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.

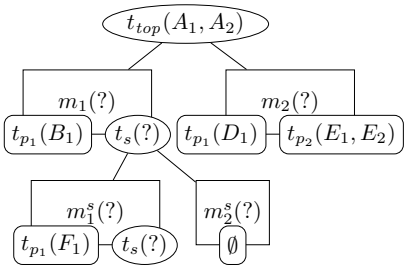
The propagation of arguments upwards would require defining a parent task for the whole hierarchy, which is difficult in the presence of recursive task definitions. To solve this issue, the function EXTRACT SUBHIERARCHIES (Alg. 1, line 2) takes each unique abstract task symbol t and convert it into a 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.

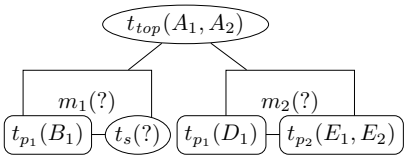
Algorithm 1: Parameter Superset Generation

```
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 = ∅
```

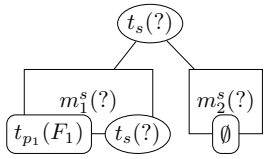
This decomposition into subhierarchies has the added benefit of permitting the parameter extraction process to work simultaneously with demonstrations for different top level tasks $t_{top} \in \mathcal{T}_{dem}$.



(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** $m \in \mathcal{M}_t$ **do**
2: **for all** $\hat{p} \in \text{args}(\text{SUBTASKS}(m))$ **do**
3: $\text{args}(m) \leftarrow \text{args}(m) \cup \{\hat{p}\}$
4: **if** $m \notin \hat{\mathcal{M}}_p$ **then**
5: $\hat{p}' \leftarrow (p, \hat{\mathcal{M}}_p \cup \{m\})$
6: $\text{args}(t_h) \leftarrow \text{args}(t_h) \cup \{\hat{p}'\}$
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:

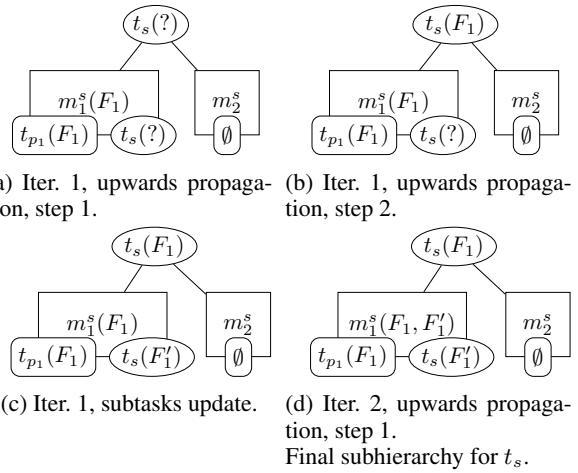
1. Parameterize all the non-recursive subtasks.
2. For each recursive subtask instantiation, choose whether the parameters are the same as the parent task or not.

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

1: **for all** $t_s \in \text{SUBTASKS}(h_{sub})$ **do**
2: $t \leftarrow \text{sym}(t_s)$
3: $\mathcal{P}_{new}^t \leftarrow \{\hat{p} \mid \hat{p} \in \text{args}(t) \wedge \hat{p} \in \mathcal{P}_{new}\}$
4: **for all** $\hat{p} \in \mathcal{P}_{new}^t$ **do**
5: $\text{args}(t_s) \leftarrow \text{args}(t_s) \cup \hat{p}$
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.

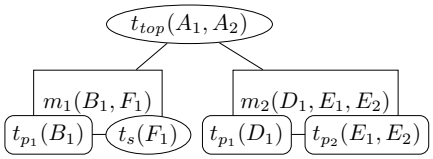
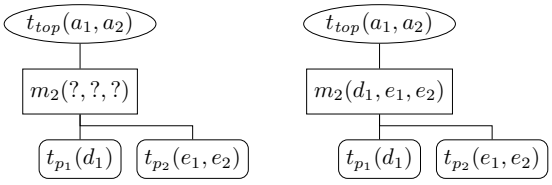


Figure 3: Extracted parameters for t_{top}

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.

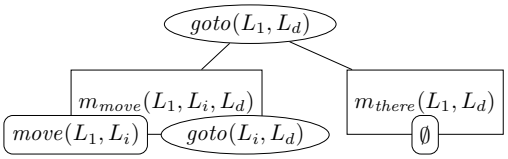
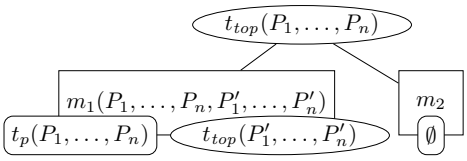
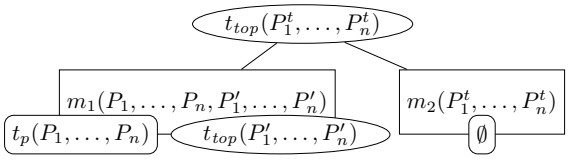


Figure 5: Subhierarchy for a $goto$ pattern. Preconditions omitted for clarity.

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.

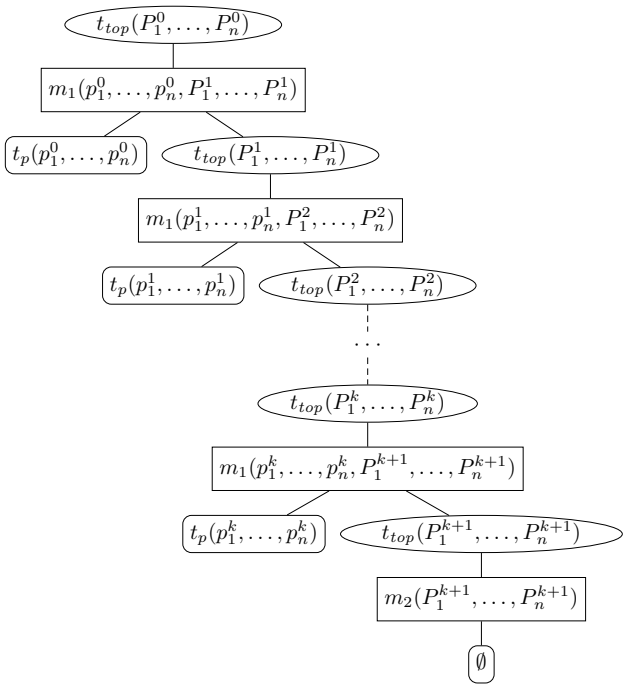


Figure 7: Generic decomposition tree for a recursive hierarchy.

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.

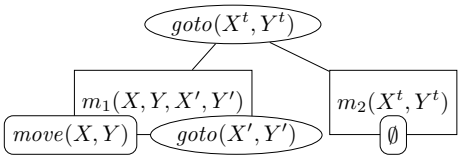
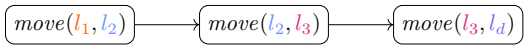
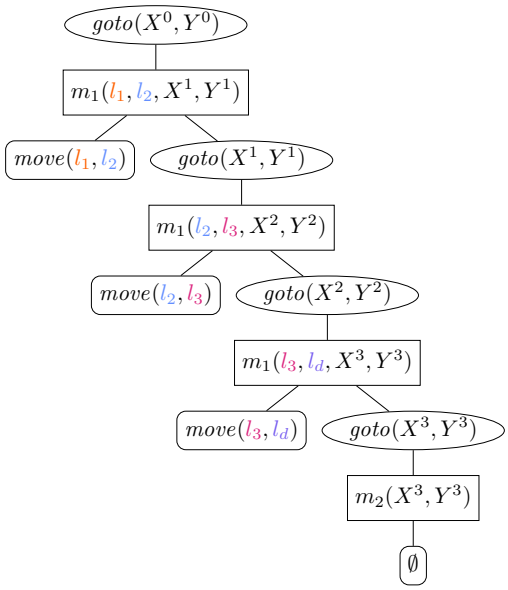


Figure 8: Extracted subhierarchy for a $goto$ task before recursion processing.

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.

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

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

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

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

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

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

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

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.

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

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.

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

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

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:

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

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

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

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:

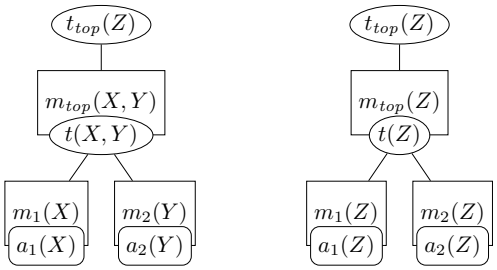
$$\forall h_{sub} \in \mathcal{H}_{subs}, \forall t_s \in \textsc{Subtask}(h_{sub}),$$
$$\forall (P_s, P_h) = \arg\mathrm{s}(t_s) \times \arg\mathrm{s}(t_h)$$
For any ground instantiation of h_{sub} in $d \in \mathcal{D}$ and associated grounding p_s of P_s (resp. p_h of P_h):
$$\begin{cases}\textsc{Soft}(P_s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

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.

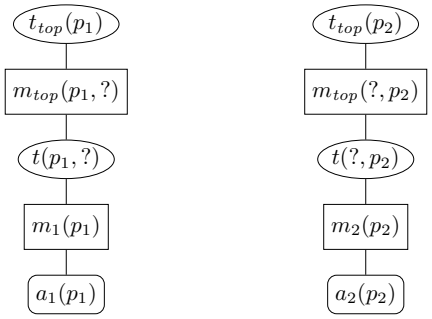


Figure 11: Example decomposition trees for downward propagation.

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 $p \in \arg\mathrm{s}(\mathcal{H}_{subs})$ **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
  $\arg\mathrm{s}(\mathcal{H}_{subs}) \leftarrow \arg\mathrm{s}(\mathcal{H}_{subs}) \setminus \mathcal{P}_{drop}$
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

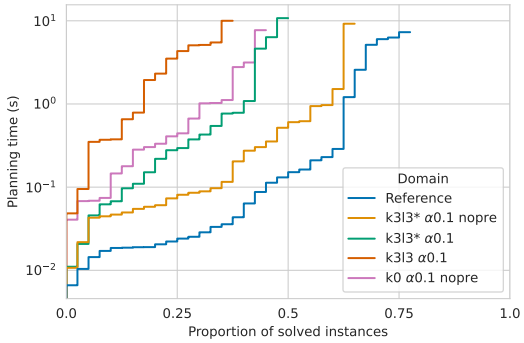


Figure 12: Planning time for models learned the proposed approach for parameter learning using the LILOTANE planner.

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.