Learning Operational Models from Demonstrations: Parameterization and Model Quality Evaluation

Philippe Hérail, Arthur Bit-Monnot

Learning Operational Models from Demonstrations: Parameterization and Model Quality Evaluation

Philippe Hérail, Arthur Bit-Monnot

LAAS-CNRS, Université de Toulouse, INSA, CNRS, Toulouse, France
philippe.herail@laas.fr, abitmonnot@laas.fr

Abstract

When acting in non-deterministic environments, autonomous agents must balance between long-term, complex goals with unpredictable events and reactive behavior. In this context, hierarchical operational models are attractive in that they allow the execution of complex behavior either in a purely reactive fashion or guided by a planning process. Just like for HTN models with which they share most characteristics, one key bottleneck in the exploitation of operational models is their acquisition.

In this paper, we introduce an algorithm for learning hierarchical operational models from a set of demonstrations. Given an initial vocabulary of tasks and some demonstrations of how they could be achieved, we present how each task can be associated to a set of methods capturing the operational knowledge of how it can be achieved. We present the structure of the learned models, the algorithm used to learn them as well as a preliminary evaluation of this algorithm.

Introduction

To allow an autonomous agent to operate in its environment, we can differentiate two family of approaches, split by the way they select their next action: reactive or deliberative. While a reactive approach may cover several distinct acting techniques, these often present the common characteristic of being fast to act but short-sighted. Meanwhile, deliberative approaches typically rely on planning to consider long-term impacts of a decision, at the cost of increased deliberation time. In order to find a compromise between both these approaches *hierarchical operational models* (Ghallab, Nau, and Traverso 2014) have been developed along with acting engines such as in (Patra et al. 2021). This combination allows to have models that distinguish between different ways to achieve a given task through methods. These hierarchical models allow the agent to act either reactively, selecting one method without any reasoning about the future, or to make more elaborate choices. Furthermore, the hierarchical aspect allows to reduce the possibilities that need to be considered at any given time, similarly as in hierarchical planning.

Even though these hierarchical models are a formalism that allows to act more efficiently while remaining interpretable by human engineers, it is cumbersome to design such models from scratch. This difficulty stems from the quickly exploding number of possible contexts that need to be considered when carrying out even basic tasks in a simple environment. To address this issue, we intend to allow the agent to learn such operational models from previously observed execution traces, and in particular the ones resulting from a tutor's demonstration.

The goal of such a learning system would be to be able to solve any previously demonstrated tasks through a solution of at least equivalent quality to the demonstrated one. It should also be able to generalize the demonstrations to solve new unseen tasks, or previously demonstrated tasks in a new environment. This is done by learning, for any given task in the considered domain, a set of methods that achieve the high-level objectives associated to the task. This set of methods should cover all possible ways of achieving this task with the exception of clearly suboptimal ways. Any method should be associated with a validity scope that define whether it is *applicable* in a given state. When applicable, it should achieve the task.

Intuitively, if a learned operational model has these desirable properties, an acting engine facing a task to achieve could pick any applicable method and have the guarantee that it will fulfill the objectives associated to the task. While this might lead to suboptimal behavior, it should be possible for an automated planner to guide the choice of the method to obtain the optimal behavior.

The objective of this paper is to present a method for building hierarchical operational models based on past experiences. The experience takes the form of execution traces that can be the result of the agent's own activity or of demonstrations by a tutor. The agent is assumed to have a fixed set of primitive capabilities and to act in a Fully Observable Non-Deterministic (FOND) environment.

Related Work

Over the years, several approaches have been developed to learn hierarchical models.

HTN-MAKER (Hogg, Muñoz-Avila, and Kuter 2008) learns Hierarchical Task Networks (HTNs), for use in fully-observable deterministic domains from execution traces. This approach uses tasks annotated with preconditions and postconditions so as to extract sequences that allow to achieve the postconditions starting from a state where the preconditions hold. The method was later extended by Hogg, Kuter, and Muñoz-Avila (2009) to handle non-deterministic

domains, through the use of a right recursive structure instead of sequences. However, the models learned are restricted to a limited task and method structure, resulting in rather flat hierarchies, which limits the guidance offered to an agent using them for planning.

HTNLearn (Zhuo, Muñoz-Avila, and Yang 2014) also learns HTNs with similarly annotated tasks, through converting the learning problem into one of constraint satisfaction, building the constraints from, e.g., the ordering of tasks within the examples and the state preceding the application of a method. The problem is then solved using a MAXSAT approach and the solution converted back into an HTN model. While this approach does not handle nondeterminism, it supports partially observable states. A similar approach is even able to use partial, disordered input traces (Zhuo, Peng, and Kambhampati 2019).

Recently, the learning of Hierarchical Goal Networks (HGNs) structure instead of HTNs, for nondeterministic domains, has been proposed as a preliminary work by Fine-Morris and Muñoz-Avila (2019), leveraging a vector representation of the states and unsupervised learning procedures to learn such networks while limiting the burden of annotating demonstration data.

Due to their similarities with HTNs, some work aiming at learning grammars is relevant and in particular the work on learning Combinatory Categorial Grammars (CCGs) for plan and goal recognition (Geib and Goldman 2011; Kantharaju, Ontañón, and Geib 2019). While the learned CCGs are not always practically usable, the authors propose several ideas for extracting interesting patterns from a set of execution traces.

The algorithm for learning probabilistic primitive action models (i.e. not hierarchical) presented by Pasula, Zettlemoyer, and Kaelbling (2007) develop interesting solutions for handling the specificities of FOND environments.

Work has been done on the automated learning of Behavior Trees (BTs), a common framework for implementing hierarchical, reactive operational models. Colledanchise, Parasuraman, and Ögren (2018) and Zhang et al. (2018) develop techniques to efficiently apply genetic programming to these structures.

Learning Problem

Operational Model

For an agent acting in its environment, an operational model, as defined by Ghallab, Nau, and Traverso (2014), represents an agent's knowledge about how to carry out a given activity in its environment. In this work, we are specifically interested in nondeterministic, fully observable environments.

We define an operational model O as an HTN-like structure which can be written as a tuple $O = (T, A, M)$ where T is a set of abstract tasks, A a set of primitive actions and M a set of possible methods decomposing the tasks $t \in T$ into subtasks $\{t_d \mid t_d \in \{T \cup A\}\}$. We consider the tasks to be possibly annotated with postconditions, similarly as Hogg, Muñoz-Avila, and Kuter (2008).

A primitive action $a \in A$ models the basic acting capabilities of the agent, and represent directly executable primitives. They are represented using an identifier and a set of parameters, such as $a = \text{action\_name}(\text{arg}_1, \ldots, \text{arg}_n)$. We do not assume any knowledge on the preconditions and effects of a primitive action. Furthermore, as we consider a non-deterministic environment, there is no guarantee that applying an action twice in the same state will produce an identical result.

An abstract (or non-primitive) task $t \in T$ is defined as a tuple $t = (Post_t, M_t)$, where $Post_t$ are the postconditions of the task, that is the predicates that must hold after executing t for it to be considered a success. These are especially important in nondeterministic domains. A task without postconditions is considered successful whenever one of its method has been executed without failure. M_t is the set of methods decomposing t.

A method $m \in M_t$ is a tuple $m = (Pre_m, N_m)$, where Pre_m are the preconditions of the method, and N_m is a task network defining a way to decompose t into subtasks. This task network represents a way to advance the task t towards its intended effects, in the case of a task with postconditions, or a way to achieve t if $Post_t = \emptyset$. A method is applicable in a given state s iff its preconditions hold in this state. For the sake of simplicity, we will assume that the set N_m of subtasks is totally ordered.

We define an acting problem as an initial task network N_p, representing the activity we wish to carry out, as well as an associated environment and a starting state s_i described by a set of boolean state variables. When trying to solve an acting problem p, we associate it to an operational model O that will be used to select the actions to execute. We consider that at any instant, the current state s is fully observable and that it only evolves when a primitive action is executed (i.e. there are no exogenous events). This evolution may however be nondeterministic.

Acting with an Operational Model

To act with an operational model, abstract tasks are refined down to a sequence of executable primitive actions. This refinement is achieved through the methods associated with the tasks: each time the agent must choose an action to execute, the current best applicable method decomposing the task at hand is chosen. The process is iteratively repeated until a sequence of tasks starting with a primitive action is obtained. This sequence is then executed until a non-primitive task is encountered, at which point the same process is applied again. The choice of the best method may be done either reactively or deliberatively. In the first case, the best method is chosen greedily according to some metric, while in the second one an acting engine, such as the one described by Patra et al. (2021), may use planning techniques to select a method considering the long term implications of the actions.

During the refinement of a task t using the method m starting in a state s, several failure types may occur:

1. A primitive action fails to be executed, causing the whole parent method to be considered a failure.
2. A non-primitive descendant t' from m has no applicable method. Then, the parent method of t' is considered to

have failed.

3. t can be completely refined, but the postconditions of t do not hold in the final state. This case is obviously valid only for a task t such that $Post_t \neq \emptyset$.

In the first two cases, when a method m decomposing a task t is detected as having failed, then some retrial strategy must be used to continue acting. A simple strategy could be to try another applicable method m' applicable in the current state s. If no such method is available in the hierarchy, then the failure is to be propagated upwards to the parent task t, until we reach either a task with an applicable method or the root task, the latter case leading to a failure to solve the acting problem.

Learning of an Operational Model

Inputs to the Learning Problem For the learning problem itself, we consider as input a fixed set A of primitive actions as well as a vocabulary of non-primitive tasks T_I.

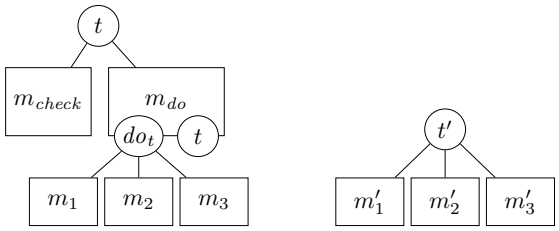
Recall that each primitive action has the form action_name(param$_1$, ..., param$_n$) and corresponds to a primitive directly executable by the agent. For a non-primitive task $t_I \in T_I$, we assume its postconditions $Post_{t_I}$ to be non-empty, as they are required to assess the success of the task execution in the agent's nondeterministic environment. Note that these postconditions might be learned independently of the methods, which is out of the scope of this paper.

Then, for each task $t_I \in T_I$, the agent is given a set D_{t_I} of demonstration traces from the tutor. Each trace $d \in D_{t_I}$ is an alternating sequence of states and tasks (either primitive or non-primitive), starting from a given initial state and ending in a final state in which the task t_I has been successfully achieved. d is considered optimal and maximally abstract with regard to the initial task vocabulary: for every demonstrated task, no other more abstract task from the initial vocabulary T_I may be used to abstract a subsequence of d, and each demonstration is optimal according to a chosen metric. For a case where actions are uniform in cost, one may naturally consider the total number of primitive actions required to achieve t_I as the optimality metric.

Learner Objectives Given a learned operational model and an acting problem, we say that the operational model solves the problem if, when used by the acting engine, it allows to refine a given task network into an executable sequence of primitives. Considering a successful solution to the problem, we define the soundness metric as the number of failed method execution (i.e. number of times we had to resort to recovery behavior) within the execution, to allow the evaluation of the quality of the learned preconditions of the methods. We also define the efficiency metric as the ratio between the cost of the executed behavior and the cost of executing an optimal model.

These metrics have been defined for a single acting problem. To assess the generalization capabilities of the model, we should consider a set of acting problems not encountered during learning. We define three metrics over this test set, namely the ratio of solved problems (coverage), the average number of method failures (average soundness), and the average efficiency.

Finally, to evaluate the algorithm as a whole, we should consider a set of demonstration and a set of acting problems. The performance of the algorithm is defined as the average performance of the model produced from the demonstrations on the acting problems, as defined in the previous paragraph.



(a) Task with postconditions. (b) Task without postconditions.

Figure 1: Structure of a task, with or without postconditions. The arguments of tasks and methods are omitted for clarity.

Approach to Model Learning

In our approach, we develop operational models designed to handle nondeterministic environments. To this end we distinguish tasks with and without postconditions, as shown in figure 1, presenting the structural differences a task t with postconditions and a task t' without.

In the case of t, ($Post_t \neq \emptyset$, figure 1a), we observe that t has two methods, m_{check} and m_{do}, and a right recursive structure. The former has no subtask, and its preconditions correspond to the postconditions of t, while the latter has no preconditions and two subtasks, do_t and t, recursively. The methods m_i of do_t encode different possible ways to act for the agent, with the intent to try and achieve t.

When decomposing t, m_{check} has priority over m_{do}, and is used to assess whether the goal associated with t has been achieved. We can note that t will thus be decomposed recursively until its associated goal is achieved, even in the case of unexpected events, provided there is a valid m_i to handle it, as in HTN-MAKER (Hogg, Kuter, and Muñoz-Avila 2009). Therefore, when decomposing t, in the case where its postconditions do not hold, we will choose one of the available method of do_t, refine and execute it, and then try to refine the original instance of t again, hoping to have achieved the intended effects. It should be noted that a method of a task may either represent a way to completely achieve it, or merely to advance it (or recover from mistakes), leveraging the recursive call to finally achieve the original endeavor.

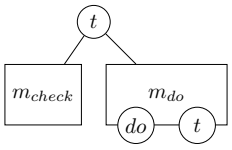
In the case of t', a task without postconditions ($Post_{t'} = \emptyset$, figure 1b), we have a task with a more standard structure, as a method will then always be considered successful if it is refined and executed to its completion. Indeed, it would not be possible to decide if the task should be recursively retried or not.

Requirements of Model Selection

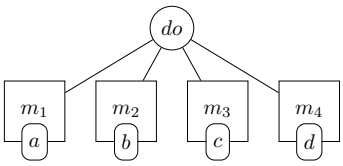
Let us now give an initial intuition about the shape of models that could be learned and the implication for the learn-

$t \rightarrow a\ b\ c$
$t \rightarrow a\ b\ d$

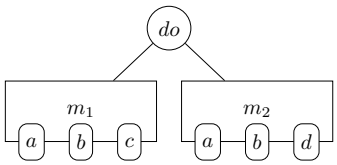
(a) Available demonstrations, showing that t was once achieved with the $a\ b\ c$ action sequence and once with the $a\ b\ d$ action sequence.



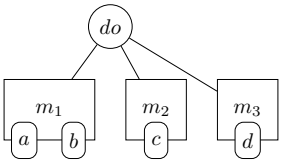
(b) Common right recursive structure, missing the refinement of the do task. A handful of possible models for the do task are shown in the subsequent figures.



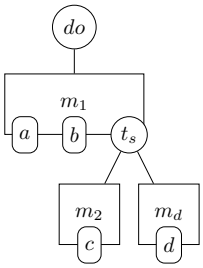
(c) Generic model where the actor might pick any of the primitive actions and rely on the recursive call in m_{do} to continue if needed.



(d) Model where each demonstration is fully encoded into a dedicated method.



(e) Intermediate model the common $a\ b$ sequence is grouped. It relies on the recursive call to t in m_{do} to produce a full sequence.



(f) Model where the $a\ b$ sequence is shared, requiring a synthetic task t_s

Figure 2: Illustration of the possible structures of the learned model for a simple learning task with two demonstration of how to perform a task t. Note that for conciseness the parameters or preconditions or the task and methods are omitted.

ing process. Figure 2 presents several possible models (figures 2c-2f) that could be generated based on two example sequences (figure 2a). All candidate models share the same recursive structure that we just saw (figure 2b) and only differ in the refinement of the do task.

The first one (2c) allows the choice of any of the four primitive actions $\{a, b, c, d\}$, each placed in a specific method. This model relies on the recursion to repropose the same choice until the task's postconditions are achieved. While this model allows building any sequence of actions it does not help the agent towards a meaningful sequence based on demonstrations. The second model (2d) takes the opposite approach and records each known trace into a method. This model is obviously strongly tied to the demonstration set and would fail to generalize to new problems. In between these two extreme, we have the models (2e) and (2f) that take different options to abstract common subsequences. The former encodes the repeated $a\ b$ sequence in a single method and relies on the recursive call to complete the sequence. The latter delays the choice between c and d to after the execution of a and b, using a synthetic task t_s.

These four models are just a handful of examples among the many possible models that could be generated. Denoting as Θ the set of possible models, the objective of a learning system is to find, or at least approach, the optimal model $\theta^* \in \Theta$

$$\theta^* = \arg\min_{\theta \in \Theta} cost(\theta)$$

where $cost(\theta)$ is a function that measures the cost of a particular model and should typically account for the size of the model as well as its capacity to solve both demonstrated and unseen problems. With this in mind we now turn our attention to the characterization of the set of possible models Θ. In a later section, we will propose a cost function to evaluate the models.

Generation of Candidate Operational Models

At a high level, the goal of the learning problem is to generate a model where some subtasks group together behaviors that happen repeatedly, with a sensible parameterization of methods depending on the current task, as well as reasonable preconditions to limit the search effort of the acting engine. It is easy to imagine an iterative process for this: we may extract some subsequence as a new task t, with a single method m_1 consisting of this subsequence. Once this is done, we may find another subsequence that achieves a similar goal, but using different tasks, thus allowing us to add a new method m_2 to t. Next, we may be able to extract a new task t' using t as a subtask of this method. This process is to be repeated until it produces a model that become too complex, not improving the score defined in the previous section.

Devising such an iterative learning process is however error-prone as it is easy to get stuck in a local minimum due to bad decision in the early stages of learning. Because the quality of a model depends not only on its structure, but also on its parameterization, in order to efficiently develop this latter part of the learning algorithm, we developed a method for generating a large number models so as to conduct a preliminary evaluation and relegate a more efficient exploration of the set of possible models to future work.

Considering a set of primitive actions and a set of demonstrations, it is easy to imagine a way to generate a number of operational models that can be used to achieve any of the given demonstration. For instance, we could start with a basic flat model, and add any number of possible subtasks whose methods correspond to arbitrary sequences of primitive actions, with possible duplications. This model can al-

ways fall back to choosing one of the single primitive actions if no subtask's method can be used, and therefore will always remain valid. As an infinite number of possible sequences can be used as subtasks, an infinite number of such models can be elicited, thus foregoing an exhaustive generation procedure. We therefore chose to restrict ourselves to a certain structure, described below, so that we could generate models exhaustively within this limited search space.

Considering an initial set of primitive actions $\{a, b, c, d\}$, we try and find all the ways to partition it, i.e. $P = (\{\{a\}\{b\}\{c\}\{d\}\}, \{\{a, b\}\{c\}\{d\}\}, \{\{a, b, c\}\{d\}\}, \dots)$. For each new set in P, we recursively apply the same process to each subpartition. The resulting sets will have a format such as $\{\{\{a\}\{b, c\}\}\{d\}\}$ each of which corresponding to a particular model (e.g. see the one presented in figure 3 for this particular example). To build such a model, we consider each level, starting from the first one (here, this first level contains the subpartitions $\{\{a\}\{b, c\}\}$ and $\{d\}$), considering that it shows different ways to decompose some task t that has been demonstrated. We then consider the subpartitions. If it is not further subpartitioned (as $\{d\}$ here), we create a method refining t down to the primitive action contained. Otherwise, we create a subtask st and a method that refines t into st and recursively apply the same procedure, considering that this subpartition shows ways to decompose st.

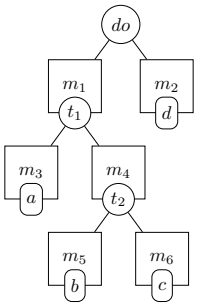


Figure 3: Model corresponding to the $\{\{\{a\}\{b, c\}\}\{d\}\}$ partitioning.

With this procedure, we will obtain models that are limited, as they will, e.g., never use the same primitive to decompose two different subtasks, nor will they contain methods that decompose as sequences of tasks. However, this generation process produces a number of models that scales exponentially with the number of primitive tasks given as input, due to the number of possible subpartitions of a set, therefore quickly making the use of the generated model set for evaluating our algorithm impractical. Adding more possible model structures would only worsen this issue, thus warranting the use of some local search procedure.

While this generation procedure allows us to generate new model structures, we still need to parameterize their tasks and methods to evaluate them properly on the existing demonstration set, and to eventually use the corresponding models for acting. This parameterization as well the extraction of methods' preconditions is detailed in the next section.

Parameters & Preconditions Extraction

For each method in the model, we need to identify the parameters that should be passed to its subtasks. When considering the abstract tasks, we need to distinguish two cases:

1. The set T_I of tasks given as the starting vocabulary for which arguments and postconditions are known.
2. The tasks that are inferred during learning, by grouping common subsequences for example, for which arguments must be extracted. This second type of tasks are called *synthetic tasks* and their set is called T_S. For these tasks, we do not consider the identification of intended effects in this work.

Process Overview Parameterization is a bottom-up, iterative process that selects a task or method whose subcomponents are already parameterized (i.e. their parameters are given or were previously identified). Those subcomponents are analyzed to identify the parameters of the current task/method, which in turn will enable the analysis of tasks/methods higher-up in the hierarchy.

1. We first consider any method m whose subtasks are all parameterized. For each such method we:
   - Identify its parameters based on the parameters of its subtasks
   - Extract its preconditions by considering each state that precedes an instance of the method in the demonstrations.
2. For any synthetic task $t \in T_S$ whose methods are all parameterized (as a result of the previous step), we identify the parameters of t based on the ones of its methods.
3. For any initial task $t \in T_I$ whose methods are all parameterized (as a result of the previous step), we associate the parameters of t to the parameters of its methods.

We repeat this process until all tasks and methods are parameterized, resulting in a bottom up identification of the parameters of the tasks and methods.

For this process, we need to know how our model would have been used if it had to generate the demonstrations, while not having yet access to its parameterization. To this end, we use the technique developed for HTN plan verification (Höller et al. 2021) to obtain such a trace from an HTN planner. To use this strategy with a candidate model O_c, we generate an HTN model by removing every argument and method precondition. We also replace each instance of a primitive action a by a new task a_{obs} whose methods each correspond to a single observation of an instance of a in the currently considered demonstration d. Each such method has a single precondition: being currently at the right point in the sequence. This means the method containing the 6^{th} observation in the demonstration can only be selected between adding the 5^{th} and 7^{th} to the final plan.

Identifying the Arguments of a Method from its Subtasks When extracting the arguments of a method, we

leverage the fact that we know the arguments of its subtasks as well as, for each method demonstration instance, their mapping to the ground arguments.

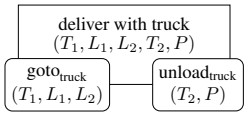
This allows us to unify the arguments of the subtasks as much as possible, by identifying the subtask arguments that are always bound to the same constant in each example. For instance, consider for a method m with $st_1(X_1, X_2)$ and $st_2(Y_1, Y_2, Y_3)$ as its subtasks. In this example, without any unification, the arguments of m could be $(X_1, X_2, Y_1, Y_2, Y_3)$, that is, the union of the arguments of its subtasks. If X_1 and Y_1 are bound to the same constant in each instantiation of the method (over all traces), we can unify them using a new variable Z_1. Therefore, the arguments of m become (Z_1, X_2, Y_2, Y_3), and its subtasks become parameterized as $st_1(Z_1, X_2)$ and $st_2(Z_1, Y_2, Y_3)$

To make the explanation clearer, let us consider an example *deliver with truck* method as presented in figure 4b, considering we also have two demonstration instances of this method, as presented in figure 4a. Since T_1 and T_2 are always bound to the same variable in each of the examples, we can introduce a new variable T, to replace each occurrence of T_1 or T_2. This leads to the method presented in figure 4c.

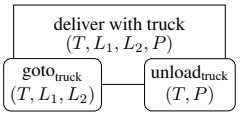
$$E_1 : \begin{cases} s_0^1 \rightarrow \text{goto}_{\text{truck}}(t_1, l_1, l_2) \\ \quad \hookrightarrow s_1^1 \rightarrow \text{unload}_{\text{truck}}(t_1, p_1) \rightarrow s_2^1 \end{cases}$$

$$E_2 : \begin{cases} s_0^2 \rightarrow \text{goto}_{\text{truck}}(t_2, l_3, l_4) \\ \quad \hookrightarrow s_1^2 \rightarrow \text{unload}_{\text{truck}}(t_2, p_1) \rightarrow s_2^2 \end{cases}$$

(a) Example traces associated to the method.



(b) Before unification.



(c) After unification.

$$\left\{\begin{array}{l} \text{package\_in}\,(t_1, p_1) \\ \text{truck\_at}(t_1, l_1) \\ \text{road}(l_1, l_2) \\ \text{sunny}(l_2) \\ \text{truck\_at}(t_2, l_2) \\ \begin{pmatrix} T \mapsto t_1 & P \mapsto p_1 \\ L_1 \mapsto l_1 & L_2 \mapsto l_2 \end{pmatrix} \end{array}\right\}_{s_0^1} \quad \left\{\begin{array}{l} \text{package\_in}(t_2, p_1) \\ \text{truck\_at}(t_2, l_3) \\ \text{road}(l_3, l_4) \\ \text{cloudy}(l_2) \\ \text{truck\_at}(t_3, l_2) \\ \begin{pmatrix} T \mapsto t_2 & P \mapsto p_1 \\ L_1 \mapsto l_3 & L_2 \mapsto l_4 \end{pmatrix} \end{array}\right\}_{s_0^2}$$

(d) State in the examples, with the mapping from method arguments to constants.

Figure 4: Example method structure before and after argument identification.

Extracting the Preconditions of a Method Having identified the arguments of a method m, we can extract its preconditions. As we know the mapping between method arguments and constants, we can easily filter the predicates in the states preceding every instance of m to keep only the ones fully specified by its arguments. Then, we take the intersection of these sets of predicates.

Consider the method *deliver with truck* from figure 4, and the starting states as defined in figure 4d. Then, using our knowledge of the mappings between the method parameters and the examples' constants, we can first exclude the last *truck_at* predicate from both examples, as at least one constant is not unified with a parameter. The preconditions of the method are therefore the intersection of the lifted and filtered states, i.e. $Pre = \{\text{package\_in}(T, P), \text{truck\_at}(T, L_1), \text{road}(L_1, L_2)\}$.

Associating the Arguments of a Method to the *Known* Arguments of the Task it achieves When learning methods for a task t in our initial vocabulary T_I, we have to map the (known) arguments of t to the arguments of its methods, which were extracted as described previously. These arguments are mapped in the same way as when unifying arguments for methods subtasks. Having some examples of decompositions of t, we follow the same logic as for the subtasks in a method: all parameters that are always bound the same value are unified.

Consider a task $t(Z_1, Z_2, Z_3)$ being decomposable with two methods, $m_1(X_1, X_2)$ and $m_2(Y_1, Y_2, Y_3, Y_4)$. Based on the traces of equation 1, we can replace X_1 and Y_1 with Z_1, X_2 with Z_2 and Y_3 with Z_3. We can therefore rewrite the methods' arguments as $m_1(Z_1, Z_2)$ and $m_2(Z_1, Y_2, Z_3, Y_4)$.

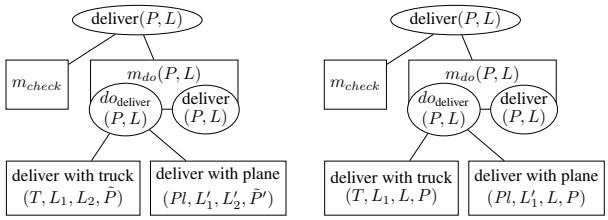
$$\begin{bmatrix} E_1 = t(a,b,c) : m_1(a,b),\ t(a,b,c) : m_2(a,c,c,d) \\ E_2 = t(x,y,x) : m_1(x,y) \\ E_3 = t(m,n,o) : m_2(m,n,o,q) \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} E_1 : \begin{cases} (X_1 \mapsto a), (X_2 \mapsto b) \\ (Y_1 \mapsto a), (Y_2 \mapsto c), (Y_3 \mapsto c), (Y_4 \mapsto d) \\ (Z_1 \mapsto a), (Z_2 \mapsto b), (Z_3 \mapsto c) \end{cases} \\ E_2 : \begin{cases} (X_1 \mapsto x), (X_2 \mapsto y) \\ (Z_1 \mapsto x), (Z_2 \mapsto y), (Z_3 \mapsto x) \end{cases} \\ E_3 : \begin{cases} (Y_1 \mapsto m), (Y_2 \mapsto n), (Y_3 \mapsto o), (Y_4 \mapsto q) \\ (Z_1 \mapsto m), (Z_2 \mapsto n), (Z_3 \mapsto o) \end{cases} \end{bmatrix} \quad (1)$$

To illustrate this with an example, let us consider the task described in figure 5a that represents the act of delivering a package P to a location L. It can be achieved either through a method delivering it via a truck (the same as described in the previous paragraph) or by a plane. Assuming we have some supporting examples, we can unify the arguments as presented in figure 5b.

Extracting the Arguments of a Synthesized Task from the Arguments of its Methods We now turn our attention to the identification of the arguments of synthetic tasks. This is done by leveraging the context of the instances of t to find method parameters that are always bound to the same constant in a given context.

Let us consider a new task $t \in T_S$, for which we do not know the arguments, and whose methods' arguments have been identified. We assume that t has a parent task t_p, possibly several levels higher in the hierarchy, whose arguments

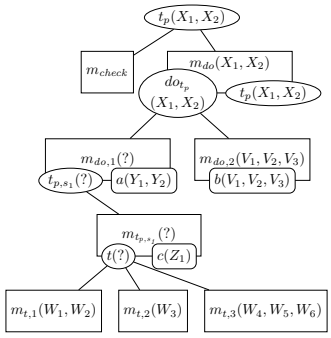


(a) Example task before unification.

(b) Example task after unification.

Figure 5: Example task before and after unification.

are known (i.e. $t_p \in T_I$). We note $\mathcal{C}_t$ the context surrounding an instantiation of t, defined as the variables bound from the instantiation of t_p and from all the definitive parts of the methods' instantiations on the path down to t. Figure 6a shows an example for such a task t, for which a context would be an instantiation of X_1, X_2, Y_1, Y_2, Z_1.



(a) Example task structure for contexts.

$$\text{args}(t) = \{a_1, a_2\}$$
$$a_1 \mapsto \{W_1, W_3\}$$
$$a_2 \mapsto \{W_4, W_5\}$$

(b) Example of a valid parameterization, assuming some supporting contexts.

Figure 6: Example method structure and parameterization.

After extracting all the contexts from the demonstrations, we wish to find a valid and relevant parameterization of t, such as the one of figure 6b. A valid parameterization is a mapping from $args(t)$ (the set of arguments of t) to $args(M_t)$ (the set of arguments of its methods) such that $\forall a_i \in args(t), \exists b_j \in args(M_t) : a_i \mapsto b_j$.

A single task parameter $a_i \in args(t)$ may map to several method arguments in $args(M_t)$. The core process for identifying the a_i arguments lies in identifying subsets of the methods' parameters that can be bound to the same task parameter. Intuitively, two methods parameters b_j and b_k can be bound to the same task parameter a_i iff:

- there is a context $\mathcal{C}_t$ where both b_j and b_k map to the same constant (positive example, noted $b_j \equiv_{\mathcal{C}_t} b_k$), and
- there is no context $\mathcal{C}'_t$ where b_j and b_k map to a different constant (counter-example, noted $b_j \not\equiv_{\mathcal{C}_t} b_k$).

More formally, a parameterization of a task can be seen as a set of grouped methods parameters $\mathcal{U} = \{U_0, \ldots, U_n\}$ where each U_i is the subset of $args(M_t)$ that the a_i parameter maps to. To be valid, a parameterization $\mathcal{U}$ must be such that:

$$b_j, b_k \in U_i \Rightarrow \begin{Bmatrix} \exists \mathcal{C}_t : b_j \equiv_{\mathcal{C}_t} b_k \\ \nexists \mathcal{C}_t : b_j \not\equiv_{\mathcal{C}_t} b_k \end{Bmatrix}$$
$$U_i \cap U_j = \emptyset \quad \text{if } i \neq j$$

Among all the possible valid parameterizations, the one leading to the smallest number of task arguments and containing the largest total number of method arguments is to be favored, in order to find relevant unifications. In practice, we implemented it by enumerating possible sets U_i, which we then try and combine together.

Evaluation of Operational Models

Now that we are able to generate a set of full operational models from an initial structure candidate, we ought to compare these in order to find the best one among all candidates.

The metric used to evaluate the operational model should compromise between models that generalize too much and models that overfit. Indeed, as described earlier, we wish to allow our agent to generate solutions to new acting problems, but we also want these solutions to be sound, i.e., with limited failures during execution.

To devise such a metric, we turn to the Minimum Description Length (MDL) principle (Grünwald 1996). This principle, coming from information theory, states that learning can be viewed as a form of data compression, as both intend to find some regularity in some source material. Therefore, in this framework, the best model is the one that can compress the data the most. Furthermore, as the total size of the compressed data includes the size of the model itself, used to reconstruct the source, this principle naturally guards against overly specific models.

Example uses of this technique range from learning Context Free Grammars (CFGs) of which HTNs, and thus our operational models, are close (Sapkota, Bryant, and Sprague 2012), to finding common graph patterns (Bariatti, Cellier, and Ferré 2020).

Framing our problem in the context of this MDL principle, we can say that a desirable model should be able to "compress" the demonstrations, while not being so specific that nothing else can be generated, thus limiting the complexity and therefore the size of this model.

Assuming we have generated an operational model candidate O_c as part of our search process, we define the description length of the operational model $L_{\text{opmod}}(O_c)$ and the description length of the demonstrations in $\mathcal{D}$ knowing O_c, written as $L_{\text{dem}}(\mathcal{D}|O_c)$. In order to compute the global length of the model and the demonstration set, we combine these two metrics, using a factor α to balance their relative importance, as defined in equation 2.

$$L(O_c, \mathcal{D}) = \alpha L_{\text{opmod}}(O_c) + L_{\text{dem}}(\mathcal{D}|O_c) \tag{2}$$

To compute $L_{\text{opmod}}(O_c)$, we consider a simple (arbitrary) alphabet to describe our candidate model O_c and compute the length of an optimal encoding of this description of our model. Considering a random variable X_{O_c} that takes as possible values the symbols in our alphabet, information theory tells us that the entropy $H(X_{O_c})$ of this variable bounds the expected codeword length of our optimal code

as $H(X_{O_c}) \leq \mathbb{E}(L) \leq H(X_{O_c}) + 1$. Noting $x_1, \ldots, x_n$ the symbols of our alphabet, and their occurrence probability $P(x_1), \ldots, P(x_n)$, the entropy formula is: $H(X_{O_c}) = -\sum_{i=1}^{n} P(x_i) \log(P(x_i))$. As we know the frequency of each symbol in our model as well as their total number, we can therefore compute a bound on the optimal model encoding length as in the example below.

As an example, consider the structure of the simple model learned for performing a task t shown in figure 2c. Viewing our model as a grammar-like structure, we can describe it with the following rule: $do : a \mid b \mid c \mid d$;. The frequency table of each symbol is given in the table in figure 7a, giving us the entropy $H(X_{O_c})$. Given that there are 9 symbols occurrence in the rule, this bounds the optimal value of $L_{\text{opmod}}(O_c)$ as shown in the equation in figure 7b. As this value will only be used for comparison, we arbitrarily choose the lower bound as L_{opmod}.

Symbol	do	a	b	c	d	$\mid$	;	Total
Frequency	1	1	1	1	1	3	1	9

(a) Frequency table

$$H(X_{O_c}) = -\left(6\left(\frac{1}{9}\log\left(\frac{1}{9}\right)\right) + \frac{3}{9}\log\left(\frac{3}{9}\right)\right)$$
$$= 2.64 \text{ bits}$$
$$L_{\text{opmod}}(O_c) \in [9 \times 2.64, 9 \times 3.64] = [23.76, 32.76] \text{ bits}$$

(b) Entropy and model length bounds.

Figure 7: Model length calculation for the one presented in figure 2c.

To compute $L_{\text{dem}}(\mathcal{D} \mid O_c)$, we need to evaluate the cost of encoding a trace $d \in \mathcal{D}$ based on our operational model. The trace can be mapped to a sequence of refinements from the original task to the primitive sequence. When planning or acting, each refinement of a task constitutes a choice point at which the engine must select one method among the applicable ones in the current state. Recalling all choices, we can reconstruct the refinement sequence and the corresponding trace. We define $L_{\text{dem}}(\mathcal{D} \mid O_c)$ as the encoding size of all choice points, averaged over all examples. This leads us to equation 3, where $\mathcal{C}$ is the set of choices an optimal planner has to make to reconstruct d from O_c, and $M_{\text{app},cp}$ is the set of applicable methods at a choice point cp.

$$L_{\text{dem}}(\mathcal{D} \mid O_c) = \frac{1}{|\mathcal{D}|} \sum_{d \in \mathcal{D}} \sum_{cp \in \mathcal{C}} \log\left(|M_{\text{app},cp}|\right) \qquad (3)$$

As an example, we will study the set of demonstration presented in figure 2a using again the model of figure 2c, assuming the methods have no preconditions. For each of the sequences in the demonstration set, we know that we have to choose three times among four methods, therefore $L_{\text{dem}} = \frac{1}{2}\left(3\log(4) + 3\log(4)\right) = 6$ bits

To show how this metric can be used, table 1 presents the different values computed for the examples presented in figure 2, using $\alpha = 0.505$. In order to normalize the tree length and the sequence, based on the value obtained from the most basic model, i.e. the one presented in figure 2c. Using $\frac{1}{2}\alpha$ instead of α reduces the relative importance accorded to the model size compared to its efficiency at compressing the demonstrations.

O_c	$L_{\text{opmod}}(O_c)$	$L_{\text{dem}}(\mathcal{D}\|O_c)$	$L(O_c, \mathcal{D})$	
			α	$\frac{1}{2}\alpha$
Fig. 2c	23.76	12	24	18
Fig. 2d	24.57	2	14.4	8.2
Fig. 2e	22	6.34	13.1	11.89
Fig. 2f	29.21	2	16.75	9.37

Table 1: Description length in bits for the examples presented in figure 2.

As the previous paragraph should have made clear, we need to know the choices that are required while acting with our candidate model O_c, in order to generate a given demonstration trace d, for which we use again the technique presented by Höller et al. (2021). We can then recover the choices made and their associated state by analyzing the resulting trace. Using an optimal planner, the computed demonstration length will be the true value of $L_{\text{dem}}(\mathcal{D} \mid O_c)$, while a non-optimal one may lead to an overestimation.

Preliminary Evaluation

While we are still implementing the ideas presented in this paper, we have been able to obtain some preliminary results on the metric to evaluate a candidate model for which preconditions and parameters have been identified as previously discussed.

A preliminary evaluation using these models on a given set of demonstration traces show that our metric appears indeed to favor structures that allow to efficiently generate the demonstrations, while still being of limited complexity, i.e. not favoring the deepest hierarchies, nor ones that are completely flat.

While we have shown some results earlier in table 1, the examples were too simple to show the interest of the metric, favoring the "lookup" model. However, working with more realistic examples, even simple ones, show the interest of our metric. Table 2 shows the results from an example dataset from a nondeterministic logistics domain with six primitive actions and three demonstration sequences long of respectively 11, 18 and 23 tasks. In the same way as before, we compute $\alpha = 1.35$ to normalize the model length based on the flat model. As we can see, reducing the model size relative importance α allows our metric indeed to favor a balanced tree. Evaluation on datasets with smaller traces for which it is possible to analyze all the generated tree structures by hand shows that the best scoring tree is indeed the one most efficient to regenerate the demonstrations.

Conclusion and Future Work

Our current work is focused on implementing a basic reactive acting engine to evaluate the currently generated models

O_c	$L_{\text{opmod}}(O_c)$	$L_{\text{dem}}(\mathcal{D}\|O_c)$	$L(O_c, \mathcal{D})$	
			α	$\frac{1}{2}\alpha$
Flat	**24.57**	33.15	**66.3**	49.7
Lookup	141.2	**1.58**	192.2	96.9
Balanced	37.71	20.98	71.9	**46.4**

Table 2: Description length in bits for some real examples.

on a test set of acting problems. This will allow us to conduct a first partial evaluation of the performance of the higher scoring models on some unseen acting problems. With this done, we will implement a local search strategy in lieu of the present crude generation method, to generate more complex models by removing the current structural restrictions. The next step will then be to integrate these operational models in a deliberative acting engine in order to leverage its lookahead capabilities, to make a better evaluation of the real world applicability of our models.

While we are still working on the implementation of the algorithm presented in the previous parts, there are some limitations that we are aware of and intend on addressing when the main algorithm will have been implemented. First, the current approach for extracting method preconditions is limited in which part of the preceding states is considered, which could be improved by integrating deictic references (Pasula, Zettlemoyer, and Kaelbling 2007), and superfluous predicates may be removed with an adaptation of the approach of Martínez (2017). Second, we wish to extract the postconditions of synthetic tasks with possibly disjunct intended effects, which may be done similarly to the extraction of the set of changes of an action of Pasula, Zettlemoyer, and Kaelbling (2007). Finally, it would be interesting to try and relax the assumptions made on the demonstration traces format.

References

Bariatti, F.; Cellier, P.; and Ferré, S. 2020. GraphMDL: Graph Pattern Selection Based on Minimum Description Length. In *IDA 2020 - Symposium on Intelligent Data Analysis*. Konstanz, Germany.

Colledanchise, M.; Parasuraman, R.; and Ögren, P. 2018. Learning of Behavior Trees for Autonomous Agents. *IEEE Transactions on Games*, 11(2): 183–189.

Fine-Morris, M.; and Muñoz-Avila, H. 2019. Learning Domain Structure in HGNs for Nondeterministic Planning. In *Proceedings of the 2nd ICAPS Workshop on Hierarchical Planning (HPlan 2019)*, 22–30.

Geib, C. W.; and Goldman, R. P. 2011. Recognizing Plans with Loops Represented in a Lexicalized Grammar. In Burgard, W.; and Roth, D., eds., *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press.

Ghallab, M.; Nau, D.; and Traverso, P. 2014. *Automated Planning and Acting*. Cambridge: Cambridge University Press. ISBN 978-1-139-58392-3.

Grünwald, P. 1996. A Minimum Description Length Approach to Grammar Inference. In Carbonell, J. G.; Siekmann, J.; Goos, G.; Hartmanis, J.; Leeuwen, J.; Wermter, S.; Riloff, E.; and Scheler, G., eds., *Connectionist, Statistical and Symbolic Approaches to Learning for Natural Language Processing*, volume 1040, 203–216. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-60925-4 978-3-540-49738-7.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2009. Learning Hierarchical Task Networks for Nondeterministic Planning Domains. In Boutilier, C., ed., *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 1708–1714.

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.

Kantharaju, P.; Ontañón, S.; and Geib, C. W. 2019. Extracting CCGs for Plan Recognition in RTS Games. In Guzdial, M.; Osborn, J. C.; and Snodgrass, S., eds., *Proceedings of the 2nd Workshop on Knowledge Extraction from Games Co-Located with 33rd AAAI Conference on Artificial Intelligence, KEG@AAAI 2019, Honolulu, Hawaii, January 27th, 2019*, volume 2313 of *CEUR Workshop Proceedings*, 9–16. CEUR-WS.org.

Martínez, D. 2017. *Learning Relational Models with Human Interaction for Planning in Robotics*. Ph.D. thesis, Universitat Politècnica de Catalunya.

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.

Patra, S.; Mason, J.; Ghallab, M.; Nau, D.; and Traverso, P. 2021. Deliberative Acting, Planning and Learning with Hierarchical Operational Models. *Artificial Intelligence*, 299: 103523.

Sapkota, U.; Bryant, B. R.; and Sprague, A. 2012. Unsupervised Grammar Inference Using the Minimum Description Length Principle. In Perner, P., ed., *Machine Learning and Data Mining in Pattern Recognition*, Lecture Notes in Computer Science, 141–153. Berlin, Heidelberg: Springer. ISBN 978-3-642-31537-4.

Zhang, Q.; Yao, J.; Yin, Q.; and Zha, Y. 2018. Learning Behavior Trees for Autonomous Agents with Hybrid Constraints Evolution. *Applied Sciences*, 8(7): 1077.

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.

Zhuo, H. H.; Peng, J.; and Kambhampati, S. 2019. Learning Action Models from Disordered and Noisy Plan Traces. *arXiv:1908.09800 [cs]*.