Learning HTN Domains using Process Mining and Data Mining techniques

José Á. Segura-Muros and **Raúl Pérez** and **Juan Fernández-Olivares**
josesegmur@decsai.ugr.es and fgr@decsai.urg.es and faro@decsai.ugr.es
Department of Computer Science and Artificial Intelligence
University of Granada.

Abstract

HTN Planning is a powerful technique for problem-solving. Modelling an HTN domain is a cumbersome task that requires time and skill. Planning domain learning is a successful technique that allows reducing the bottleneck of writing HTN domains. This paper describes the first stage of an algorithm for obtaining HTN domains from a set of plan traces. These domains are learned using Process Mining and Data Mining techniques. Process Mining is used to learn the domain's hierarchical structure and Data mining is used to learn to action model of the domain's primitive tasks and decomposition methods.

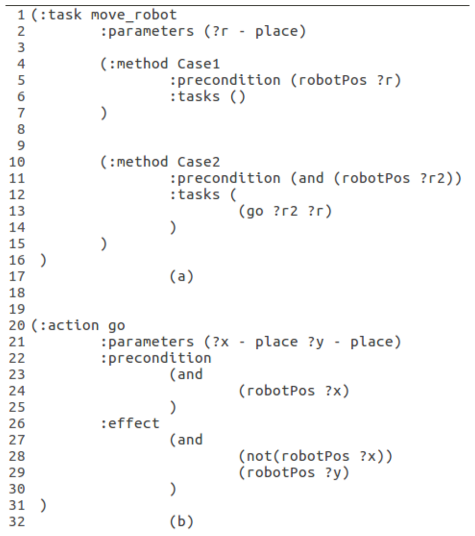
1 Introduction

Hierarchical Task Network (HTN) planning is an effective method for problem-solving (Ghallab, Nau, and Traverso 2004) that has been successfully used in a great number of real-world applications (Georgievski and Aiello 2015). HTN planning was developed with the objective of expressing planning problems in a structured way (Erol, Hendler, and Nau 1994). The bases of HTN planning are: actions, or primitive operators, that are non-decomposable activities which encode changes in the world, and high-level tasks, or compound tasks, that may be decomposed into simpler tasks (primitives or other compound tasks). A compound task can have different reduction schemes, or methods, that are selected if some preconditions are true. Figure 1 shows the codification of a primitive task and a compound task in HPDL(Castillo et al. 2006), a PDDL extension for HTN.

Other HTN features are:

- The initial state is a set of literals that describe the world at the beginning of the problem.
- The goal is a partially ordered set of tasks that need to be carried out.
- The HTN planning algorithm takes the goal, explores the space of possible decompositions replacing it by its component activities until there are only primitive actions.

HTN domains describe how primitive tasks affect the world and how they relate to other tasks, following a hierarchical relation. Defining an HTN domain is a cumbersome task traditionally done by knowledge engineers that requires time and skill. Additionally, this task is very difficult even for domain designers because it requires a lot of domain knowledge-engineering effort. Alleviating this issue can help to extend the use of HTN planning in several fields.



```
(:task move_robot
        :parameters (?r - place)

        (:method Case1
                :precondition (robotPos ?r)
                :tasks ()
        )


        (:method Case2
                :precondition (and (robotPos ?r2))
                :tasks (
                        (go ?r2 ?r)
                )
        )
 )
                (a)


(:action go
        :parameters (?x - place ?y - place)
        :precondition
                (and
                        (robotPos ?x)
                )
        :effect
                (and
                        (not(robotPos ?x))
                        (robotPos ?y)
                )
 )
                (b)
```

Figure 1: The basics of HTN planning domains in the HPDL domain language: (a) A compound task with two different methods of decomposition. (b) A primitive action.

Planning domain learning has become a successful technique to reduce the bottle-neck in the process of writing HTN planning domains. Planning domain learning algorithms take as input the plan execution traces of previously solved problems and discover the planning domain that generates such traces. Planning domain learning, and particularly HTN domain learning, has been already addressed by different approaches with relative success. A common limitation of current HTN domains learners is that they lack enough expressiveness to deal with real-world problems (like numerical predicates or time-resources management) and that they require some extra structural information of the domain to work.

In order to avoid the need of extra information and provide a higher degree of expressiveness HTN domain learner we present this paper: a first step to develop an HTN domain learning process that overcomes the limitations of current approaches, integrating Process Mining and inductive learning techniques.

On the one hand, our approach uses Process Mining (PM) procedures as a supporting technique to learn the structure of the HTN domain. Process Mining (Weijters and van der Aalst 2001) is a set of techniques that allow the discovery of process models from event traces, to extract information from those models and to check the conformance of the models with real-world processes. The problem of learning planning domains' structures have been dealt previously using other techniques (Gopalakrishnan, Muñoz-Avila, and Kuter 2016), but Process Mining techniques allow us to get closer to real-world problems.

On the other hand, with the structural information and the plan traces, we can learn the tasks definition using inductive learning data mining techniques. Specifically, from inductive learning, we are interested in rule learning algorithms. Because as opposed to other approaches (Mourao et al. 2012), rule learning classification algorithms allows us to learn not only logic predicates but also numerical functions and time and resources constraints.

The main difference between PM and traditional Data Mining (DM) is that DM focus in learning of relations between data attributes and PM tries to detect patterns of data that explain a certain behaviour. We use this difference to learning the needed information in each step of the algorithm. The major challenge of the approach presented in this paper is to deal with this difference too. Planning, PM and DT requires the proper type of data to carry out its goals and we have to adequate the data in each step to guide them and solve the problem. Although, the advantage of using this array of techniques is that we can learn some information instead of providing it, minimising the input of the algorithm.

The rest of the paper is organized as follows. The next section discusses related work in more detail. Section 3 defines the problem of domain learning from plan traces and presents our solution. Section 4 explains our domain learner algorithm. Section 5 shows the experimental results. And finally, Section 6 discusses the conclusions obtained during this research and presents some future work.

2 Related Work

There are several proposed solutions to learn planning domains (Jiménez et al. 2012). These solutions can be classified into two types: action model learners, solutions that learn how the domain's tasks relate with the world. And search control learners, algorithms that learn how the tasks relate between themselves.

On the one hand, the action model learners workspace is the preconditions and effects learning problem. This is a problem that can be categorised by the observability (full or partial) of the world states and the type of the tasks' effects (deterministic or stochastic).

ARMS, one of the most successful solutions, uses a MAX-SAT solver to learn STRIPS-like action models from incomplete plan traces (Yang, Wu, and Jiang 2007). This approach codifies logic predicates, restrictions between them, and domain information in a single logic formula, and then, using a weighted MAX-SAT solver extracts information of preconditions and effects of the domain's tasks. This approach works even if some information about intermediate states is missing.

The problem with ARMS is that although it can handle missing state information, it can not work with noisy plan traces. AMAN (Zhuo and Kambhampati 2013) considers all plan traces as noisy and creates a set of different action models from them. Finally, AMAN selects the action model that better fits the plan traces.

Also, there are new approaches that use classic machine learning algorithms to learn tasks definitions. For example, (Mourao et al. 2012) uses a support vector machine (SVM) to learn rules that model the preconditions and effects of tasks. This SVM approach trains a set of classifiers for each task and then creates a rule from each set. These rules correspond to the preconditions and effects of the domain's tasks.

On the other hand, the search control problem ranges from the learning of macro-actions to the learning of heuristics for a planner or the learning of decomposition methods for high-level tasks into simpler ones. As we only care about the problem of learning decomposition methods, specifically the decomposition in a hierarchical way, we are going to discuss only the control search learners that concern about this problem.

HTNMaker (Hogg, Muñoz-Avila, and Kuter 2008) learns the methods structure incrementally using a bottom-up strategy using a set of plan traces and a collection of annotated tasks. An annotated task is a tuple $< Task, Pre, Eff >$ where $Task$ is a task or HTN method, Pre are its preconditions and $Post$ are its effects. HTNMaker traverses the plan traces and if there is an annotated task whose effects match the state S_{i+n} and whose preconditions match the state S_i, it returns the set of tasks $Task_i, ..., Task_{i+n}$ as a reduction scheme of the annotated task matched. The main problem of HTNMaker is that it can't handle incomplete or noisy plan traces.

HTNLearn (Zhuo, Muñoz-Avila, and Yang 2014) takes the idea of ARMS and improves it with structural information. This is done by coding the annotated tasks of HTNMaker as a logic constraint to be solved by a MAX-SAT algorithm. The benefits of this approach are that not only can work with missing intermediate states, but it can learn with incomplete annotated tasks.

WORD2HTN (Gopalakrishnan, Muñoz-Avila, and Kuter 2016) uses text semantics analysis to learn the domain's structure. It identifies sub-goals in the plan traces and clusters tasks with the same purpose. The main problem of this solution is that is very affected by noise.

Finally, we can find pHTN (Li et al. 2010). This algorithm defines each compound task's reduction scheme as a tuple $< p, t >$ where p is the probability of the reduction scheme to be executed and t the correspondent list of tasks.This approach's weak point is that it rely only on probability to control the flow of the planner rather than a deterministic logic preconditions.

3 Developing an HTN domain learning process using Process Mining and Data Mining techniques

The main problem with the solutions proposed in the last section is that very few of them addresses the whole structured planning domain learning problem. And even HTNLearn, the only solution that solves both problems together, can't handle noise or other information than logic predicates. Another problem that can impede the extension of HTN planning as a solution to solve real-world problems is that some of these algorithms need extra information rather than the plan traces.

This paper presents the firsts steps of a solution to these problems by providing an algorithm that can learn the actions models of a planning domain and the relation between the domain's tasks in a hierarchical way. We aim to solve the problem without the need of extra structural information and using incomplete and noisy plan traces. Another goal of this solution is to be able to handle numerical information and manage time or resources consumption. To achieve this, we present an approach to learn HTN planning domains using process mining techniques and inductive learning.

On the one hand, process mining allows us to learn the structure of the domain. This is achieved by treating plan traces as event logs and using a process discoverer with them. Process discovery is the branch of process mining that takes care of learning models that fit a given event log. Process mining techniques are widely used with real-world problems, and they are very robust against noise and incompleteness. The ability to discover the structural information eliminates the need of providing our approach with extra structural knowledge and enables our HTN domain learner to manage real-world problems.

On the other hand, when the domain's structure has been learned we use inductive learning to learn the preconditions and effects of tasks and methods. The inductive learning techniques try to find a hypothesis that explains a set of given observed instances. We consider the problem of learning the task's definition as a rule-based classification problem where the observed instances are the states of the plan traces. The rule-based classification algorithm is used to find the rules that model the pre-states and post-states for each primitive task and method of the domain. We are interested on using rule-based learning algorithms because rules are easy to interpret from a human point of view and they explicitly show the relationships among the variables involved in the learning problem. Another benefit of using inductive learning techniques is that these algorithms can not only deal with logic predicates but to learn numerical values or time/resources consumption too. This property allows us to propose an approach that can produce more expressive models that can be applied to more complex problems.

Next section will explain in detail how our algorithm works, what kind of techniques uses and how it combine them to learn an HTN domain.

4 Algorithm overview

Given a set of plan traces PT, we want to learn an HTN planning domain capable of solving the problems from which the plan traces were generated. To achieve this we use process mining and data mining techniques. We divide the problem of learning HTN domains in two subproblems:

1. Learn the structure of the HTN domain.
2. Learn the action model of the tasks and methods of the domain.

First, define a plan trace as an ordered set of primitive tasks and states such as:

$$PlanTrace = \{S_0, T_0, S_1, T_1, S_2, ..., S_n, T_n, S_{n+1}\}$$

Where S_0 is the initial state of the problem, S_{n+1} a the goal state, $S_i|(0 < i < n+1)$ are intermediate states and T_i are primitive tasks. For every task T_i, S_i is its pre-state and S_{i+1} is its post-state.

Both the problem of learning HTN structures as the problem of learning require different techniques and our algorithm tries to join them. This implies that in every step our algorithm must adapt the data so they can be used by every technique, and at the end, it must fuse the output of both kind of techniques in a single valid output. An overview of our algorithm is as follows:

HTN Domain Learner.

Input: A set of plan traces PT.
Output: An HTN planning domain D.

1. Create an event log EL from PT. Each plan of PT is considered an event trace. Those event traces groups a collection of events. These events correspond to each plan's tasks. Each event is included in the event trace following the order of appearance in the plan. Finally, every event trace is grouped in an event log EL
2. Create a valid process model from EL. In this step, we use Inductive Miner (IM) (Leemans, Fahland, and van der Aalst 2013), a process discovery algorithm, to find a process tree T that fits EL. IM takes EL and extracts the events recursively splitting the event log, trying to find the relation between them. IM follows a divide and conquer strategy to do so and returns a process tree capable of reproducing EL. T can be seen as a representation of the hierarchical structure of the domain.
3. Create sets of pre-states and post-states using PT and the information provided by T. Starting from PT, our algorithm takes the pre-state and post-state of each task of PT. Then, using the domain's structural information contained in T the algorithm arranges the tasks of PT in hierarchical methods using a bottom-up strategy. Finally, the algorithm proceeds to calculate the corresponding pre-states and post-states of each of them. As a result, this process creates a set of pre-states and post-states for every task and method in the domain.
4. Calculation of the schema form of the pre-states and post-states and building of datasets. For each state in the sets

of pre-states and post-states, the algorithm calculates its schema form. This is done by substituting the constants in the state's predicates by a generalised form of the parameters of his associated task. Finally, the algorithm builds a dataset for every task and method of the domain, using the schematized predicates as attributes.

5. Learning of tasks and methods preconditions and effects from the datasets. By considering the preconditions and effects learning problem has a classification problem where the classes are ”pre-state” and ”post-state” our algorithm learns the action model of every task and method of the HTN domain using a rule learner classifier. The classifier tries to find the combination of attributes (state's predicates) models the pre-states and post-states of each task and method. The algorithm used to do so is NSLV. This algorithm uses fuzzy rules (Zadeh 1965) to represent classification hypothesis and a genetic algorithm to learn them.
6. HTN domain generation. Finally, our algorithm creates an HTN planning domain written in HPDL from T and the rules of the previous step. T allows generating the structure of the domain while the rules allow generating the preconditions and effects of each task and method of the domain.

In the next lines, we explain in further detail the algorithm step by step. During these explanations, we provide the background concepts needed to understand how the algorithm works and a motivation example to help follow the whole process.

Step 1

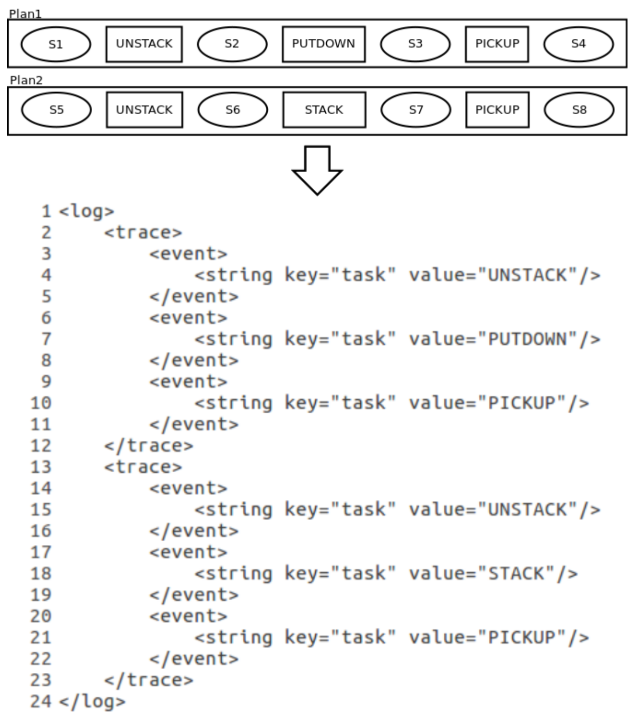
In process mining, an event trace is a record of a sequence of steps. These steps, called events, indicate a non-decomposable activity performed at a particular time point. An event trace is the definition of the steps necessary to carry out a single high-level process. An event log consists of multiple event traces that carry out their tasks from start to finish.

On the other hand, a plan is a set of tasks partially ordered by a temporal relation. Given this similarity, our algorithm considers each task as an event and creates an event trace for every plan of PT. These event traces are grouped in an event log EL that will be used in the next steps of the algorithm. Figure 2 shows exemplifies this process.

Step 2

In this step, our algorithm tries to learn the structure of the domain contained in EL. This is achieved by finding a model that fits the event log produced in step 1. To discover this model we use Inductive Miner (IM) (Leemans, Fahland, and van der Aalst 2013) a process discovery algorithm based on Process Trees. A process tree is a representation of a workflow net: a rooted tree in which leaves are events or activities and all other nodes are operators. A process tree describes a process model, and his operators describe how the subprocesses of their subtrees are executed. Figure 3 shows an example of a process tree.

The standard operators defined for process trees are the following: XOR (X) represents exclusive selection of one



```
<log>
    <trace>
        <event>
            <string key="task" value="UNSTACK"/>
        </event>
        <event>
            <string key="task" value="PUTDOWN"/>
        </event>
        <event>
            <string key="task" value="PICKUP"/>
        </event>
    </trace>
    <trace>
        <event>
            <string key="task" value="UNSTACK"/>
        </event>
        <event>
            <string key="task" value="STACK"/>
        </event>
        <event>
            <string key="task" value="PICKUP"/>
        </event>
    </trace>
</log>
```

Figure 2: Plan traces to event log example. Each plan trace consists of a set of tasks and logic states. The result is a log composed by event traces. Each task in the plan traces not only contains the task's name but its parameters instantiated.

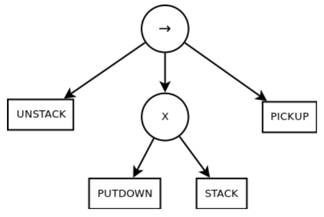


Figure 3: Process tree representing the event log showed in figure 2.

subtree, SEQUENCE $(\rightarrow)$ represents the sequential execution of all subtrees, XORLOOP $(\bigotimes)$ represents the loop of any exclusively selected subtree, AND $(\wedge)$ represents the parallel execution of all subtrees, and OR $(+)$ represents the execution of any combination of subtrees.

IM algorithm takes an event log, and using a divide and conquer strategy, splits the event log extracting recursively the events and their relation to the rest of events of the event traces. IM selects the process tree operator that best fits EL, dividing the activities in log EL into disjoint sets. EL is split into sublogs using these sets. Then, the sublogs are mined recursively, until they contain a single event.

A process tree not only contains structural information about the mined log but contains control information. If we consider the events of the event log as primitive tasks again this control information explains how the tasks relate between them and simplifies the problem of HTN domain

Control Node	HTN Structure
SEQUENCE	$Method_1(Task_1, Task_2...Task_N)$
AND	$Parallel[Task_1, Task_2...Task_N]$
XOR	$(Method_1, Method_2...Method_N)$
XORLOOP	$(Method_1, Method_2...Method_N)$ (a)
OR	$(Method_1, Method_2...Method_N)$ (b)

Table 1: Process Tree to HTN structure conversion table.
(a) Every method has a call to the parent task.
(b) The algorithm creates a method for each combination of child nodes.

structure learning. By considering the process' tree sub-trees has sub-tasks of an HTN compound task we can generate easily HTN compound tasks using the conversions of table 1 to the process' tree internal nodes. Each operator is considered a compound task of the HTN domain and depending of the operator its child subtrees are considered as methods of its parent node or tasks of a single method compound task.

Step 3

Given a set of plan traces PT our algorithm proceeds to separate each state of PT in sets of pre-states and post-states. A pre-state S_i is the state where primitive task T_i is applied, and a post-state S_{i+1} is the state produced by T_i. I.E. given the first plan trace showed in figure 2 we can see that S_1 is the initial state of the problem solved by the plan, S_4 is its goal state, and UNSTACK, PUTDOWN and PICKUP are the primitive tasks of the planning domain that solved the problem. Every primitive task's parameter P_i is instantiated with proper the constant.. We can separate the plan trace in a set of tasks with associated states:

$$Task:\ UNSTACK(b_{003},\ b_{000})\ Pre: S_1\ Post: S_2$$
$$Task:\ PUTDOWN(b_{003})\ Pre: S_2\ Post: S_3$$
$$Task:\ PICKUP(b_{001})\ Pre: S_3\ Post: S_4$$

Finally, using the structure information learned in step 2 our algorithm groups each primitive task in PT in high-level compound tasks. Our algorithm first sorts the methods learned by their dependencies. Then, it selects the first one and tries to find a subset of tasks in a plan trace that matches the reduction scheme of the method. If the algorithm finds a match if substitutes the subset in the plan trace by its high-level method. The algorithm follows a bottom-up strategy selecting iteratively methods substituting tasks subsets with them in PT. For every new high-level task in PT, the algorithm calculates its pre-state and post-state. To illustrate the whole process we show the next example:

Given an HTN structure previously learned (figure 4) and the plan trace presented earlier, our algorithm sorts the methods as CT_1M_0, CT_2M_0 and CT_2M_1, and then, proceeds to substitute them in the plan trace. First, selects CT_2M_0 and CT_2M_1 because every task they depend on already is in the plan trace. Each occurrence of these tasks is substituted by the proper CT_2M_x method:

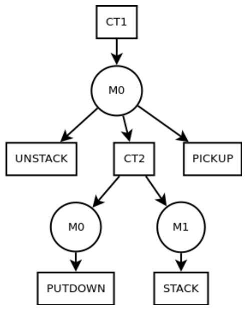
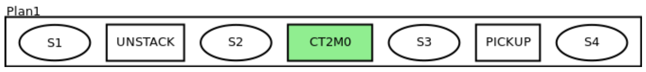


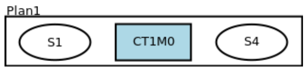
Figure 4: HTN structure extracted from the process tree presented in figure 3.



The algorithm then generates the pre-states and post-states of the new method in the plan trace.

$$Task:\ UNSTACK(b_{003},\ b_{000})\ Pre: S_1\ Post: S_2$$
$$Task:\ PUTDOWN(b_{003})\ Pre: S_2\ Post: S_3$$
$$Task:\ PICKUP(b_{001})\ Pre: S_3\ Post: S_4$$
$$Task:\ CT_2M_0(b_{000})\ Pre: S_2\ Post: S_3$$

The algorithm selects CT_1M_0, the last method in the list, and proceeds the substitutions. The algorithm takes into account empty reduction schemes when matching subsets of tasks of the plan trace.



And finally, the algorithm calculates the pre-states and post-states of CT_1M_0

$$Task:\ UNSTACK(b_{003},\ b_{000})\ Pre: S_1\ Post: S_2$$
$$Task:\ PUTDOWN(b_{003})\ Pre: S_2\ Post: S_3$$
$$Task:\ PICKUP(b_{001})\ Pre: S_3\ Post: S_4$$
$$Task:\ CT_2M_0(b_{000})\ Pre: S_2\ Post: S_3$$
$$Task:\ CT_1M_0(b_{003},\ b_{000},\ b_{001})\ Pre: S_1\ Post: S_4$$

This process is repeated for each plan trace. If there's already a task in the list, the algorithm includes the respective states in it pre-states and pos-states lists.

Step 4

To solve the preconditions and effects learning problem, first, our algorithm creates a collection of datasets. These datasets follows the next structure:

$Atrib_1$	$Atrib_2$	...	$Atrib_n$	Class
$Value_{11}$	$Value_{12}$	...	$Value_{1n}$	$Label_1$
$Value_{21}$	$Value_{22}$	...	$Value_{2n}$	$Label_2$
...	...	...	...	...
$Value_{m1}$	$Value_{m2}$	...	$Value_{mn}$	$Label_m$

where each row is a logic state defined by the predicates $Atrib_j$ with values $Value_{ij}$. $Value_{ij}$ is 0 if the atom is false in the state. $Value_{ij}$ 1 if the atom is true. $Label_i$ value can be *pre-state* of *post-state* depending the relation of the state with a given task.

To do so, our algorithm first calculates the schema form of every task and state. First, the algorithm assigns a name to each parameter of the task and takes the value of that parameter. This name is generated using the parameter's order in the task header and is the same for every task of the same type. Then, the algorithm looks for the parameters values in the logic predicates of the associated states, substituting these values with the corresponding name. For example, given a task $UNSTACK(b_{003},\ b_{000})$ and a state $HOLDING(b_{003}) \wedge ONTABLE(b_{001}) \wedge ON(b_{000},\ b_{001}) \wedge CLEAR(b_{000})$ their schema form is: $UNSTACK(P_1,\ P_2)$ with a state $HOLDING(P_1) \wedge ONTABLE(b_{001}) \wedge ON(P_2,\ b_{001}) \wedge CLEAR(P_2)$. Finally our algorithm deletes every predicate of the state that have not undergone at least one substitution.

Now the algorithm creates a dataset for every task and method of the domain using propositionalization techniques (Lachiche 2011). First, the algorithm selects the states whose task type and parameters are the same. The attributes of these datasets are the union of the different predicates of the selected states, and the label of these states depend on their relation to the given task. If a predicate doesn't appear in a state the value of the correspondent cell of the dataset is set as *Missing Value*. Classification algorithm treats missing values following the Open-World Assumption.

Step 5

For each dataset, our algorithm tries to find a hypothesis that explains which attributes models the pre-states and post-states of the dataset's tasks or method. The algorithm used to learn these hypothesis is NSLV (García, González, and Pérez 2014). NSLV is a fuzzy rule-based classification algorithm that learns using a genetic algorithm. NSLV is part of the SLAVE(Gonzblez and Perez 1999) algorithms family and that are based on the sequential covering strategy. A description of this strategy can be seen in listing 1. We select fuzzy models to learn the rules that define the states because in real-world problems things aren't entirely true or false. Real-World noise and incompleteness affect the learning process and fuzzy models' vagueness allow us to deal with them more easily(Zadeh 1965).

```
function SEQUENTIAL–COVERING (X,Y,E)
    Learned−rules ← {}
    rule ← LEARN–ONE–RULE(Y,X,E)
    while PERFORMANCE(rule ,E) > 0 do
        Learned−rules ← Learned−rules +
            rule
        E ← E − examples correctly
            classified by rule
        rule ← LEARN–ONE–RULE(Y,X,E)
    end while
    return Learned−rules
end function
```

Listing 1: The Sequential Covering strategy.

As said earlier, NSLV is a fuzzy rule-based algorithm. These rules follow the DNF rule model (Michalski 1983):

$$IF\ X_1\ is\ A_1 \wedge X_2\ is\ A_2 \wedge ... \wedge\ X_n\ is\ A_n$$

$$THEN\ Y\ is\ B\ with\ weight\ w$$

where $X_1, ..., X_n$ are the attributes, $A_1, ..., A_n$ are the values taken for each attribute, each A_i is a subset of D_i, the fuzzy domain of X_i, Y is the consequent variable and B is the value of the consequent variable. Finally, w is a measure of the weight associated to the rule. These kind of rules are very useful to the pre-states and post-states learning problem because they allow us to categorize continuous variables under fuzzy sets. This feature not only simplifies the numerical predicates learning problem but also provides a degree of vagueness very useful to deal with real-world problems.

SLAVE takes a target attribute Y, a set of learning attributes X and a set of learning examples. The output set $LearnedRules$ is where the learned rules will be keep, sorted by $PERFORMANCE$. While $PERFORMANCE$ is positive SLAVE learns rules that explain al least one element of E. $PERFORMANCE$ is a function that measures the contribution of a rule to the $Learned - rules$ set. Roughly, $PERFORMANCE$ measures the increase in the degree of completeness that causes the last learned rule over the set of examples E. SLAVE creates a new rule, and if it explains some examples, the algorithm deletes them from E and include the rule in $LearnedRules$.

The *LEARN-ONE-RULE* function learns the rules using a genetic algorithm. To represent of the rules in the genetic algorithm, SLAVE uses a mixed codification based on levels. First level uses binary code for representing the assignment attribute/value. Meanwhile in the second level, uses real code for representing the relevance of each attribute for a specific value of the consequent variable. NSLV is a modified version of SLAVE that can learn several rules for different classes at once, improving the overall performance of the *LEARN-ONE-RULE* function. NSLV allows us to obtain descriptive rules. The most common rule in classification algorithm is the predictive rule. In this kind of rule, in the antecedent of the rule only appear the discriminant attributes while in the descriptive rule appears all the relevant attributes for describe the class. We able to get the descriptive rules is important to the HTN domain learning problem since to learn the model of the pre-states and post-states are needed all of the predicates contained in these states.

At last, from each dataset, we extract 2 rules. These rules models what predicates forms the pre-states (Pre) and post-states $(Post)$ of every task and method of the domain. Now we only have to extract the preconditions and effect from them. Construct the preconditions is trivial since we have a model of the pre-state of the task. To learn the effects of the task or method we simply calculate $\Delta(Pre, Post)$. $\Delta(Pre, Post)$ is the difference between two states. It can be seen as a list of added predicates in $Post$ that were not in Pre and a list of deleted predicates in $Post$ that were in Pre.

Step 6

In this final step, our algorithm takes the HTN structural information previously learned and creates an HTN Domain encoded in HPDL. This incomplete HTN Domain lacks preconditions and effects in their primitive tasks and methods. To solve this problem our algorithm simply takes the preconditions and effects learned in the previous step and fills the gaps of the domain. If a method has no preconditions and effects calculated, and it has a empty reduction scheme, the algorithm sets its preconditions as the effects of the compound parent task.

Finally, the algorithm calculates the parameters of the domain's compound tasks following the next formula:

$$Params(CTask) = ParM_1 \bigcap ParM_2 \bigcap ... \bigcap ParM_n$$

where $CTask$ is a compound task of the HTN domain and $ParM_i$ is the union of the tasks' parameters of method M_i of $CTask$

5 Experimental Evaluation and Results

As we said above, this paper presents the first stage of our domain learning algorithm. The experiments presented in this section are aimed to prove the validity of our approach with the domain learning problem. To do so, the experiments were carried out using a set of planning problems over various synthetic planning domains.

We are going to measure the accuracy of our algorithm solving a set of test problems. To do so, first, we are going to define the accuracy as $accuracy = \frac{problemsSolved}{totalProblems}$ where $totalProblems$ is the total number of test problems and $problemsSolved$ the number of problems solved by the HTN domain generated by our algorithm. A problem is considered solved if a planner can generate a plan that accomplish the problem's goals. In this stage of the research, we don't care so much about the quality of the plans rather than the fact of obtaining one.

One of the domains used in our experiments is a simplified version of the Zeno-Travel domain presented from the 3rd International Planning Competition (IPC-3). In this planning domain, the problems involve using aeroplanes to move people between cities, and they require fuel to fly. The domain's original version uses numerical predicates to manage the aeroplanes' fuel, cargo and speed, but to simplify the problem we create a domain without them.

The other domain used is Blocks-World presented from the 2nd International Planning Competition (IPC-2). In this planning domain, the problems involve a robotic arm stacking blocks. This domain usually deals with interleaved goals, for example, by unstacking others blocks to accomplish a given task.

The experimental evaluation consists of 10 runs of experiments using a set of 40 problems separated in a set of training examples and a set of test examples for each domain. The problems generated for Zeno-Travel consist on carrying various passengers to a given destination. The problems generated for Block-Worlds consist of make towers using various blocks. We use SIADEX (de la Asunción et al. 2005), a temporal HTN planner, to solve these problems using a handmade HTN planning domain and obtain a set of plan traces. SIADEX can handle numerical predicates and temporal constraints. Then, with the training plan traces, we obtain a new HTN domain. Finally, we use SIADEX again with the new domain learned and the test problems. These new plan traces are compared with the test plan traces to measure the accuracy of the algorithm. In each run, the percentage of examples in the training and test sets varies ranging from the 5% of examples in the training set and the 95% in the test sets to the 50% of examples in each set. To increase the reliability of the experiments we applied cross-validation with 10 subsets in each run.

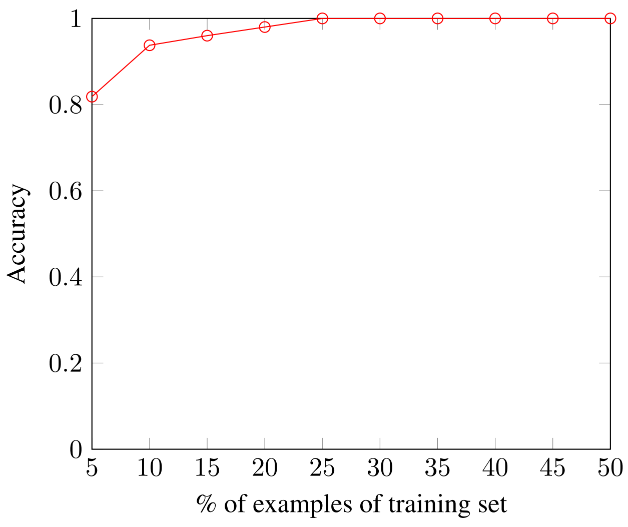


Figure 5: Zeno-Travel Domain Experimental results

The results presented in figure 5 corresponds with the experiments of Zeno-Travel domain. Overall it shows from a training set with the 25% of the examples our algorithm can learn an HTN domain that can solve every test problem. This implies that in fact, our algorithm can learn simple HTN planning domains. These problems have a very straight forward structure so with very few examples our algorithm can get a high accuracy. Our algorithm learns several complex tasks that simply call another compound tasks and can be reduced. Comparing original domain and the learned one, we can see that the primitive tasks are the same in both domains and that the learned model has 4 compound tasks in contraposition to the original that has 2 compound tasks. The quality of the plans is the same with both domains as they produce the same plans.

In the other hand, the experiments with the Blocks-World domain showed that our approach can't handle interleaved goals. The learned domain can solve simple Blocks-World's subproblems it can't detect the patterns necessaries to solve a full Blocks-World problem.

6 Conclusions and Future Work

HTN planning is an effective method for problem-solving, but design an HTN domain is a cumbersome task that requires a lot of effort from knowledge engineers. To alleviate this problem we have described the first stage of a new HTN domain learning algorithm. This algorithm uses process mining techniques to learn the domain's structure and machine learning techniques to learn the preconditions and effects of the domain's tasks and methods.

The algorithm presented in this paper takes a set of plan traces and creates an event log from them. From this event log, our algorithm extracts a process tree with the structure of the HTN domain using process discovery techniques. With this structure, the algorithm calculates the pre-states and post-states not only for the primitive tasks of the domain but for the methods of the domain's compound tasks too. Finally, the algorithm finds a rule for the preconditions and effects of each task and method of the domain. And then, joins them with the domain's structure to create a full HTN domain.

We have presented theoretical results showing that our algorithm can learn simple HTN domains. The experiments were carried out using a simplified version of the Zeno-Travel planning problem and our algorithm was able to learn how to solve them all.

Our next steps to expand this work will be in several directions. First, we are going to work with numerical predicates, trying to learn them and creating more complex HTN domains. Secondly, our work will be focused on the learning of these domains from noisy and incomplete plan traces. These two steps are very straight forward because the algorithm used to learn the preconditions and effects of the domain's tasks and methods is already designed to work with this kind of information. In the other hand, we detected problems learning the domain's structure of planning problems with interleaved subgoals. Solving these problems is a priority in the next steps of the algorithm. As well as the simplification of the domain's structures.

7 Acknowledgements

This research is being developed and partially funded by the Spanish MINECO R&D Project PLAN MINER TIN2005-71618-R

References

Castillo, L. A.; Fernández-Olivares, J.; García-Pérez, Ó.; and Palao, F. 2006. Efficiently handling temporal knowledge in an htn planner. In *ICAPS*.

de la Asunción, M.; Castillo, L.; Fdez-Olivares, J.; García-Pérez, O.; González, A.; and Palao, F. 2005. Siadex: An interactive knowledge-based planner for decision support in forest fire fighting. *AI Commun.* 18(4):257–268.

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN planning: Complexity and expressivity. *Proceedings of 20th AAAI Conference* 1123–1128.

García, D.; González, A.; and Pérez, R. 2014. Overview of the slave learning algorithm: A review of its evolution and prospects. *International Journal of Computational Intelligence Systems* 7(6):1194–1221.

Georgievski, I., and Aiello, M. 2015. Htn planning: Overview, comparison, and beyond. *Artificial Intelligence 222.* 124–156.

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

Gonzblez, A., and Perez, R. 1999. Slave: a genetic learning system based on an iterative approach. *IEEE Transactions on Fuzzy Systems* 7(2):176–191.

Gopalakrishnan, S.; Muñoz-Avila, H.; and Kuter, U. 2016. Word2HTN:learning task hierarchies using statistical semantics and goal reasoning. *The IJCAI-2016 Workshop on Goal Reasoning. AAAI Press.*

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: learning HTNs with minimal additional knowledge engineering required. *Proceedings of the 23rd national conference on Artificial intelligence - Volume 2* 950–956.

Jiménez, S.; Rosa, T. D. L.; Fernández, S.; Fernández, F.; and Borrajo, D. 2012. A review of machine learning for automated planning. *The Knowledge Engineering Review* 27(4):433–467.

Lachiche, N. 2011. Propositionalization. In *Encyclopedia of Machine Learning*. Springer. 812–817.

Leemans, S. J. J.; Fahland, D.; and van der Aalst, W. M. P. 2013. Discovering block-structured process models from event logs - a constructive approach. *34th International Conference, PETRI NETS* 311–329.

Li, N.; Cushing, W.; Kambhampati, S.; and Yoon, S. 2010. Learning probabilistic hierarchical task networks to capture user preferences. *arXiv preprint arXiv:1006.0274.*

Michalski, R. 1983. *A Theory and Methodology of Inductive Learning*. Symbolic Computation. Springer Berlin Heidelberg.

Mourao, K.; Zettlemoyer, L. S.; Petrick, R. P. A.; and Steedman, M. 2012. Learning STRIPS operators from noisy and incomplete observations. *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence.*

Weijters, A. J. M. M., and van der Aalst, W. 2001. Process mining: Discovering workflow models from event-based data. *Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001)* 283–290.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence Journal.* 107–143.

Zadeh, L. 1965. Fuzzy sets. *Information and Control* 8(3):338 – 353.

Zhuo, H. H., and Kambhampati, S. 2013. Action-model acquisition from noisy plan traces. *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence.* 2444–2450.

Zhuo, H. H.; Muñoz-Avila, H.; and Yang, Q. 2014. Learning hierarchical task network domains from partially observed plan traces. *Artificial Intelligence Vol. 212.* 134–157.