Method Composition through Operator Pattern Identification

Maurício Cecílio Magnaguagno, Felipe Meneguzzi
School of Informatics (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre - RS, Brazil
`mauricio.magnaguagno@acad.pucrs.br`
`felipe.meneguzzi@pucrs.br`

Abstract

Classical planning is a computationally expensive task, especially when tackling real world problems. To overcome such limitations, most realistic applications of planning rely on domain knowledge configured by a domain expert, such as the hierarchy of tasks and methods used by Hierarchical Task Network (HTN) planning. Thus, the efficiency of HTN approaches relies heavily on human-driven domain design. In this paper, we aim to address this limitation by developing an approach to generate useful methods based on classical domains. Our work does not require annotations in the classical planning operators or training examples, and instead, relies solely on operator descriptions to identify task patterns and the sub-problems related to each pattern. We propose the use of methods that solve common sub-problems to obtain HTN methods automatically.

1 Introduction

Finding a sequence of actions to reach a desired state may be considered a trivial problem for a human, but when faced with many possible actions, the task of finding such sequence become a complex problem. Classical planners have no global view of which actions should be prioritized in order to efficiently decide which actions to be explored during search. This domain knowledge may appear as macros (sequences of actions) (Botea et al. 2005) or hierarchical constructions that require decomposition to obtain a plan (Nau et al. 1999). Hierarchical planners often rely completely on domain knowledge to be able to solve problems orders of magnitude faster than classical planners. Such domain knowledge does not come for free and must be carefully built by a domain expert to consider all possible decompositions that may be used to obtain a solution, or risk having a planner failure, sub-optimal or invalid plan for some scenarios. This knowledge requires a domain expert to consider generalized solutions that solve common sub-problems within the domain, often involving recursive decomposition. As a consequence, describing a domain and its possible decompositions is time consuming, especially when a domain expert needs to test the general solution in order to avoid infinite recursions and ensure that the planner eventually returns a solution when one exists for all valid scenarios.

By analysing a number of existing Hierarchical Task Network (HTN) domains, we notice that similar domains rely on similar methods to solve analogous sub-problems, such as recursively moving until a certain destination is reached. While some descriptions, such as the Action Notation Modeling Language (ANML), allow a debugger to describe such dependency mechanisms (Smith, Frank, and Cushing 2008) to make domain knowledge explicit, we aim to generate domain knowledge using only the information available in a classic domain description. Descriptions of different domains may use different predicates, but share the same construction patterns to solve common sub-problems. Generalizing such construction patterns that appear in the domain operators make it possible to automate the process and obtain task descriptions to solve each sub-problem without brute-forcing operators using a recursive method. By automating the process of task description based on the classical domain description we can save development time while taking advantage of hierarchical planners potential speed-up. In this paper, we address the need for domain knowledge for HTN planners using an approach that automatically generates HTN methods from a classical planning domain. The methods generated by our approach not only improve the efficiency of the resulting HTN planner compared to a brute-force conversion from PDDL (Erol, Nau, and Subrahmanian 1995), but are also readable by a human, allowing adjustments to further improve the efficiency of the generated HTN domain knowledge. The resulting approach can then be applied to both allow HTN planners to be used efficiently to solve classical planning domains with minimal or no expert-reliant knowledge, or enhance hybrid planners such as GoDel (Shivashankar et al. 2013) and obviate the need for human-designed domain knowledge in order to achieve solution speed ups.

2 Background

2.1 Classical Planning

Automated planning is concerned with finding a set of actions that reaches a goal from a initial configuration of the world. In classical planning the goal is represented by a state. States encode properties of the objects in the world at a particular time. In order to achieve the goal state the operators defined in the domain are used as rules to determine which plans are valid based on their preconditions and effects. Preconditions and effects use predicates and free variables that, when unified with objects, enumerate the possible actions to be performed. During the planning process the preconditions of actions are used to test which actions are applicable at each state. If applicable, the action effects can be applied, creating a new state. Preconditions are satisfied when a formula (usually a conjunction of predicates) is valid at the state the action is being applied. The effects contain positive and negative sets that add or remove object properties from the state, respectively. Once we reach a state that satisfies the goal, the sequence of actions taken, starting at the initial state, is the plan or solution (Nebel 2000). The *Planning Domain Definition Language* (PDDL) was created in 1998 with the goal of becoming a standard input for planners (McDermott et al. 1998), in order to allow direct comparisons of efficiency between planning algorithms.

2.2 Hierarchical planning

Hierarchical planning shifts the focus from goal states to tasks to be solved in order to exploit human knowledge about problem decomposition using an hierarchy of methods and operators as the planning domain. This hierarchy is created by non-primitive tasks, which uses methods with preconditions and sub-tasks to decompose according to context. The sub-tasks are also decomposed until only primitive-tasks mapping to operators remain, which results in the plan itself. The goal is implicitly achieved by the plan obtained from the decomposition process. If no decomposition is possible the task is considered unachievable. Unlike classical planning, hierarchical planning only considers what appears during the decomposition process to solve the problem. With domain knowledge the domain description is more complex than the classical planning description, as recursive loops can be described. Recursive loops occur when a non-primitive task is decomposed by a method that contains itself in the sub-tasks, this may be the desired behavior when we need to apply the same set of operators several times until a stop condition is met, for example, to walk until a destination is reached.

Each task is represented by a name, a set of parameters, a set of preconditions, and a set of sub-tasks. Once a non-primitive task is decomposed, the sub-tasks generated replace the current task in the task network. Some tasks may have an empty set of sub-tasks, representing no further decomposition. Backtracking is required for flexibility, as branches may fail during decomposition. Backtracking is costly, but in some cases can be avoided by look-ahead preconditions that check an entire branch of the domain. In some domains it is possible to guide the search directly to a solution or failure. This planning formalism is capable of describing the same domains as STRIPS with a built-in heuristic function tailored to the domain and expert preferences (Lekavý and Návrat 2007), with all the methods required beforehand, which consumes project time to consider every single case.

SHOP (Nau et al. 1999) is one of the best known implementations of HTN planning algorithms. The successors of SHOP, SHOP2 and JSHOP2 (Ilghami and Nau 2003), share most of their algorithm using a more complex decision about which task to decompose at any step in order to support interleaved tasks. Since no standard description exists for HTN, we opted to use the same description used by SHOP2 and JSHOP2. SHOP2 (Nau et al. 2001), for example, supports unordered task decomposition, a feature that separates this planner from its predecessor, SHOP (Nau et al. 1999). JSHOP2 description follows a simplified version of the LISP style adopted by PDDL, without labels for every part of the operator or method. The operator represents the same as the classical operator, an action that can take place in this domain. The operators have a name, a set of parameters and three sets. The first set represents the preconditions, the second set the negative-effects with what is going to be false in the next state, while the final set represents the positive-effects with what is going to be true at the next state. The methods have a name, a set of parameters, a set of preconditions, and instead of effects they have a set of sub-tasks to be performed. Methods can also be decomposed in different ways and have an optional label for each case. The problem contains two sets, the first represents the initial state and the second a list of tasks to be performed. Instead of interpreting the domain and problem, the description is compiled to achieve better results with static structures.

3 Identifying operator patterns

To generate HTN methods based on classical planning descriptions, one must first identify common patterns of operator usage in order to obtain generic methods that could be used in planning domains. Such common patterns are based on how predicates are used by operators. The use of predicates as source of information has already been explored by Pattison and Long (Pattison and Long 2010) in goal recognition, the predicates were partitioned into groups to help differentiate which predicates are more likely to be a goal. Since their focus was on the goal recognition process they made no attempt to see a relation among the operators based on the predicates used, just the likelihood of each predicate being a goal.

Instead of the partitions defined by Pattison and Long, we partition predicates based on their mutability, as shown in Algorithm 1. Predicates that appear only in the initial state are considered *irrelevant*, they make no difference in the action application. However predicates that appear in any action precondition but never as an effect define *constant* relations of the domain. Predicates that appear in the effects of any action represent what is possible to change, the *mutable* relations of the domain. Knowing which predicates are constant helps to prune impossible values for the variables at any state, while mutable predicates can indicate which actions can take place once (adding or removing a feature from the current state) or several times in the same plan. Based on the previous observation of how actions with certain predicate types are used we defined a set operator patterns that once matched against an action can relate to a method that solves its related sub-problem. The following subsections explore such operator patterns.

Algorithm 1 Classification of predicates into irrelevant, constant or mutable

```
1: function CLASSIFY_PREDICATES($predicates, operators$)
2:     ptypes ← Table
3:     pre ← PRECONDITIONS($operators$)
4:     eff ← EFFECTS($operators$)
5:     for each $p \in predicates$ do
6:         if $p \in$ eff
7:             ptypes[p] ← mutable
8:         else if $p \in$ pre
9:             ptypes[p] ← constant
10:        else
11:            ptypes[p] ← irrelevant
12:    return ptypes
```

```
(:action move :parameters (
    ?bot - robot
    ?source ?destination - hallway)
  :precondition (and
    (at ?bot ?source)
    (not (at ?bot ?destination))
    (connected ?source ?destination) )
  :effect (and
    (not (at ?bot ?source))
    (at ?bot ?destination) ) )
```

Listing 1: Move operator with swap pattern in PDDL.

3.1 Swap pattern

Some planning instances require the application of an action several times consecutively, the only difference being the values of the parameters. Such actions usually revolve around swapping the truth value of two instances of the same predicate, such as moving from one place to another affects the predicate *at* in the example action from Listing 1. Once the swaps achieve the predicate required by another action precondition or goal predicate the process can stop. This pattern commonly appears in discretized scenarios where an agent swaps its current position among adjacent and free coordinates in an N-dimensional space, where N is the arity of the position predicate. The position is the predicate that is going to be swapped, while the adjacency is a constraint that implies this operator may be executed several times in order to traverse a discretized space.

This operator pattern is related to the pathfinding subproblem and was already identified and exploited by other planners to speed up search. Hybrid STAN (Fox and Long 2001) is one such planner; it uses a path planner and a resource manager to solve sub-problems with specialized solvers. The operators are classified as swap using Algorithm 2. Swap operators contain a constraint in the preconditions, otherwise the swap would have no restrictions requiring only one operator to solve the sub-problem, and a predicate that is modified from the preconditions to the effects using the same parameters as in the constraint. Since several operators may include the swap pattern over the same predicate, they can be merged into a single method with different constraints, such as a climb operator that changes the agent position like a move operator, but only if there is a wall nearby the current position. Swap identification can also be used to infer that an agent or object will never be at two different configurations in the same state, proving that no plan exists for such goal state. Listing 1 shows the *move* operator with the swap pattern in PDDL, in which a *robot* moves from *source* to *destination* when *source* and *destination* are *connected*. Listing 2 shows two decompositions for the generic *swap_predicate*. The first decomposition acts as the base of the recursion, with the predicate with goal values. The second decomposition applies one more step, marks the current position as visited to avoid loops, recursively decomposes *swap_predicate* and unvisits the previously visited positions to be able to reuse such positions later if needed. Visit and unvisit are internal operations done by bookkeeping operators, prefixed by *!!* in JSHOP.

Algorithm 2 Classification of swap operators

```
1: function CLASSIFY_SWAP($operators, ptypes$)
2:     swaps ← Table
3:     for each op $\in operators$ do
4:         constraints ← CONST_POS_PRECOND(op, ptypes)
5:         $pre^+$ ← MUTABLE_POS_PRECOND(op, ptypes)
6:         $pre^-$ ← MUTABLE_NEG_PRECOND(op, ptypes)
7:         $eff^+$ ← ADD_EFFECTS(op)
8:         $eff^-$ ← DEL_EFFECTS(op)
9:         for each pre $\in$ ($pre^+ \cap eff^-$) do
10:            pre2 ← NAME(pre) $\in eff^+$
11:            if pre2
12:                cparam ← PARAM(pre) $\triangle$ PARAM(pre2)
13:                for c $\in$ constraints do
14:                    if c $\subseteq$ cparam
15:                        swaps[op] ← $\langle$pre, constraint$\rangle$
16:                        break
17:    return swaps
```

3.2 Dependency pattern

In the same way some planning instances require the effects of an action to make another action applicable, fulfilling the preconditions. Such precondition turns the first action effects into a dependency for the second action preconditions to be satisfied and the action applied. The operators are classified as dependency using Algorithm 3. Each two operators are compared to find a match between effects and preconditions of operators that have not already been classified as swap operators. Listing 3 shows the *report* operator with the dependency pattern in PDDL, which requires a *robot* to be *at* the same *location* of a *beacon* to report its status. To achieve the *at* precondition there is a dependency with the operator *move*.

Three cases appear in this pattern. In the first case the goal predicate is already satisfied and no action takes place. In the second case the preconditions of the second action are already fulfilled and the action can be applied immediately. In the third case the precondition of the second action require the first action applied before the second action. This

```
(:method (swap_predicate ?object ?goal)
  base
  ( (predicate ?object ?goal) )
  ()
  using_operator
  (
    (constraint ?current ?intermed)
    (swap_predicate ?object ?current)
    (not (predicate ?object ?goal))
    (not (visited_predicate ?object
        ?intermed))
  )
  (
    (!operator ?object ?current ?intermed)
    (!!visit_predicate ?object ?current)
    (swap_predicate ?object ?goal)
    (!!unvisit_predicate ?object ?current)
  ) )
```

Listing 2: Methods for *swap* operator pattern using JSHOP description.

```
(:action report :parameters (
    ?bot - robot
    ?location - hallway
    ?beacon - beacon)
  :precondition (and
    (at ?bot ?location)
    (in ?beacon ?location)
    (not (reported ?bot ?beacon)) )
  :effect (reported ?bot ?beacon) )
```

Listing 3: Report operator with dependency pattern with move in PDDL.

operator pattern commonly appears, as several distinct actions may be required to fulfill a sequence of preconditions to achieve a goal predicate. Actions that already matched the swap pattern are not even tested against the dependency pattern, otherwise such actions would be classified with a dependency of themselves. Listing 4 shows the three cases defined for the operator dependency pattern using JSHOP description.

3.3 Free-variable pattern

Some predicates that appear in the goal state may not specify enough information to map to a task, like a position that must be occupied, which requires an agent, but no agent is mentioned to bound this variable. Methods can propagate bound variables to be used by operators or other method as tasks are decomposed. Before propagated, free-variables must be bound. In order to unify variables such as the agent we create a new method with the single purpose of unification according to the constant preconditions of the related operator, acting as typed parameters of PDDL. Therefore we add a new level to the hierarchy with a method that simply unifies and propagates to the next level with all variables bound. Listing 5 shows a possible scenario where *op1* requires 3 terms to be applied. Terms *t1* and *t2* are known based on information from goal state, while *t3* is left to be decided at run-time. It is possible to apply directly the original method in cases where the three terms are known. We do not merge both methods in one method to explicitly say that we are looking for the value of *t3*.

Algorithm 3 Classification of dependency operators

```
1: function CLASSIFY_DEPENDENCY($operators$, $ptypes$, $swaps$)
2:   dependencies ← Table
3:   for each op ∈ $operators$ do
4:     $pre^+$ ← POS_PRECOND(op, ptypes, mutable)
5:     $pre^-$ ← NEG_PRECOND(op, ptypes, mutable)
6:     $eff^+$ ← ADD_EFFECTS(op)
7:     $eff^-$ ← DEL_EFFECTS(op)
8:     for each op2 ∈ $operators$ do
9:       swap_op ← $swaps$[op]
10:      swap_op2 ← $swaps$[op2]
11:      if swap_op $\neq \varnothing$ and swap_op2 $\neq \varnothing$ and NAME(swap_op) = NAME(swap_op2)
12:        continue
13:      $pre2^+$ ← POS_PRECOND(op2)
14:      $pre2^-$ ← NEG_PRECOND(op2)
15:      if op = op2 or ($pre2^+ \subseteq eff^+$ and $pre2^- \subseteq eff^-$)
16:        continue
17:      $eff2^+$ ← ADD_EFFECTS(op2)
18:      for each pre ∈ $pre^+$ do
19:        if not NAME(pre) ∈ $eff2^+$
20:          continue
21:        if dependencies[op] = $\varnothing$
22:          dependencies[op] ← Set
23:        APPEND(dependencies[op], $\langle$op2, pre$\rangle$)
24:   return dependencies
```

```
(:method (dependency_first_before_second
    ?param)
  goal_satisfied
  ( (goal_predicate) )
  () )
(:method (dependency_first_before_second
    ?param)
  satisfied
  ( (predicate ?param) )
  ( (!second ?param) ) )
(:method (dependency_first_before_second
    ?param)
  unsatisfied
  ( (not (predicate ?param)) )
  ( (!first ?param) (!second ?param) ) )
```

Listing 4: Methods for *dependency* operator pattern using JSHOP description.

4 Composing methods and tasks

With the predicates classified and each operator related to the operator patterns previously defined we need to relate

```
(:method (three_terms ?t1 ?t2 ?t3)
  apply_op_with_three_terms
  ( (precond1 ?t1 ?t2) (precond2 ?t2 ?t3))
  ( (!op1 ?t1 ?t2 ?t3) ) )
(:method (unify_three_terms ?t1 ?t2)
  unify_term_t3
  ( (precond2 ?t2 ?t3) )
  ( (three_terms ?t1 ?t2 ?t3) ) )
```

Listing 5: Methods for *free-variable* operator pattern using JSHOP description.

the goal state with a set of tasks that achieves each part of the goal using the methods obtained from each pattern. In order to know which method to apply we select the operators that are more closely related to goals, the ones with a goal predicate in their effect list. We can use such goal operators to identify which sub-problems we are trying to solve. If the goal operator matches the dependency pattern, we use methods for each case. When the goal is already satisfied the method decomposes to an empty set of subtasks. When the precondition is already satisfied it only decomposes to the operator that achieves the goal predicate, otherwise it decomposes to the dependency operator and the goal operator. If the goal operator matches the swap pattern, we create a specific swap method for the predicate being swapped containing all operators that match the swap pattern with this predicate. Some methods may depend on other methods, to solve this we decompose the other methods first. This happens when the first operator of a dependency also requires a dependency or an operator used in a dependency is also classified as a swap. A new level in the hierarchy is created as such operators are replaced by their respective methods. With the methods we only need to add tasks with the corresponding objects taken from the goal predicates. If such information is not available in the goal predicates some variables remain free to be unified at run-time. To avoid repeating costly unifications, we use the free-variable pattern to unify a variable as high in the hierarchy as possible to doing repetitive unifications during the search. At the end of the process we have the original set of operators incremented with some bookkeeping operators, the set of generated methods, and the set of tasks replacing the goal predicates. Algorithm 4 shows such steps to convert a classical description to an HTN description using our approach. The operator patterns identified generate methods that are currently added independently of usage, as some methods may hint to the domain expert about the relation among operators even when a method is not connected to the rest of the hierarchy.

5 Use case: Rescue Robot domain

In order to illustrate how our operator patterns can be applied to a concrete domain, we use the rescue robot domain [1] as a use case, as several patterns are identified in the operator set. This domain has a small operator set and can be represented

Algorithm 4 Convert goals to tasks

```
1: function GOALS_TO_TASKS($domain, problem$)
2:     op ← OPERATORS($domain$)
3:     pred ← PREDICATES($domain$)
4:     goals ← GOALS($problem$)
5:     tasks ← Set
6:     met ← Set
7:     ptypes ← CLASSIFY_PREDICATES(op, pred)
8:     swaps ← CLASSIFY_SWAP(op, ptypes)
9:     dependencies ← CLASSIFY_DEPENDENCY(op, ptypes, swaps)
10:    goal_op ← Array
11:    for each o ∈ op do
12:        for each goal ∈ goals do
13:            if goal ∈ EFFECTS(o)
14:                APPEND(goal_op, ⟨goal, o⟩)
15:    ADD_SWAP_METHODS(swaps, op, met, ptypes)
16:    ADD_DEPEND_METHODS(swaps, dependencies, op, met, ptypes)
17:    goal_tasks ← Array
18:    for each m ∈ met do
19:        for each d ∈ DECOMPOSITION(m) do
20:            for each ⟨goal, o⟩ ∈ goal_op do
21:                if o ∈ SUBTASKS(d)
22:                    met2 ← UNIFY_VARIABLES(m, o)
23:                    APPEND(goal_tasks, ⟨goal, met2⟩)
24:    INJECT_METHOD_DEPENDENCIES(swaps, met)
25:    for each ⟨goal, met2⟩ ∈ goal_tasks do
26:        if free-variable ∈ met2
27:            APPEND(tasks, UNIFY_METHOD(met2))
28:        else
29:            APPEND(tasks, met2)
30:    return ⟨op, met, tasks⟩
```

by a 2D map, which means we can explore it deeply without complex constructions. The map contains rooms and hallways as locations where the rescue robot and beacons may be located. The robot must be in the same hallway or room of a beacon to report the status. The set of operators include:

- **Enter** a room connected to the current hallway.
- **Exit** the current room to a connected hallway.
- **Move** from the current hallway to a connect hallway.
- **Report** status of beacon in the current room or hallway.

We use our operator patterns to infer how such operators are related to the problem. The operators **Enter**, **Exit** and **Move** swap the predicate *at*. They all require source and destination to be connected locations, which matches our constraint requirement. **Move** creates a dependency for **Enter**, as **Enter** creates a dependency for **Exit**, but since they are already considered *swap* operators we can prioritize *swap* over *dependency* patterns. Swap *at* method may be needed zero or more times to match the destination *at*, which behaves as shown in Figure 1, without the invisible visit/unvisit operators to control which source and intermediate positions where visited.

[1]The rescue robot domain was created by Kartik Talamadupula and Subbarao Kambhampati.

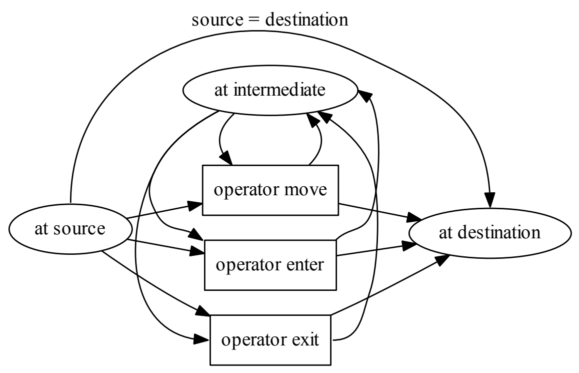

Figure 1: Source, intermediate and destinations are reachable locations the robot may visit using move, enter or exit operations.

Only one operator remains unclassified in this domain, **Report**, which has a precondition *at*. Instead of creating a dependency for each swap operator we can inject the dependency between such methods and make clear that **Report** have a dependency with the swap *at* method previously created, generating a new method. Now this higher level task can be used to report each beacon in the problem, the possible branches the dependency method may take are shown in Figure 2.

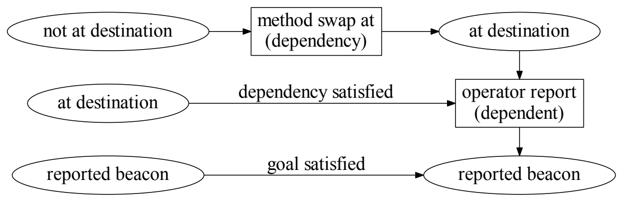

Figure 2: The destination must be reached by the swap at method before any non-reported beacon is reported.

6 Implementation and Experiments

We implemented a domain converter in Ruby, which operates in an intermediate representation after a PDDL domain and problem are parsed. Pattern identification and method composition only have access to the intermediate representation, therefore maintaining the system independent of language and style choices, as special features from the language are downgraded to common supported features. This is the case for PDDL type support, typed objects are added as propositions to the initial state and typed parameters become preconditions. After the goals to tasks process the intermediate representation contains the generated methods and tasks, and we can select either JSHOP or Ruby output. The JSHOP output is useful to a domain expert to extend and/or use with JSHOP. The Ruby output is used by our own implementation of the SHOP algorithm (Nau et al. 1999).

Since some domains require very specific methods we add redundant tasks that use brute-force search for HTN (Erol, Nau, and Subrahmanian 1995) as fallback. To avoid infinite loops in the brute-force mechanism we mark visited actions as they are applied during the recursion, only to unvisit them to release their usage by other tasks. Currently we are limiting actions to be applied at most once by each task. Since our current approach does not verify task interference and order we permute the generated tasks until the original goal state is satisfied. Such behavior can be emulated by other planners using order constraints.

We have tested our approach with multiple domains to discover variations of the operator patterns identified. Our approach took 0.1s or less to generate HTN domain knowledge using the patterns in this paper, and thus it is very efficient in terms of the overall time it can save during search. We performed our experiments using the Windows operating system running on an Intel E5500 2.8GHz CPU with 2GB of RAM. The classical planner (CP) used in the comparison was also developed in Ruby and is doing Breadth-first search with a bitset state representation. In a small set of problems for the Rescue Robot domain, Table 1, we can see that the patterns found were enough to greatly speed up plan search when compared with the pure Brute-Force (BF) approach. Consider the Goldminers domain at Table 2, in which two problems with 10x10 grids contain agents that must move to pick gold and deposit at certain positions. State-space planners suffer with more positions, gold and agents available, while an HTN can focus its search and solve such problems much faster. The sequences of movement actions is where HTN can focus its search in the experiments. More complex domains, such as the ones from ICAPS, still require human intervention to either complete or correct the domain with knowledge that was not inferred from the provided PDDL. Such as the Floortile domain in which an agent can move to the four cardinal directions and paint either to its north or south position, our approach fails to see that both actions are required to color the top and bottom rows of the grid, which returns failures for solvable problems. Other domains such as the Grid, requires an agent to collect keys to open doors in a labyrinth scenario to reach a goal position, which requires several journeys to move towards key, door and goal. Our solution only generates methods to fulfill a single journey, making problems with several doors unsolvable.

Table 1: Rescue Robot tested with diverse planners using 100s as time-out.

Problem	CP	HTN BF	HTN Patterns + BF
pb1	0.001	0.044	0.067
pb2	0.002	11.190	0.255
pb3	0.009	Time-out	0.072
pb4	0.004	20.353	0.197
pb5	0.001	96.979	0.218
pb6	0.001	Time-out	0.132

Table 2: Goldminers tested with diverse planners using 100s as time-out.

Problem	CP	HTN BF	HTN Patterns + BF
pb1	Time-out	Time-out	6.270
pb2	Time-out	Time-out	3.668

7 Conclusions and Future Work

In this paper we have developed an approach to automatically generate HTN domain knowledge using a PDDL specification. Our approach relies exclusively on a number of patterns of state changes we identified as common in most planning domains, therefore obviating the need for example plans.

Existing work has investigated techniques to bridge the gap between classical planning and HTN in multiple ways, however, most such work either require a dataset comprised of a number of solution plans or generated methods that are not competitive with a fast classical planner. The first approach comparable to ours is the brute-force conversion (Erol, Nau, and Subrahmanian 1995). Although this approach translates any PDDL problem into an HTN one that generates equivalent plans, the resulting domain knowledge is not competitive with current classical planners. Indeed, the translation proposed by Erol *et al.* (with some modifications) is our fallback approach when neither of the patterns we found apply. The approach from Lotinac and Jonsson (Lotinac and Jonsson 2016) is the most comparable to ours and generates HTNs from invariance analysis (Lotinac and Jonsson 2016). Finally, the GoDeL (Shivashankar et al. 2013) planner is an hybrid approach that uses methods with sub-goals and landmarks to guide search. Here, instead of trying to decompose a task, methods generate plans to achieve parts of a state-based goal, and uses a classical planner as a fallback option when methods fail or are insufficient. GoDeL's approach is able to perform better if a domain expert supplies more domain knowledge while performing as a classical planner if only classical operators are supplied.

Empirical evaluation has shown that our approach is capable of not only generating valid HTN methods for domains that relate with our operator patterns, it also generates efficient HTN method libraries that can greatly speed-up search. Nevertheless, the HTN knowledge generated for many domains does not allow an HTN planner using blind search to surpass a fast classical planner. Domains in which most of our patterns apply tended to result in better performance, whereas domains that relied on a brute force translation of a PDDL task into HTN methods did worst. Thus, as future work, we aim to investigate mechanisms to further improve the efficiency of the resulting HTN domain knowledge and its interaction with more advanced HTN planners. First, we aim to search for new patterns that could be applicable to the remaining domains. Second, we will evaluate the performance of the methods we generate with HTN planners that selectively choose methods for decompositions rather than performing blind search.

We acknowledge the support given by CAPES/Pro-Alertas (88887.115590/2015-01) and CNPQ.

References

[Botea et al. 2005] Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research* 24:581–621.

[Erol, Nau, and Subrahmanian 1995] Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence* 76(1-2):75–88.

[Fox and Long 2001] Fox, M., and Long, D. 2001. Hybrid STAN: Identifying and managing combinatorial optimisation sub-problems in planning. In *Proceedings of International Joint Conference on Artificial Intelligence*, 445–452. Morgan Kaufmann.

[Ilghami and Nau 2003] Ilghami, O., and Nau, D. S. 2003. A general approach to synthesize problem-specific planners. Technical report, DTIC Document.

[Lekavý and Návrat 2007] Lekavý, M., and Návrat, P. 2007. Expressivity of strips-like and htn-like planning. In *Agent and Multi-Agent Systems: Technologies and Applications*. Springer. 121–130.

[Lotinac and Jonsson 2016] Lotinac, D., and Jonsson, A. 2016. Constructing hierarchical task models using invariance analysis. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI-16)*.

[McDermott et al. 1998] McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems 1998 (AIPS'98)*.

[Nau et al. 1999] Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, 968–973. Morgan Kaufmann Publishers Inc.

[Nau et al. 2001] Nau, D.; Munoz-Avila, H.; Cao, Y.; Lotem, A.; and Mitchell, S. 2001. Total-order planning with partially ordered subtasks. In *Proceedings of International Joint Conference on Artificial Intelligence*, volume 1, 425–430.

[Nebel 2000] Nebel, B. 2000. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research* 12:271–315.

[Pattison and Long 2010] Pattison, D., and Long, D. 2010. Domain Independent Goal Recognition. In Ågotnes, T., ed., *STAIRS*, volume 222 of *Frontiers in Artificial Intelligence and Applications*, 238–250. IOS Press.

[Shivashankar et al. 2013] Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. 2013. The GoDeL planning system: a more perfect union of domain-independent and hierarchical planning. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, 2380–2386. AAAI Press.

[Smith, Frank, and Cushing 2008] Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML language. In *The*

ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling.