into a *sequence* $\langle \mathbf{c}'_1 \cdots \mathbf{c}'_n \rangle$ of *compound* task schemas with

$$\mathbf{c}'_i = \texttt{P}_i(x_i)$$

for each $1 \leq i \leq n$. The compound task schema $\texttt{P}_i(x)$ can either be decomposed into an *action* schema

$$\mathbf{a}_i = \texttt{ValueP}_i(x)$$

or into an empty task network. The precondition, positive effects, and negative effects of $\mathbf{a}_i$ are all empty. We can view a grounded version of $\mathbf{a}_i$ as an assertion of the value of the proposition p_i in a state. If $\mathbf{a}_i$ is grounded by letting $x = 1$, it means that p_i holds in the respective state, and vice versa.

The plan to be verified should be $\langle a_{i_1}, \cdots, a_{i_j} \rangle$ where $a_{i_k} = \texttt{ValueP}_{i_k}(1)$ for each $1 \leq k \leq j$ (recall that each p_{i_k} is a proposition in the goal). This is to say that for each proposition in the goal, its truth value must be asserted. Since each task schema $\texttt{P}_i(x)$ can be decomposed into an empty task network no matter what object the variable x is grounded to, we can ensure that any action that is not in the constructed plan can be easily eliminated. Thus, the classical planning problem has a solution *iff* the constructed plan is a solution to the HTN planning problem. □

A by product of the presented proof is that it shows the expressive power of a decomposition hierarchy, i.e., a decomposition hierarchy can carry out certain semantics. For instance, Fig. 1 shows how the semantics of actions is encoded by a decomposition hierarchy. Thus, we also believe that the proof here can serve as a counter-argument for the *incorrect* commonsense that decomposition hierarchies in hierarchical planning can only serve as a guidance for finding plans but do not carry any information (semantics).

As a simple corollary of Thm. 1, **PSPACE**-hardness holds as well in general lifted HTN planning.

Corollary 1. *The plan verification problem in lifted HTN planning is **PSPACE**-hard.*

For membership, one can observe that the lifted HTN plan verification problem is in **NEXPTIME**. This is because for any lifted HTN planning problem $\mathbf{\Pi}$ and a plan π, we can first ground $\mathbf{\Pi}$ into a grounded one Π in exponential time according to Prop. 1. Since the grounded HTN plan verification problem is in **NP**, we can non-deterministically verify whether π is a solution to Π in polynomial time with respect to $\|\Pi\|$ and $\|\pi\|$. It thus follows that whether π is a solution to $\mathbf{\Pi}$ can be checked non-deterministically in exponential time with respect to $\|\mathbf{\Pi}\|$.

Theorem 2. *The plan verification problem in lifted HTN planning is **PSPACE**-hard and is in **NEXPTIME**.*

In fact, one can recognize that plan verification for lifted TOHTN planning is actually in **EXPTIME**. This is because the grounded TOHTN plan verification problem is in **P**, and hence, after grounding a lifted problem $\mathbf{\Pi}$ into a grounded one Π, we can verify *deterministically* whether a given plan is a solution in polynomial time with respect to $\|\Pi\|$, which is exponential time with respect to $\|\mathbf{\Pi}\|$, e.g., by using the CYK algorithm for TOHTN problems (Lin et al. 2023).

Bounded Plan Existence

We now move on to discuss the complexity of the bounded (k-length) plan existence problem, which is to decide, given a planning problem and a $k \in \mathbb{N}$, whether there is a solution plan π to the problem of length up to k. We start with some general properties of this problem and then discuss its complexity in specific planning formalisms.

One insight into this problem is that it can always be decided non-deterministically by a two-steps procedure independent of any planning formalism, namely, we can first guess a plan of length up to the bound k and then verify whether this plan is a solution to the given planning problem. Further, notice that since the bound k is normally encoded in *binary*, guessing a plan that is bounded in length by k would thus require exponential time complexity. On top of this observation, we can obtain two properties which assert **NEXPTIME**-membership of the bounded plan existence problem for a planning formalism if these two properties hold in that formalism. Concretely, the two properties are as follows:

1) An action can be encoded in polynomial bits with respect to the size of the planning problem. This thus implies that guessing a plan of length up to the bound k can be done in time $O(2^{\|k\|^q})$ for some constant $q \in \mathbb{N}$.
2) Verifying whether a plan is a solution to the planning problem can be done non-deterministically in exponential time with respect to the encoding size of the planning problem and of the plan, i.e., the plan verification problem is in **NEXPTIME**.

These two properties together ensure that guessing and verifying a plan can be done in exponential time.

As a result, the bounded plan existence problem in both classical and HTN planning, including both the grounded and the lifted setting, is in **NEXPTIME**. The problem is actually **PSPACE**-complete (Erol, Nau, and Subrahmanian 1991; Bylander 1994) in *grounded* classical planning (note that this is *not* a contradiction because **PSPACE** is a subset of **NEXPTIME**), making it as hard as its unbounded version (Bylander 1994). **NEXPTIME**-completeness of the problem in lifted classical planning has also been proved by Erol, Nau, and Subrahmanian (1991), and its unbounded counterpart in the lifted setting is **EXPSPACE**-complete. For HTN planning, we will show that **NEXPTIME**-hardness holds as well in both the grounded and lifted HTN planning.

Theorem 3. *The k-length (bounded) plan existence problem for both grounded and lifted HTN planning is **NEXPTIME**-complete.*

Proof. Membership follows from Prop. 2 and Thm. 2. For hardness, we first show that the problem is **NEXPTIME**-hard in the grounded setting. We reduce from the *grounded acyclic* HTN plan existence problem. The basis for such a reduction is the fact shown by Behnke et al. (2016) that for any acyclic HTN planning problem, the length of a solution is bounded by an exponential number k^* with

$$k^* = \left(\max_{(c,(T,\prec,\gamma)) \in \mathcal{M}} |T| \right)^{|\mathcal{A}|}$$

Hence, by letting $k = k^*$, deciding whether an acyclic HTN planning has a solution is equivalent to deciding whether that acyclic HTN problem has a solution bounded in size by k. **NEXPTIME**-hardness of the bounded plan existence problem in grounded HTN planning follows immediately. Since a grounded HTN problem can be viewed as a special case of a lifted one, it follows that **NEXPTIME**-hardness holds as well in lifted HTN planning. □

Encoding the Bound in Unary Our discussion about the k-length plan existence problem so far is restricted to the case where the bound k is given in *binary*. That is, the encoding size of k is growing exponentially while its magnitude is growing polynomially. This however might contradict the intention of giving such a bound. More concretely, in practice, when a user uses a planner to find a plan of length up to a certain bound, the user is actually concerned with the *magnitude* of this bound but not the encoding size.

Bearing this scenario, Bäckström and Jonsson (2011) investigated the k-length plan existence problem from a different aspect where they developed its complexity with respect to the magnitude of the bound. This is done by assuming that the bound is encoded in *unary*. The authors studied this for *finite functional planning* (FFP) and proved its **NP**-completeness. They further justified that a *grounded* classical planning problem can be reduced to an FFP problem in poly-time (Bäckström and Jonsson 2011, Prop. 1) (note that this does *not* hold for the lifted formalism), and hence, **NP**-completeness also holds in grounded classical planning.

We now extend the result by Bäckström and Jonsson to lifted classical planning and HTN planning. Notice first that when k is given in unary, we can again identify two properties of a planning formalism which assert **NP**-membership and which are similar to the previous two that assert **NEXPTIME**-membership. Concretely, for any planning formalism, its k-length plan existence problem with k given in unary is in **NP** if the formalism holds the following two properties: 1) a plan step can be encoded in polynomial bits, and 2) the plan verification problem for the formalism is in **NP**. This is because the first property now implies that guessing a plan of length up to the bound k can be done in polynomial time with respect to the size of the planning problem and k, due to the unary encoding of k. Consequently, the time complexity of the entire guess-and-verify procedure can be done in polynomial time.

Hence, **NP**-membership in lifted classical planning and grounded HTN planning follows immediately because these formalisms preserve the two properties. In particular, **NP**-hardness in lifted classical planning holds as well due to **NP**-hardness in the grounded setting.

Theorem 4. *The k-length plan existence problem for lifted classical planning is **NP**-complete if k is encoded in unary.*

Next we will prove **NP**-hardness for grounded HTN planning (and hence **NP**-completeness). Our proof relies on the reduction proposed by Erol, Hendler, and Nau (Erol, Hendler, and Nau 1996) from a grounded classical planning problem to a *regular* TOHTN problem[1] (which thus shows

[1] A regular HTN problem is such that a compound task can only occur at the last place in a method, i.e., all other tasks are ordered before it.

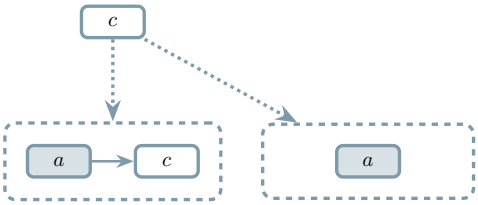


Figure 2: Simulating a grounded classical planning problem.

that total order regular HTN problems are **PSPACE**-hard).

Theorem 5. *The k-length plan existence problem for grounded HTN planning is **NP**-complete, if k is encoded in unary.*

Proof. Membership for both the grounded and lifted formalisms has been obtained. We will first show **NP**-hardness for the grounded setting, which thus implies **NP**-hardness for the lifted one.

We reduce from the k-length plan existence problem for *grounded classical* planning with k given in unary. Given a grounded classical planning problem $\Pi = (\mathcal{D}, s_I, g)$ with $\mathcal{D} = (F, \mathcal{A}, \alpha)$ and a $k \in \mathbb{N}$ in unary, we first simulate the classical problem by an HTN problem, using the construction proposed by Erol, Hendler, and Nau (Erol, Hendler, and Nau 1996). The HTN planning problem has the same action set as the classical one and solely one compound task c (which thus also serves as the initial task network). For each $a \in \mathcal{A}$, we construct two methods m_1 and m_2 with $m_1 = (c, (\{t\}, \emptyset, \{(t, a)\}))$ and

$$m_2 = (c, (\{t_1, t_2\}, \{(t_1, t_2)\}, \{(t_1, a), (t_2, c)\}))$$

An illustration of this construction is shown in Fig. 2. It simulates selections of actions in finding a solution to the classical planning problem. The initial state and the goal of the HTN problem are also identical to the classical one. The reduction can then be done by copying k (in unary). □

Unfortunately, **NP**-membership does *not* hold in lifted HTN planning because the plan verification problem in lifted HTN planning is **PSPACE**-hard.

Theorem 6. *The k-length plan existence problem in lifted HTN planning with k given in unary is **PSPACE**-hard and is in **NEXPTIME**.*

Proof. Membership: Membership can be obtained by first guessing a plan of length up to k in poly-time (because k is encoded in unary) and then verifying non-deterministically whether it is a solution to a lifted HTN problem in exponential time (cf. Thm. 2).

Hardness: We again reduce from the grounded classical plan existence problem. The construction of the lifted HTN planning problem is identical to the one presented in the proof for Thm. 1 except few changes. More specifically, for each action schema $\texttt{ValueP}_i(x)$, we construct a predicate $\texttt{Prop}_i(x)$ as its positive effect, and the goal of the lifted HTN problem is $\{\texttt{Prop}_{i_j}(1) \mid p_{i_j} \in g\}$ where g is the goal description of the grounded classical planning problem. These modifications thus encode the solution criteria for the grounded classical planning problem and can replace the constructed plan that is to be verified in the proof for Thm. 1. Lastly, notice that any solution plan to the constructed lifted HTN planning problem is of length at most $|F|$ (where F is the proposition set of the given grounded classical planning problem). We can simply let $k = |F|$, and hence, by construction, the classical planning problem has a solution *iff* the constructed lifted HTN planning problem has one which is of length smaller or equal to k. □

Verification of Plan Optimality

Lastly, we discuss the problem of plan optimality verification, which is to decide, given a planning problem and a plan, whether there exist *no* other solution plans of length smaller than that of the given one. Many tasks of great importance are centered on plan optimality verification, for instance, the task of *model reconciliation* and of *plan post-optimization*. The former one is to change a planning problem's domain with the least number of changes so as to turn a plan into an *optimal* solution, and this task is Σ_2^p-complete (Sreedharan, Bercher, and Kambhampati 2022). The latter one is concerned with whether a plan can be further optimized by removing some redundant actions in it, and it is **NP**-complete in both classical planning (Fink and Yang 1992) and POCL planning (Olz and Bercher 2019).

Despite that the complexity results for those related problems are well-developed, the problem of plan optimality verification itself has not yet received particular attention. One remark of great importance is that the plan optimality verification problem can be viewed as a complement of the bounded plan existence problem with the bound given in unary. The reason is that each action in the plan π provided in the plan optimality verification problem does not matter. What we are really concerned with is the *length* $|\pi|$ of that plan. Thus, asking whether the plan π is an optimal one is identical to asking whether there exist no solution plans of length smaller or equal to $|\pi| - 1$ with $|\pi| - 1$ encoded in unary, which is a complement of the bounded plan existence problem with the bound given in unary.

As a result, the complexity of the plan optimality verification problem for a specific planning formalism is naturally the complement of that of the bounded plan existence problem with the bound given in unary for that formalism.

Proposition 3. *The plan optimality verification problem for classical planning, including both the grounded and lifted representations, and grounded HTN planning is in **coNP**-complete, and it is in **coNEXPTIME** and is **PSPACE**-hard for lifted HTN planning.*

PSPACE-hardness in lifted HTN planning is due to the fact that **PSPACE** = **coPSPACE** (Arora and Barak 2009).

Since optimality is often diametral to efficiency, and finding a strict optimal solution is time-consuming in practice, it is quite often the case that a solution whose length lies in an acceptable range of the length of an optimal solution is practically more desirable.

Bearing this scenario, we thus formulate the problem of *bounded optimality* verification, which is to decide, given a

planning problem Π, a solution plan π to Π, and a bound k, whether the length $|\pi^*|$ of an optimal solution π^* to Π satisfies $|\pi| < |\pi^*| + k$. In other words, we want to verify whether the length of π is *not* larger than the length of an optimal solution by the bound k. (Note that both $|\pi^*|$ and π^* are *not* given as input.)

In spite of the fact that the bounded optimality verification problem describes a scenario different from the one described by the plan optimality verification problem, these two problems are actually equivalent from the theoretical point of view. This is because the bounded optimality verification problem is identical to asking whether there exist *no* solution plans π' to Π such that $|\pi| - k > |\pi'|$. For if such a π' exists, we have $|\pi^*| \leq |\pi'|$ because π^* is an optimal solution, and hence, $|\pi| > |\pi'| + k \geq |\pi^*| + k$, which is a contradiction. Consequently, for any planning formalism, the bounded optimality verification problem with π and k being the given plan and bound, respectively, is again the complement of the bounded plan existence problem in which the bound is $|\pi| - k$ and is encoded in *unary*.

Proposition 4. *The bounded plan optimality verification problem (with the bound given in binary) has the same complexity as the plan optimality verification problem, independent of planning formalisms.*

We have already mentioned earlier that in the (bounded) plan optimality verification problem, what really matters is the length of the given plan. As a consequence, we can further generalize those problems by replacing the given plan with the length of the plan. That is, given a planning problem Π, and *two* numbers k_π and k where k_π is the length of some solution, we want to decide whether there exist *no* solution plans π' to Π of length k' such that $k_\pi - k' > k$. We argue that this generalized version is useful in the scenario of *modeling assistance* where a (planning) domain modeler would like to know whether a domain is correctly modeled (McCluskey, Vaquero, and Vallati 2017; Lin and Bercher 2021, 2023; Lin, Grastien, and Bercher 2023). One way to do so is by validating whether certain properties hold in the domain. In our case, one could ask whether there exists an optimal solution within a range of k, provided a claim that there is a solution π with $|\pi|$ steps (in some domains, the modeler might be aware that the solution π exists, but doesn't want to write it down for the purpose of asking this question).

For this generalized version of the bounded plan optimality verification problem, since we replace the given plan with a number, one could recognize that its complexity is the complement of the bounded plan existence problem *without* encoding the bound in unary, independent of planning formalisms.

Proposition 5. *The complexity of the bounded plan optimality verification problem (with the bound given in binary) where the plan is not explicitly given is the complement of the bounded plan existence problem, independent of planning formalisms.*

As a special case, when the bound is zero, the bounded plan optimality verification problem boils down to the plan optimality verification problem where a plan is replaced by its length.

Proposition 6. *The complexity of plan optimality verification where only the length of a plan is given is the complement of the bounded plan existence problem (with the bound given in binary).*

Conclusion and Extension

We studied the computational complexity of several questions centered at the bounded plan existence problem. Our results show that in classical planning and grounded HTN planning, the computational complexity of plan verification lies in the range of **P** to **NP**-complete, whereas it increases dramatically in lifted HTN planning. For the bounded plan existence problem, its complexity ranges from **PSPACE**-complete to **NEXPTIME**-complete depending on planning formalisms, and the complexity decreases when the bound is encoded in unary. The problem of (bounded) plan optimality verification is the complement of bounded plan existence with the bound given in unary if the plan to be verified is explicitly given, and it is the complement of the bounded plan existence problem with the bound given in binary if only the length of the plan is given.

Extension In practice, an optimal solution usually refers to a plan of a minimal *cost*. That is, each action (in a planning problem) has a certain cost, and we want to find a solution plan which has an optimal cost (the cost of a plan is the sum of the cost of each action in it). The bounded plan existence problem studied in the paper can be viewed as a special case of this task where each action has cost one. Thus, the complexity results presented here naturally serve as a lower bound. In fact, the same upper bound also holds because for finding a cost optimal plan, we can again guess a plan up to a certain cost and then verify whether the guessed plan is a solution.

References

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight Bounds for HTN Planning. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling, ICAPS 2015*, 7–15. AAAI.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2014. On the Feasibility of Planning Graph Style Heuristics for HTN Planning. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling, ICAPS 2014*, 2–10. AAAI.

Arora, S.; and Barak, B. 2009. *Computational Complexity – A Modern Approach*. Cambridge University Press.

Bäckström, C.; and Jonsson, P. 2011. All PSPACE-Complete Planning Problems Are Equal but Some Are More Equal than Others. In *Proceedings of the 4th Annual Symposium on Combinatorial Search, SoCS 2011*, 10 – 17. AAAI.

Barták, R.; Ondrcková, S.; Behnke, G.; and Bercher, P. 2021. On the Verification of Totally-Ordered HTN Plans. In *Proceedings of the 33rd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2021*, 263–267. IEEE.

Behnke, G.; Höller, D.; Bercher, P.; and Biundo, S. 2016. Change the Plan - How Hard Can That Be? In *Proceedings*

of the 26th International Conference on Automated Planning and Scheduling, ICAPS 2016, 38–46. AAAI.

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 25th International Conference on Automated Planning and Scheduling, ICAPS 2015*, 25–33. AAAI.

Behnke, G.; Höller, D.; and Biundo, S. 2019. Finding Optimal Solutions in HTN Planning - A SAT-based Approach. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019*, 5500–5508. IJCAI.

Behnke, G.; and Speck, D. 2021. Symbolic Search for Optimal Total-Order HTN Planning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence, AAAI 2021*, 11744–11754. AAAI.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019*, 6267–6275. IJCAI.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI 2017*, 480–488. IJCAI.

Bercher, P.; Lin, S.; and Alford, R. 2022. Tight Bounds for Hybrid Planning. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence, IJCAI 2022*, 4597–4605. IJCAI.

Bylander, T. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, 94(1-2): 165–204.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity Results for HTN Planning. *Annals of Mathematics and Artificial Intelligence*, 18(1): 69–93.

Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1991. Complexity, Decidability and Undecidability Results for Domain-Independent Planning: A Detailed Analysis. Technical Report CS-TR-2797, UMIACS-TR-91-154, SRC-TR-91-96, University of Maryland, College Park, Maryland, USA.

Fink, E.; and Yang, Q. 1992. Formalizing Plan Justifications. In *Proceedings of the 9th Conference of the Canadian Society for Computational Studies of Intelligence, CSCSI 1992*, 9–14. ACM.

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011*, 1955–1961. IJCAI.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning – Theory and Practice*. Elsevier.

Helmert, M. 2006. New Complexity Results for Classical Planning Benchmarks. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling, ICAPS 2006*, 52–62. AAAI.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In *Proceedings of the 21st European Conference on Artificial Intelligence, ECAI 2014*, 447–452. IOS.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2004*, 294–301. IEEE.

Karpas, E.; and Domshlak, C. 2009. Cost-Optimal Planning with Landmarks. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009*, 1728–1733. IJCAI.

Lin, S.; Behnke, G.; Ondrčková, S.; Barták, R.; and Bercher, P. 2023. On Total-Order HTN Plan Verification with Method Preconditions – An Extension of the CYK Parsing Algorithm. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI 2023*. AAAI.

Lin, S.; and Bercher, P. 2021. Change the World - How Hard Can that Be? On the Computational Complexity of Fixing Planning Models. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI 2021*, 4152–4159. IJCAI.

Lin, S.; and Bercher, P. 2023. Was Fixing this *Really* That Hard? On the Complexity of Correcting HTN Domains. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence AAAI 2023*. AAAI.

Lin, S.; Grastien, A.; and Bercher, P. 2023. Towards Automated Modeling Assistance: An Efficient Approach for Repairing Flawed Planning Domains. In *AAAI 2023*. AAAI.

McCluskey, T. L.; Vaquero, T. S.; and Vallati, M. 2017. Engineering Knowledge for Automated Planning: Towards a Notion of Quality. In *Proceedings of the 9th Knowledge Capture Conference, K-CAP 2017*, 1–8. ACM.

Olz, C.; and Bercher, P. 2019. Eliminating Redundant Actions in Partially Ordered Plans – A Complexity Analysis. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling, ICAPS 2019*, 310–319. AAAI.

Olz, C.; Biundo, S.; and Bercher, P. 2021. Revealing Hidden Preconditions and Effects of Compound HTN Planning Tasks - A Complexity Analysis. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence, AAAI 2021*, 11903–11912. AAAI.

Pommerening, F.; Röger, G.; Helmert, M.; and Bonet, B. 2014. LP-Based Heuristics for Cost-Optimal Planning. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling, ICAPS 2014*, 226–234. AAAI.

Sreedharan, S.; Bercher, P.; and Kambhampati, S. 2022. On the Computational Complexity of Model Reconciliations. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence, IJCAI 2022*, 4657–4664. IJCAI.

Can HTN Planning Make Flying Alone Safer?

Jane Jean Kiam, Prakash Jamakatel

University of the Bundeswehr Munich, Neubiberg, Germany
{jane.kiam, prakash.jamakatel}@unibw.de

Abstract

Safety aspects in general aviation can be a limiting factor to gear toward introducing more single-pilot operations (SPOs), which are currently commonly practised by private pilots of ultralight aircraft, but are also a key to future developments in urban air mobility. The risks of SPOs are mainly due to the lack of redundancy, especially in case of emergeny; the development of reliable onboard companion technology is therefore deemed beneficial. This paper investigates how Hierarchical Task Networks (HTN), and more specifically the Hierarchical Domain Definition Language (HDDL), can be used to encode private pilots' maneuvers. Additionally, challenges are underlined on onboard companion technologies for SPOs, alongside with some features to be derived from hierarchical planning techniques to overcome these challenges.

Introduction

Human factors have been identified as a critical aspect in aviation safety risks, and are considered as part of the European Plan for Aviation Safety - EPAS (European Union Aviation Safety Agency (EASA) 2021). Statistical evidence on fatal accidents in the operations of ultralight aviation reflect that a lack of redundancy in single-pilot (SP) cockpits will accentuate the criticality of human factors in causing fatal accidents. According to accident analyses for ultralight aviation (De Voogt et al. 2018; BFU 2022), two main contributing human factors to be accounted for are: i) lack of knowledge or experience leading to skill-based and decision errors, as well as ii) excessive mental workload leading to perception and decision-based errors. Both factors affect the pilot's decision-making capability even more substantially in emergency situations.

Meanwhile, (Biundo et al. 2016) analysed how the cross-disciplinary field of "companion technology" leverages sensor data fusion, planning and learning, as well as human-machine interaction to achieve artificially intelligent companion to human users, which can also assist human users in accomplishing complex tasks. Recent advancements have integrated AI planning techniques into companion technologies. For example, ROBERT in (Behnke et al. 2020) exploits Hierarchical Task Network (HTN) planning for instructing novices in operating complex hand tools, while CHAP-E in (Benton et al. 2018) also leverage hierarchical planning to guide pilots through a safety-checklist in view of more reliable operation of modern aircraft. However, the potential extension of the work on CHAP-E by the bigger community is unclear, since i) CHAP-E uses PLEXIL (Verma et al. 2005) for modelling the tasks, and ii) the task models are not available. Besides hierarchical planning, the use of a hybrid (PDDL+ compatible) planner (Scala et al. 2016) was also demonstrated in (León, Kiam, and Schulte 2021) to plan for emergency landing trajectories, so that the onboard autopilot can take over, should the (single) pilot be incapacitated.

Although planning techniques have been investigated for use as onboard companion technology of an aircraft, some shortcomings are still prominent. While (León, Kiam, and Schulte 2021) only considers the low-level trajectory planning, (Benton et al. 2018) focuses mainly on guiding the pilot to execute plans according to standard operating procedures, without explaining how the companion technology knows if the guidance is appropriate at the moment. In this paper, we explore the use of hierarchical planning techniques as part of the onboard companion technology in aviation, and more specifically in an SP ultralight-cockpit, as the resolution of this problem paves way to future Urban Air Mobility (UAM). To this end, we illustrate how Hierarchical Domain Description Language (HDDL) (Höller et al. 2020a) can be used to model pilot tasks defined in structured manuals (for private pilots)[1]. Subsequently, critical features of an onboard companion technology are analyzed alongside insights of how HTN-planning can contribute. Finally, usability of the features with respect to their reliability and performance are discussed. Note that we keep to the term "onboard companion technology" in the paper, which is the equivalent of "cockpit assistance system" in many engineering-related fields.

Modelling Private Pilot's Tasks in HDDL

Enabling SPOs is essential to scale up fleet size in general aviation for future UAM (European Union Aviation Safety Agengy - EASA 2021). To pave the way for acceptable and safe UAM, advancements in companion technology for SP-cockpits are necessary. While the definition of pilots' roles

[1]Detailed domain modelled in HDDL is available here: https://github.com/UniBwM-IFS-AILab/ValidationTests

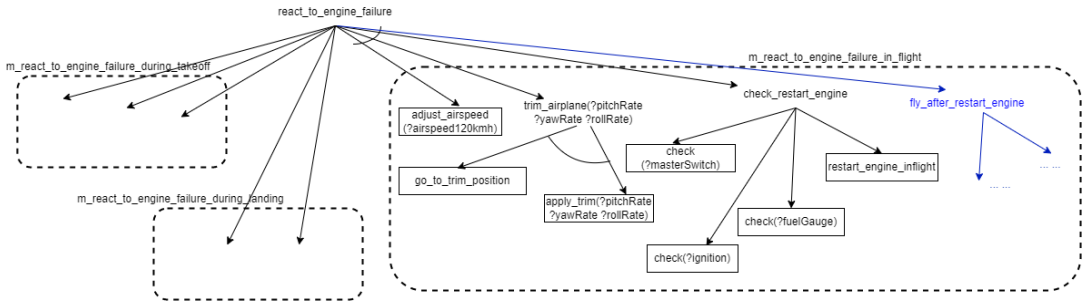


Figure 1: HTN model of pilot's initial task network to react to engine failure. Tasks that are encircled by boxes with black edges are known as actions (or primitive tasks) that do not further decompose. Compound tasks are decomposed into subtasks.

in UAM is still in its infancy[2], private pilots trained to fly ultralight aircraft (Pooley 2003; SHARK 2017) in a SP-cockpit provide a solid basis to study onboard companion technologies for SPOs, while also benefiting directly from the advancements too, as ultralight aircraft to date is generally not equipped with computed-aided intelligence.

First and foremost, knowledge on the standard operating procedures must be possessed by the onboard companion technology in order to provide meaningful assistance. To this end, we model the private pilots' tasks as totally or partially ordered tasks using HDDL, a task modelling language for hierarchical planning that was formally defined in (Höller et al. 2020a) and used in the previous IPC on Hierarchical Planning[3]. Furthermore, PANDA, a framework developed for hierarchical planning with HDDL as the native modelling language, is to date the only one with an integrated Plan and Goal Recognition (PGR) method[4], which is an essential feature of the onboard companion technology.

In the following two tasks are depicted, namely to react to an engine failure during flight and to react to the engine on fire during flight. We illustrate the similarity of both tasks, as well as the different actions required from the pilot.

Engine Failure

Without immediate and appropriate countermeasures from the pilot, engine failure can lead to fatal accident. However, given the frequency of occurrence, as well as the training program that does not prescribe obligatory refreshment course, private pilots are not necessarily fit to react to these events, causing either the undertaking of wrong decisions or mental overload that leads to delayed reactions.

Encoding the HTN models of the pilot's tasks in HDDL enables the possession of standard knowledge by the onboard companion technology. Figure 1 represents the decomposition graphically of the compound task `react_to_engine_failure` using different methods, depending on the flight

```
(:method m_react_to_engine_failure_in_flight
 :parameters (?airspeed120kmh - Airspeed
    ?pitchRate ?yawRate ?rollRate - AttitudeRate)
 :task (react_to_engine_failure)
 :precondition (and (p_engineFailure) (p_inFlight))
 :subtasks (and
    (task1 (adjust_airspeed ?airspeed120kmh))
    (task2 (trim_airplane
            ?pitchRate ?yawRate ?rollRate))
    (task3 (check_restart_engine))
    (task4 (fly_after_restart_engine)))
 :ordering(and
    (< task1 task3) (< task2 task3) (< task3 task4)))

(:action restart_engine_inflight
 :parameters ()
 :precondition (and (p_engineFailure) (p_inFlight))
 :effect (and (p_attemptedEngineRestart)))
```

Figure 2: HDDL-encoding of the method to decompose pilot's task once an in-flight engine failure is detected, and the action to restart engine during flight.

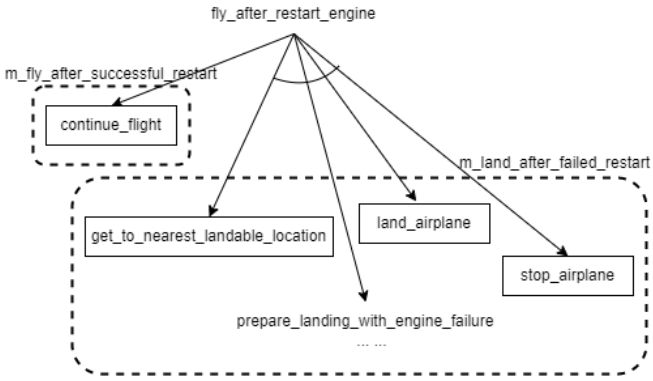


Figure 3: HTN model of pilot's task to fly after an in-flight engine restart. For more concise representation, the further decomposition of the compound task `prepare_landing_with_engine_failure` is not shown.

[2]Training programs for pilots of future UAM are being conceptualised as a substantial surge of demand in pilots for air taxi is predicted (CAE 2021).

[3]http://gki.informatik.uni-freiburg.de/competition/

[4]https://github.com/panda-planner-dev/pandaPIpgrRepairVerify

phase (i.e. takeoff, in-flight, or landing). The method `m_react_to_engine_failure_in_flight` encoded in HDDL is shown in Figure 2. The decomposition of the subtask `fly_after_restart_engine` (in blue) is depicted in Figure 3. However, since the decomposition depends on if the engine is restarted successfully or not during flight (as an information gained from a reactive observation of the system), the subtask is currently not encoded in HDDL as part of the method `m_react_to_engine_failure_in_flight`.

Engine on Fire During Flight

Figure 4 depicts a method for decomposing the compound task to react to the engine on fire during flight (which is itself a different initial task network than `react_to_engine_failure`), alongside with its formulation in HDDL in Figure 5. Although the emergency stems from the engine, but due to a different root cause, i.e. a fire instead of a mechanical failure, the pilot is required to act differently, i.e. instead of trying to check all mechanical parts and restart the engine, here, the pilot has to cut all injections to the engine and perform an emergency landing immediately to evacuate.

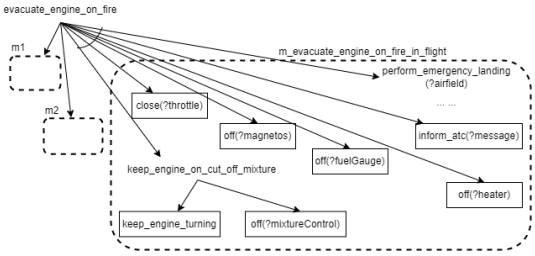


Figure 4: HTN model of pilot's task to evacuate if the engine has caught fire; for a more concise representation, the method `m_evacuate_engine_on_fire_during_takeoff` and `m_evacuate_engine_on_fire_during_landing` are replaced by `m1` and `m2` respectively.

Challenging Essential Features for the Onboard Companion

While it is possible to model complex tasks extracted from the pilot's manual using HDDL, the exploitation of these in an end-to-end onboard companion remains challenging. Below are a few identified requirements, or rather challenges (according to gaps identified by domain experts in aviation (BFU 2022; SIAF 2009)) to be fulfilled in view of a more meaningful exploitation of these HTN models.

Challenge 1: Dynamic guidance

Providing guidance using software-enabled companion technology is a timelier mean than the current practice, which consists of having the pilot to refer to the flight manual in case of an unexpected emergency situation, for which the pilot does have possess adequate knowledge to tackle. Furthermore, it also avoids erroneous execution of tasks.

```
(:method m_evacuate_engine_fire_in_flight
:parameters ( ?mixtureControl - MixtureControl
    ?throttle - AircraftPart ?magnetos - Magnetos
    ?fuelGauge - FuelGauge ?heater - CabinHeater
    ?message - Message ?atc - ATC ?airfield - Location)
:task (evacuate_engine_fire ?message ?airfield)
:precondition (and (p_pilotInAirplane)
    (p_inFlight) (p_engineOnFire)
    (p_mixtureControlOn ?mixtureControl)
    (p_open ?throttle) (p_magnetosOn ?magnetos)
    (p_fuelGaugeOn ?fuelGauge)
    (p_cabinHeaterOn ?heater))
:subtasks (and
    (task1 (keep_engine_cut_off_mixture ?mixtureControl))
    (task2 (close ?throttle))
    (task3 (off ?magnetos))
    (task4 (off ?fuelGauge))
    (task5 (off ?heater))
    (task6 (inform_ATC ?atc ?message)))
    (task7 (perform_emergency_landing ?airfield))
:ordering (and (< task1 task2) (< task2 task3)
    (< task3 task4) (< task4 task5)
    (< task5 task6) (< task6 task7)))
```

Figure 5: HTN model in HDDL

While the task models encoded in HDDL enable the use of HDDL-compatible planners to determine plans (sequences of actions to be executed by the pilot), the planners are meant mainly for offline planning, i.e. information gained during the execution of the plans cannot be considered automatically. For example, the feasibility of the method in Figure 3 for decomposing `fly_after_restart_engine`, which is a compound task of `react_to_engine_failure`, depends on if the engine restarts after the action `restart_engine_inflight`. However, in an offline planning framework, the effect of the action `restart_engine_inflight` does not contain this information, but merely the fact that the attempt to restart engine was completed.

Hierarchical Operational Models as described in (Patra et al. 2020) can be useful to parameterize the method to be undertaken for decomposing compound tasks of further future, depending on the outcomes (i.e. effects detected during execution) of previous actions. (Höller et al. 2020b) proposed an HTN plan repair approach that does not involve changing the planning engine. Instead, it suffices to generate a problem file for re-planning by considering the executed plan prefix. However, the approach requires first a complete plan to be generated, with complete knowledge on effects of all actions, which is in our case impossible since the decomposition of `fly_after_restart_engine` depends on if the engine is restarted, and how the syntax of HDDL is defined currently does not allow the effect of an action be modeled in a way that information is proactively extracted from the external environment.

Challenge 2: Automated context-based guidance

Using HTN-planning to provide flight guidance proactively can help i) to reduce reaction time to emergency, ii) to ensure that the pilot carries out the correct steps (and in some cases, even in the right ordering), as well as iii) to enable the pilot to maintain an acceptable mental workload, which is essential in emergency situations (SIAF 2009).

However, *any form of guidance is only meaningful, if the context is known*. For the onboard companion technology to function in real time, information on the pilot's intention (i.e. the intended "goal") must be tracked in real time, so that the guidance, or rather the plan suggestion according to the HTN models is appropriate. The intended "goal" can either be communicated "on-demand" by the pilot, which in an emergency situation may add on to the pilot's mental workload, or can be detected in an automated manner.

Höller et al. describe in (Höller et al. 2018) an automated plan and goal recognition method based on hierarchical planning using observed actions executed by the human actor (i.e. "observations"). While this is a function to be integrated into the onboard companion technology to recognise the initial task network (as the pilot's intention) in an automated manner and in real-time, for a more robust intention recognition, some shortcomings of the work in (Höller et al. 2018) must be overcome:

- recognition of multiple plans/goals, as the pilot may be multitasking;
- plans/goals recognition despite *missing* observations due to non-critical actions missed by the pilot, and *noisy* observations due to the pilot being unclear about his/her actions in emergency situations (e.g. wrong button pressed, undo wrong actions, etc.).

Challenge 3: Warning system with forward prediction

Automatic alert has been used for many Advanced Driver Assistance Systems (ADAS) to communicate warning signals in view of mitigating risks of fatal accidents (Ziebinski et al. 2017). With the automated PGR integrated, the automatic warning system with forward prediction can be developed for the onboard companion technology, i.e. a risk prediction based on pilot's current (recognised) plan/goal. Using forward prediction, by extrapolating the effects of actions and cross-checking them with predicted (future) environment, risks encountered in near future, and therefore also potential danger, can be estimated so that pilots can still abort his/her current goal or correct his/her action.

Additionally, with the formally encoded HTN models in HDDL, another advantage to derive from them is the capability to check and warn for missing or erroneous actions, e.g. in case the pilot does not sink to a low enough altitude while performing an emergency landing on a short runway.

Conclusion and Discussions on Usability and Acceptability

In this paper, we describe the motivation of using hierarchical planning for onboard companion technology in single-pilot cockpits. In addition to augmenting safety of ultralight micro-aircraft, increased safety in single-pilot operations will facilitate the development of future UAM. To this end, we use HDDL, a very formally defined modelling language for HTN, to model private pilots' tasks. As HTN-planning techniques compatible with HDDL are being further developed, the HTN-models encoded in HDDL will call for more contributions among the hierarchical planning community. Subsequently, we underline several challenges to overcome in order to fully utilize these HTN-models in assisting pilots in a timely and reliable manner.

While an onboard company technology for aircraft in SPOs is beneficial, we highlight that the technology is only acceptable and usable if the following criteria are met.

Real-time and realistic capability

As an onboard companion technology is intended also for automated guidance and warning during flight, the real-time capability is critical to ensure that assistance is provided without delay, or in practice, with acceptable latency. For this purpose, the performance in planning, as well as in PGR must be tested and validated using realistic mission scenarios and by basing on human-in-the-loop tests, as well as by adapting to the dynamics of the platform (e.g. airspeed).

Replanning capability

In highly dynamic environment, during flight and moreover in emergency situations, the pilot's intended goal can change. For example, the pilot decides to perform an emergency landing after anomalies in the engine are detected, but realizes that the targeted landing stripe is not free for landing. In this case, the pilot can decide to continue flight while scouting for the next landing stripe.

The PGR method as developed by (Höller et al. 2018) uses a sequence of observations to predict the plan/goal currently intended by the human actor. However, as described above, a pilot can change the course of action, resulting in past actions being "invalid". Without a reliable method to filter past actions in a temporal manner (i.e. older actions lose their validity), the PGR can be obscured. Whether or not the approach for plan repair described in (Höller et al. 2020b) (that is independent of the planning engine used) can be adapted for PGR, will be investigated.

Interpretability

For AI-techniques to be considered part of a mission-critical system with human-AI interactions, system transparency is essential (European Union Aviation Safety Agency (EASA) 2020), to ensure that the human actor can intervene whenever necessary, thanks to adequate plan explanation, and also to ensure that complacency will not become an issue due to over-reliance on AI. Metrics to measure interpretability must be considered. While classical planning has advanced substantially in this regard (Sreedharan et al. 2021), little work is done on this aspect for hierarchical planning.

Acknowledgments

This work is funded by the German Federal Ministry of Economic Affairs and Climate Action (Project MOREALIS).

References

Behnke, G.; Bercher, P.; Kraus, M.; Schiller, M.; Mickeleit, K.; Häge, T.; Dorna, M.; Dambier, M.; Manstetten, D.; Minker, W.; Glimm, B.; and Biundo, S. 2020. New Developments for Robert – Assisting Novice Users Even Better in DIY Projects. *Proceedings of the International Conference on Automated Planning and Scheduling*.

Benton, J.; Smith, D.; Kaneshige, J.; Keely, L.; and Stucky, T. 2018. CHAP-E: A Plan Execution Assistant for Pilots. *Proceedings of the International Conference on Automated Planning and Scheduling*.

BFU. 2022. Studie zur Flugsicherheit von Luftsportgeräten - Analyse von Unfällen und Störungen mit Luftsportgeräten in Deutschland in den Jahren 2000-2019. Technical Report BFU22-803.1, Bundesstelle für Flugunfalluntersuchung.

Biundo, S.; Höller, D.; Schattenberg, B.; and Bercher, P. 2016. Companion-Technology: An Overview. *KI - Künstliche Intelligenz*, 30(1): 11–20.

CAE. 2021. Pilot Training for Advanced Air Mobility.

De Voogt, A.; Chaves, F.; Harden, E.; Silvestre, M.; and Gamboa, P. 2018. Ultralight Accidents in the US, UK, and Portugal. *Safety*, 4(2): 23.

European Union Aviation Safety Agency (EASA). 2020. Artifial Intelligence Roadmap: A Human-Centric Approach to AI in Aviation.

European Union Aviation Safety Agency (EASA). 2021. The European Plan for Aviation Safety - EPAS 2022-2026.

European Union Aviation Safety Agengy - EASA. 2021. Study on the Societal Acceptance of Urban Air Mobility in Europe.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2018. Plan and Goal Recognition as HTN Planning. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020a. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020b. HTN Plan Repair via Model Transformation. In *Proceedings of the 43th German Conference on Artificial Intelligence (KI)*, 88–101. Springer.

León, B. S.; Kiam, J. J.; and Schulte, A. 2021. A Fault-Tolerant Automated Flight Path Planning System for an Ultralight Aircraft. In *AIxIA 2020 – Advances in Artificial Intelligence*. Springer International Publishing.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. 2020. Integrating Acting, Planning, and Learning in Hierarchical Operational Models. In *Proceedings of the 30th ICAPS*.

Pooley, D. 2003. *POOLEYS Private Pilots Manual: JAR Flying Training, Volume 1*. Cranfield, UK: POOLEYS.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramirez, M. 2016. Interval-based relaxation for general numeric planning. In *ECAI 2016*.

SHARK. 2017. *Flight Manual: UL airplane*. SHARK.AERO CZ s.r.o.

SIAF. 2009. Ultralight Aviation Safety and its Improvement through Accident Investigation. Technical Report Safety study S1/2009L, Onnettomuustutkintakeskus Centralen för undersökning av olyckor - Accident Investigation Board of Finland.

Sreedharan, S.; Kulkarni, A.; Smith, D.; and Kambhampati, S. 2021. A Unifying Bayesian Formulation of Measures of Interpretability in Human-AI Interaction. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization.

Verma, V.; Estlin, T.; Jonsson, A.; Pasareanu, C.; Simmons, R.; and Tso, K. 2005. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*.

Ziebinski, A.; Cupek, R.; Grzechca, D.; and Chruszczyk, L. 2017. Review of advanced driver assistance systems (ADAS). In *AIP Conference Proceedings*.

HDDL 2.1: Towards Defining a Formalism and a Semantics for Temporal HTN Planning

Damien Pellier,[1] **Alexandre Albore,**[2] **Humbert Fiorino,**[1] **Rafael Bailon-Ruiz** [2]

[1]University of Grenoble Alpes, LIG, Grenoble, France
{damien.pellier, humbert.fiorno}@imag.fr
[2]ONERA/DTIS, University of Toulouse, France
{alexandre.albore, rafael.bailon_ruiz}@onera.fr

Abstract

Real world applications as in industry and robotics need modelling rich and diverse automated planning problems. Their resolution usually requires coordinated and concurrent action execution. In several cases, these problems are naturally decomposed in a hierarchical way and expressed by a Hierarchical Task Network (HTN) formalism. HDDL, a hierarchical extension of the Planning Domain Definition Language (PDDL), unlike PDDL 2.1 does not allow to represent planning problems with numerical and temporal constraints, which are essential for real world applications. We propose to fill the gap between HDDL and these operational needs and to extend HDDL by taking inspiration from PDDL 2.1 in order to express numerical and temporal expressions. This paper opens discussions on the semantics and the syntax needed for a future HDDL 2.1 extension.

1 Introduction

Real world applications of Automated Planning, like in industry and robotics, require modelling rich and diverse scenarios. Such planning problems are often naturally decomposed in a hierarchical way, with compound tasks that refine in different ways their execution model. These real world applications of planning use both numerical and temporal constraints to define the agents synchronisation on collaborative tasks, and sub-task decomposition. In fact, concurrency between actions, their duration, and agents coordination in HTN problems are needed to find solutions for nontrivial tasks in complex scenarios and require to make explicit the representation of time (Ghallab, Nau, and Traverso 2016).

The Hierarchical Task Network (HTN) formalism (Erol, Hendler, and Nau 1994) is used to express a wide variety of planning problems in real-world applications, e.g., in task allocation for robot fleets (Milot et al. 2021), video games (Menif, Jacopin, and Cazenave 2014) or industrial contexts such as software deployment (Georgievski et al. 2017). Over the last years, much progress has been made in the field of hierarchical planning (Bercher, Alford, and Höller 2019). Novel systems based on the traditional, search-based techniques have been introduced (Bit-Monnot, Smith, and Do 2016; Ramoul et al. 2017; Shivashankar, Alford, and Aha 2017; Bercher et al. 2017; Höller et al. 2019, 2020; Höller and Bercher 2021), but also new techniques like the translation to STRIPS/ADL (Alford, Kuter, and Nau 2009; Alford et al. 2016; Behnke et al. 2022), or revisited approaches like the translation to propositional logic (Behnke, Höller, and Biundo 2018, 2019; Schreiber et al. 2019; Schreiber 2021; Behnke 2021). Despite these advances, not all planning systems use the same formalism to represent hierarchical task decomposition, making it difficult to compare approaches and promote HTN planning techniques.

An extension of PDDL (Planning Domain Description Language) (Mcdermott et al. 1998), called HDDL (Hierarchical Planning Domain Description Language) (Höller et al. 2020), has been proposed to address this issue. HDDL is based on PDDL 2.1 (Fox and Long 2003) and is the result of several discussions within the planning community (Behnke et al. 2019) to fill the need of a standard language for the first Hierarchical Planning track of International Planning Competitions (IPC) in 2020. However, it was decided that the first version of HDDL would not include any of the temporal or numerical features of PDDL due to efforts to develop the language and related tools. In this paper, we illustrate the challenge of defining the semantics for a temporal extension of HDDL to meet the needs of the planning community and planning applications.

Our motivation is grounded on the compelling need to devise applications involving autonomous systems. We propose to extend HDDL, by including elements of PDDL 2.1 and ANML (*Action Notation Modeling Language*) (Smith, Frank, and Cushing 2008), to express temporal and numerical constraints. This is intended to initiate discussions within the HTN community on establishing a standard – HDDL 2.1 – aimed at filling the gaps between existing hierarchical-temporal planning approaches. To that end, we make this preliminary extension of HDDL an open source project with a public repository, where we propose a full syntax as well as a set of benchmarks based on this extension[1] and a parser for it, as part of the PDDL4J[2] library (Pellier and Fiorino 2018).

The rest of the paper is organised as follows. In Section 2 we define the basic concepts of the proposed extension. In Sections 3 and 4 we set down the semantics for Temporal

[1]https://github.com/pellierd/HDDL2.1
[2]https://github.com/pellierd/pddl4j

HTN planning. We conclude on the central aspects of this planning paradigm, and on future work.

2 Lifted Temporal HTN planning

Throughout this section, we will use common notations from first-order logic, which we assume to be known. In the lifted formalism of HDDL 2.1, we assume for the sake of simplicity that all logical formulas are over a *function-free* first-order logic language $\mathcal{L} = (V, C, P)$. $\mathcal{L}$ consists of sufficiently many *constant* $c \in C$ representing the *objects* in the real world, *variables* $x \in V$ and *predicates* $p \in P$. Predicates have parameters that are either variables or constants. The predicate arity is the number of predicate parameters. For instance, $p(x, c)$ is a 2-arity predicate. We can now define *formulas* in a function-free first-order logic: (i) a predicate is a formula ; (ii) if ϕ and ψ are formulas, then $\neg\phi$, $\phi \vee \psi$ and $\phi \wedge \psi$ are formulas ; (iii) if ϕ is a formula and x is a variable, then $\forall x\phi$ is a formula. We define $\exists x\phi$ as $\neg\forall x\neg\phi$, and $\phi \rightarrow \psi$ as $\neg\phi \vee \psi$. $\forall$ and $\exists$ are respectively the universal and the existential quantifier. Conceptually, grounding a formula consists in generating a set of variable-free i.e. *ground* formulas (Helmert 2009) as follows: a variable x in a quantifier-free formula ϕ is eliminated by replacing ϕ with $|C|$ copies, one for each $c \in C$, where x is substituted with c in the respective copy. This substitution is denoted by $\phi[x/c]$. Regarding quantified formulas, $\forall x\phi$ is replaced by $\bigwedge_{c\in C} \phi[x/c]$ and $\exists x\phi$ by $\bigvee_{c\in C} \phi[x/c]$. We refer the reader to the work of by Behnke et al. (2020); Ramoul et al. (2017) for further details on grounding implementation. Note that it is always possible to transform a formula in function-free first-order logic into a finite set of ground formulas in propositional logic.

A *state* s is a set of ground predicates. For the sake of conciseness, we will also consider s as a Herbrand interpretation that assigns $true$ to all ground predicates in s, and $false$ to all ground predicates not in s. From this, a truth value can be computed for every *ground* formula from $\mathcal{L}$ by using the usual rules for logical composition. Without loss of generality, a formula (not necessarily ground) ϕ is true in s if and only if grounding ϕ generates at least one ground formula true in s. We will use the notation $s \models \phi$ to mean that the formula ϕ is true in s.

A key concept in HTN planning and a fortiori in temporal HTN planning is the concept of *task*. Each task is given by a name and a list of parameters. We distinguish two kinds of tasks: the primitive tasks (also called actions), and the abstract tasks (or compound tasks). Primitive tasks are carried out by durative actions in the sense of classical temporal planning (Fox and Long 2003), while abstract tasks can be refined by applying methods that define the decomposition of the task into subtasks. The purpose of abstract tasks is not to induce a state transition. Instead, they refer to a predefined mapping to one or more tasks that can refine the abstract task. For instance, in the task of serving a dinner, *deliver-dinner(?food-style, ?place)* is the compound task consisting in performing first the task of serving the starters, then the main course, etc. In that sense, *deliver-dinner(?food-style, ?place)* can be refined in: ⟨ *serve-starters(?food-style, ?place)*, *serve-main-course(?food-style, ?place)*, etc.⟩ This mapping between tasks is achieved by a set of decomposition methods, namely the methods (Def. 5) and the Temporal Task Networks (Def. 6).

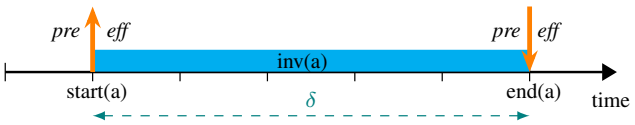


Figure 1: Timeline of a durative action *a* application.

We first define the planning domain and problem for Temporal HTN Planning.

Definition 1. *A* planning domain $\mathcal{D}$ *is a tuple* $(\mathcal{L}, \mathcal{T}, \mathcal{J}, \alpha, \mathcal{A}, \mathcal{M})$*, where* $\mathcal{L}$ *is the first-order logic language,* $\mathcal{T}$ *is the set of tasks,* $\mathcal{J}$ *is the set of task identifiers*[3]*,* $\alpha : \mathcal{J} \rightarrow \mathcal{T}$ *is the function that maps task identifiers to tasks,* $\mathcal{A}$ *is a set of actions constituted by* snap *actions (Def. 3) and* durative *actions (Def. 4),* $\mathcal{M}$ *is the set of methods (Def. 5).*

The domain implicitly defines the set of all states S defined over all subsets of all ground predicates in $\mathcal{L}$.

Definition 2. *A* planning problem $\mathcal{P}$ *is a tuple* $(\mathcal{D}, s_0, w_0, g)$*, where* $\mathcal{D}$ *is a planning domain,* $s_0 \in S$ *is the initial state,* w_0 *is the initial temporal task network (not necessary ground), and* g *is a formula (not necessary ground) describing the goal.*

Let us start by defining the concepts of *snap* and *durative actions*, which are the primitive tasks, based on the definitions from Abdulaziz and Koller (2022). A snap action is an action whose execution is instantaneous in the sense of classical planning, meaning that it has a null duration between checking the preconditions and applying the effects.

Definition 3 (Snap Action)**.** *A* snap action a *is a tuple* $\big(name(a), precond(a), effect(a)\big)$*, where* $name(a)$ *is the name of* a*, the precondition* $precond(a)$ *is a first-order formula, and the effects* $effect(a) = effect^+(a) \cup effect^-(a)$ $(effect^+(a) \cap effect^-(a) = \emptyset)$*,* $effect^+(a)$ *and* $effect^-(a)$ *are conjunctions of predicates.*

Definition 4 (Durative Action)**.** *A* durative action a *is a tuple* $\big(name(a), start(a), end(a), inv(a), \delta\big)$*:* $name(a)$ *is the name of* a*;* $start(a)$ *and* $end(a)$ *are snap actions;* $inv(a)$ *is a first-order formula that must hold in all the states after the execution of* $start(a)$ *and until the execution of* $end(a)$*, and* δ *is the duration of* a*.*

Actions do change the state of the world. Durative actions also change the time of the model, shifting it by a quantity δ, as shown in Figure 1. State transitions will be formally defined in Section 3.

Unlike a primitive task, a *compound task* does not directly change the world state. A compound task is identified by a name and defines the way other –possibly ordered– tasks (either primitive or compound) must be achieved with respect

[3]Task identifiers are arbitrary symbols, which serve as place holders for the actual tasks they represent. Identifiers are needed because tasks can occur multiple times within the same task network, as we will see below.

to some constraints in order to refine it. Like primitive tasks, compound tasks have also a start event and an end event, which will be associated to dates in Section 3. In this sense, methods allow to refer tasks to temporal task networks.

Definition 5 (Method)**.** *A* method m *is a tuple* $\big(name(m), task(m), tn(m)\big)$*, where* $name(m)$ *is the name of the method,* $task(m)$ *is the task refined by the method, and* $tn(m)$ *is the temporal task network decomposing* $task(m)$*.*

Definition 6 (Temporal Task Network)**.** *A* temporal task network $w = (\mathcal{I}, \mathcal{C})$ *is given by:*

- $\mathcal{I} \subseteq \mathcal{J}$ *is a set (possibly empty) set of tasks identifiers;*
- $\mathcal{C}$ *is the set of constraints, with* $\mathcal{C} =< \mathcal{C}_o, \mathcal{C}_v, \mathcal{C}_d, \mathcal{C}_t >$*:*
  - $\mathcal{C}_o$ *is a set of temporal qualitative ordering constraints over the start or the end events of the tasks in* $\mathcal{I}$*. The possible qualitative temporal ordering are those from the classical point algebra (Broxvall and Jonsson 2003):* $<, \leq, >, \geq, =$ *and* $\neq$*;*
  - $\mathcal{C}_v$ *is a set of parameter constraints. Each constraint can bind two variables to be equal or non-equal, or similarly bind a variable to a constant;*
  - $\mathcal{C}_d$ *is a set of durative constraints over the duration of the tasks in* $\mathcal{I}$*;*
  - $\mathcal{C}_t$ *is a set of temporal decomposition constraints of the form* $(at\ e\ \phi)$*, expressing that some properties defined by the formula* ϕ *must hold in the state at date* e*.*

The temporal task network w *implicitely defines a temporal ordered multi-set of tasks* $\mathcal{T}' = \{\alpha(i) \mid i \in \mathcal{I}\}$*.*

A temporal task network explicits the decomposition of abstract tasks into subtasks. Note that a temporal task network is ground if all its variables are bound to constants, and primitive if all its tasks $\mathcal{T}'$ are primitive.

3 Temporal HTN Planning Semantics

The solution of a temporal HTN planning problem is an *executable* temporal task network that is obtained from the problem initial task network by applying method decomposition and constraint satisfaction.

Lifted problems are just a compact representation of their ground instances. Variable constraints are satisfied by the grounding, so there is no need to use them with ground instances. Therefore, for simplicity, this section defines the semantics of a lifted problem in terms of its ground instances. For details on the grounding process, the reader is referred to (Behnke et al. 2020; Ramoul et al. 2017).

Let us start by defining a temporal sequence of tasks.

Definition 7 (Temporal Sequence of Tasks)**.** *A* temporal sequence of tasks π *over a planning domain* $\mathcal{D}$*, is a sequence of tuples* $\langle (t_0, e_0, \delta_0), \ldots, (t_n, e_n, \delta_n)\rangle$ *where* $t_0, \ldots, t_n$ *are ground tasks defined over* $\mathcal{T}$*, and for* $0 \leq i \leq n$*, the natural numbers*[4] $e_i \in \mathbb{N}_{\geq 0}$ *and* $\delta_i \in \mathbb{N}_{\geq 0}$ *are the starting date and the duration of the task* t_i*, respectively. For a temporal sequence of tasks* π*, the set of dates* $\{e \mid (t, e, \delta) \in \pi\} \cup \{e + \delta \mid (t, e, \delta) \in \pi\}$ *induces a sorted sequence* $< e_0, \ldots, e_n >$ *of* happening events *of* π*. A temporal sequence of tasks* π *is primitive if and only if for every task* $(t, e, \delta) \in \pi$*,* t *is primitive, i.e.* t *is carried out by an action (either snap or durative).*

The *duration* of a task t is generally unbounded, as the bound would be the sum of the durations of the tasks of which t is compounded of. Only when a task t is primitive, then $duration(t)$ is given by the duration δ of the action that achieves t.

In order to guarantee the exutability of concurrent plans, in the sense of the "required concurrency" as described by Cushing, Kambhampati, and Weld (2007), a central notion is *non-interference*, i.e. when preconditions and effects of snap actions do not overlap. We consider that two snap actions a and b are *not interfering* if and only if (i) $precond(a) \cap \big(effect^+(b) \cup effect^-(b)\big) = \emptyset$, (ii) $precond(b) \cap \big(effect^+(a) \cup effect^-(a)\big) = \emptyset$, (iii) $effect^+(a) \cap effect^-(b) = \emptyset$ and $effect^+(b) \cap effect^-(a) = \emptyset$.

Two snap actions or more can be executed at the same time if they are pairwise non-interfering. The execution semantics of snap actions are similar to the semantics of $\forall$-step parallel plans (Rintanen, Heljanko, and Niemelä 2006), and used in PDDL 2.1 (Fox and Long 2003). With this notion in mind, we define an executable temporal sequence of tasks.

Definition 8 (Executable Temporal Sequence of Tasks)**.** *A temporal sequence of tasks* $\pi = \langle (t_0, e_0, \delta_0), \ldots, (t_n, e_n, \delta_n)\rangle$ *is executable in a state* s_0 *if and only if for every* $(t, e, \delta) \in \pi$*: (i)* t *is primitive; (ii)* e *is a happening event of* π*; (iii) state* s_i *at date* e_i *transitions to a new state* s_{i+1} *s.t.: given the set* B_{e_i} *of snap actions being executed at* e_i*:* $B_{e_i} = \big\{start(a_j) \,|\, (a_j, e_j, \delta_j) \in \pi \text{ and } e_j = e_i\big\} \cup \big\{end(a_k) | (a_k, e_k, \delta_k) \in \pi \text{ and } e_i + \delta_k = e_k\big\}$ *and given the set* I_{e_i} *of the invariants holding at* e_i*:* $I_{e_i} = \big\{inv(a_j) \,|\, (a_j, e_j, \delta_j) \in \pi \wedge e_j < e_i < e_j + \delta_j\big\}$ *the transition of* s_i *to* s_{i+1}*, given that all the snap actions in* B_{e_i} *are* pairwise non-interfering*, is defined as* $s_i \models precond(a)$ *for every* $a \in B_{e_i}$*,* $s_i \models inv(a)$ *for every* $inv(a) \in I_{e_i}$ *and* $s_{i+1} = \big(s_i - \cup_{a \in B_{e_i}} effect^-(a)\big) \bigcup_{a \in B_{e_i}} effect^+(a)$*.*

Definition 8 can be extended to a temporal task network.

Definition 9 (Executable Temporal Task Network)**.** *A temporal task network* $w = (\mathcal{I}, <\mathcal{C}_o, \mathcal{C}_v, \mathcal{C}_d, \mathcal{C}_t>)$ *is executable in a state* s_0 *if and only if there exists an executable temporal sequence of tasks* $\pi = \langle (\alpha(i_0), e_0, \delta_0), \ldots, (\alpha(i_n), e_n, \delta_n)\rangle$ *where* $i_0, \ldots, i_n$ *are task identifiers in* $\mathcal{I}$ *that matches the following conditions: (i)* π *matches the temporal constraints* $\mathcal{C}_o$*, (ii)* π *matches the duration constraints* $\mathcal{C}_d$*, and (iii) the sequence of states and their associated dates* $\langle (s_0, e_0), \ldots, (s_n, e_n)\rangle$ *resulting from the execution of* π *matches the constraints* $\mathcal{C}_t$*.*

It remains to define how to transform a temporal task network into another one by using method decomposition in order to obtain an executable task network, and what represents a temporal task network solution.

[4] Rational numbers are used in the definition from Fox and Long (2003). However, integers should be used for dates, because using rationals without adding further conditions can yield to an undecidable planning problem (Barringer, Kuiper, and Pnueli 1986).

Definition 10 (Decomposition)**.** *A task network* $w_1 = (\mathcal{I}^1, \mathcal{C}^1)$ *is decomposed into a new task network* $w_2 = (\mathcal{I}^2, \mathcal{C}^2)$ *by refinement by a method* $m = \big(name(m), task(m), (\mathcal{I}^m, \mathcal{C}^m)\big)$ *if and only if there exists* $i \in \mathcal{I}^1$ *such that* $\alpha(i) = task(m)$ *and* $\mathcal{I}^2 = (\mathcal{I}^1 - \{i\}) \cup \mathcal{I}^m$*, and*

$$\begin{aligned}\mathcal{C}^2_o = \; & \mathcal{C}^1_o \cup \mathcal{C}^m_o \\ & \cup \{start(t) \leq start(t') \mid t = \min\left(e_{start(\alpha(i))}, e_{start(\alpha(j))}\right), \\ & \quad t' = \max\left(e_{start(\alpha(i))}, e_{start(\alpha(j))}\right),\ j \in \mathcal{I}^m\} \\ & \cup \{end(t) \geq end(t') \mid t = \max\left(e_{start(\alpha(i))}, e_{start(\alpha(j))}\right), \\ & \quad t' = \min\left(e_{start(\alpha(i))}, e_{start(\alpha(j))}\right), j \in \mathcal{I}^m\}\end{aligned}$$

with $e_{start(\alpha(k))}$ *the start date of* $\alpha(k)$ *for* $k \in \mathcal{I}^1 \cup \mathcal{I}^2$*.*

$\mathcal{C}^2_v = \mathcal{C}^1_v \cup \mathcal{C}^m_v$, $\mathcal{C}^2_d = \mathcal{C}^1_d \cup \mathcal{C}^m_d$, $\mathcal{C}^2_t = \mathcal{C}^1_t \cup \mathcal{C}^m_t$

Definition 11 (Temporal Task Network Solution)**.** *Let* $\mathcal{P} = (\mathcal{D}, s_0, w_0, g)$ *be a planning problem with* $\mathcal{D} = (\mathcal{L}, \mathcal{T}, \mathcal{A}, \mathcal{M})$*. A task network* $w_s = (\mathcal{I}, \mathcal{C})$ *is solution to a temporal HTN planning problem* $\mathcal{P}$ *if and only if:*

- *There exists a sequence of decompositions from* w_o *to* w_s *resulting from the application of the methods* $\mathcal{M}$ *of* $\mathcal{D}$*;*
- w_s *is executable and the temporal sequence of states resulting from its execution starts with* s_0*, and achieves a state* $s \models g$*.*

4 Decomposition Constraint Semantics

Decomposition constraints are conditions that must be satisfied by all the states visited while executing a solution task network. They are expressed through temporal modal operators over first-order formulas involving state predicates, as in PDDL. The semantics of the decomposition constraints can be formally specified similarly to the approach taken by Gerevini and Long (2005). Let $w = (\mathcal{I}, \mathcal{C})$ be a ground task network, a state s_0 and a primitive temporal sequence of tasks $\pi = \langle (t_0, e_0, \delta_0), \ldots, (t_n, e_n, \delta_n) \rangle$ with $t_0 = \alpha(i_0), \ldots, t_n = \alpha(i_n)$ resulting from the decomposition of w. We denote by $\tau = \langle (s_0, e_0), \ldots, (s_n, e_n) \rangle$ the temporal sequence of states produced by the execution of π in s_0, ordered according to its happening events, with $i \leq 0 \leq n$. Decomposition w satisfies a constraint $(at\ e\ \phi) \in \mathcal{C}_t$ iff $\exists (s_k, e_k) \in \tau$ such that $s_k \models \phi$. Note that it is required that every temporal constraint of the form $(at\ e_i\ \phi) \in \mathcal{C}_t$ to be defined for $0 \leq i \leq n$ so as to avoid defining constraints that are out of the scope of the temporal task network w. Each decomposition constraint defined in HDDL 2.1 can be rewritten in terms of constraints of the form $(at\ e\ \phi)$. The proposed HDDL extension distinguishes two types of decomposition constraints: (1)the *temporal decomposition constraints* that define the constraints that must hold at specific happening events whose semantics is based on the plan trajectory constraints from PDDL 3.0 (Gerevini and Long 2005) and (2) the *classical decomposition constraints* (*before*, *after*, *between*) used in HTN planning (Erol, Hendler, and Nau 1994) to represent constraints between tasks.

For the temporal decomposition constraints, we suggest to keep the same syntax and semantics as introduced in PDDL 3.0 to mantain a language consistency between different versions. For classical decomposition constraints, it can be shown that they can be expressed in terms of the former, as are method precondition semantics (*at start*, *at end*, *overall*).

5 Discussion and Conclusion

We have introduced the main features of a HDDL version for hierarchical temporal planning tasks. This aims at bridging the gap between HTN planning and real world applications, where temporal features like concurrent actions, coordination, and hierarchical distribution of tasks, are prominent. In our view, the absence of a unified language for temporal and numerical constraints in PDDL's evolution is an obstacle that needs addressing. Although the community has developed various approaches to model complex planning problems, including temporal features and hierarchical task decomposition, the variety of language solutions hinders the development of common tools and solvers.

Action Notation Modeling Language (ANML) (Smith, Frank, and Cushing 2008) has an expressivity close to what we seek here. In ANML effects that happen at time intervals during an action duration can be specified. This can also be represented in HDDL 2.1 by using constraint semantics and dividing actions with intermediate effects into separate durative actions.

Some planners propose benchmarks where ordering constraints are delayed, e.g., FAPE (Dvorák et al. 2014). For instance in ANML it is possible to specify that an action must happen at least some amount of time after the end of a previous action. With the proposed syntax and semantics, such expressivity can be reached by using auxiliary tasks that decompose in a durative primitive task (of the desired duration) with no effects.

Time sampling represents another open question for this extension of HDDL with time. Basically, two approaches exist. Sampling can be either constant —when time is divided into regular-spaced discrete steps— or with variable time steps instantiated when effects and preconditions are applied. The latter can benefit from the Simple Temporal Problem formalism to model the temporal features of the plan and to include timed initial effects, and Interval Temporal Logic can be used to define truth of formulas relative to time intervals, rather than time points (Bresolin et al. 2014).

In order to be fully compatible with PDDL 3.0 features, the language HDDL 2.1 needs to include axioms and preferences, besides the associated parsing and validation tools. For this reason, the present work has to be seen as a baseline for the planning community to build upon. In fact, many applications require features that have been (on purpose) omitted in this paper. Most importantly, we did not explicitly report a syntax, and we did not allow to define delays with point algebra for ordering constraints. Other features are simply not detailed here, e.g., continuous effects. Such language elements are left as natural extension of this paper for future work.

References

Abdulaziz, M.; and Koller, L. 2022. Formal Semantics and Formally Verified Validation for Temporal Planning. In *AAAI*, 9635–9643.

Alford, R.; Kuter, U.; and Nau, D. 2009. Translating HTNs to PDDL: A Small Amount of Domain Knowledge Can Go a Long Way. In *IJCAI*, 1629–1634.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. 2016.Hierarchical Planning: Relating Task and Goal Decomposition with Task Sharing. In *IJCAI*, 3022–3029.

Barringer, H.; Kuiper, R.; and Pnueli, A. 1986. A really abstract concurrent model and its temporal logic. In *ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 173–183.

Behnke, G. 2021. Block Compression and Invariant Pruning for SAT-based Totally-Ordered HTN Planning. In *ICAPS*, 25–35.

Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; Pellier, D.; Fiorino, H.; and Alford, R. 2019. Hierarchical planning in the IPC. In *ICAPS Workshop on the International Planning Competition*.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT-Totally-ordered hierarchical planning through SAT. In *AAAI*, volume 32.

Behnke, G.; Höller, D.; and Biundo, S. 2019. Bringing order to chaos–A compact representation of partial order in SAT-based HTN planning. In *AAAI*, volume 33, 7520–7529.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On Succinct Groundings of HTN Planning Problems. In *AAAI*, 9775–9784.

Behnke, G.; Pollitt, F.; Höller, D.; Bercher, P.; and Alford, R. 2022. Making Translations to Classical Planning Competitive with Other HTN Planners. In *AAAI*, 9687–9697.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning - One Abstract Idea, Many Concrete Realizations. In *IJCAI*, 6267–6275.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In *IJCAI*, 480–488.

Bit-Monnot, A.; Smith, D.; and Do, M. 2016. Delete-Free Reachability Analysis for Temporal and Hierarchical Planning. In *ECAI*, volume 285, 1698–1699.

Bresolin, D.; Della Monica, D.; Montanari, A.; Sala, P.; and Sciavicco, G. 2014. Interval temporal logics over strongly discrete linear orders: Expressiveness and complexity. *Theoretical Computer Science*, 560: 269–291.

Broxvall, M.; and Jonsson, P. 2003. Point algebras for temporal reasoning: Algorithms and complexity. *Artif. Intell.*, 149(2): 179–220.

Cushing, W.; Kambhampati, S.; and Weld, D. S. 2007. When is temporal planning really temporal? In *IJCAI*, 1852–1859.

Dvorák, F.; Barták, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. Planning and Acting with Temporal and Hierarchical Decomposition Models. In *ICTAI*, 115–121.

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN Planning: Complexity and Expressivity. In *AAAI*, 1123–1128.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Int. J. of Artif. Intell. Res.*, 20: 61–124.

Georgievski, I.; Nizamic, F.; Lazovik, A.; and Aiello, M. 2017. Cloud Ready Applications Composed via HTN Planning. In *IEEE Conference on Service-Oriented Computing and Applications*, 81–89.

Gerevini, A.; and Long, D. 2005. Plan constraints and preferences in PDDL3 - the language of the fifth International Planning Competition. Technical Report 2005-08-07, Department of Electronics for Automation (Imperial College).

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press. Chap. 4.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5-6): 503–535.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *AAAI*, 9883–9891.

Höller, D.; and Bercher, P. 2021. Landmark Generation in HTN Planning. In *AAAI*, 11826–11834.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2019. On Guiding Search in HTN Planning with Classical Planning Heuristics. In *IJCAI*, 6171–6175.

Mcdermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Menif, A.; Jacopin, E.; and Cazenave, T. 2014. SHPE: HTN Planning for Video Games. In *Workshop on Computer Games*, 119–132.

Milot, A.; Chauveau, E.; Lacroix, S.; and Lesire, C. 2021. Solving Hierarchical Auctions with HTN Planning. In *ICAPS Workshop on Hierarchical Planning*.

Pellier, D.; and Fiorino, H. 2018. PDDL4J: a planning domain description library for Java. *J. Exp. Theor. Artif. Intell.*, 30(1): 143–176.

Ramoul, A.; Pellier, D.; Fiorino, H.; and Pesty, S. 2017. Grounding of HTN Planning Domain. *Int. J. of Artif. Intell. Tools*, 26(5).

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13): 1031–1080.

Schreiber, D. 2021. Lilotane: A Lifted SAT-based Approach to Hierarchical Planning. *J. of Artif. Intell. Res.*, 70: 1117–1181.

Schreiber, D.; Pellier, D.; Fiorino, H.; and Balyo, T. 2019. Tree-REX: SAT-Based Tree Exploration for Efficient and High-Quality HTN Planning. In *ICAPS*, 382–390.

Shivashankar, V.; Alford, R.; and Aha, D. 2017. Incorporating Domain-Independent Planning Heuristics in Hierarchical Planning. In *AAAI*, 3658–3664.

Smith, D.; Frank, J.; and Cushing, W. 2008. The ANML Language. In *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling*.