Otto-Friedrich-Universität Bamberg

Lehrstuhl für Smart Environments

Projektbericht

SME-Projekt-B

Erarbeitet von:
Maximilian Götz, Julian Leisin

Betreuer:
Prof. Dr. Dietrich Wolter
Tobias Schwartz M.Sc.

Contents

1 Introduction

There are often situations in which something repetitve has to be done, like moving boxes from one position to another. It is nearly always the same procedure with some edge cases which can occur from time to time. Since the actions are not varying, think about the boxes which are taken from a pile and are placed on another pile, there is no need to have a human to control the machinery which does the action. Instead, the machine can work autonomously by using a plan. The plan describes what has to be done and which decisions needs to be made in which situation. Planning is used in several areas and systems, so there are also different approaches on creating these plans. With different types of plans existing, it becomes necessary to create an automated translation between them. We will focus on HTN (Hierarchical Task Network) and $STRIPS$ (Stanford Research Institute Problem Solver) and the translation from HTN to $STRIPS$.

2 HTN

HTN problems are characterised by their hierarchical structure. They consist of primitive tasks which can form compound tasks. A HTN problem P can be described as a tuple (D, s_I, tn_I) where D is the domain, $s_I \in s^F$ an initial state and tn_I an initial task network. The domain D is a tuple (F, N_P, N_C, δ, M). F is the finite set of facts, N_P the finite set of primitive task names, N_C the finite set of compound task names, δ the action mapping $\delta : N_P \mapsto 2^F \times 2^F \times 2^F$ which assigns action to primitive task names and M is the finite set of decomposition methods. [1]
The HTN problem can then be solved by either using A^* or compiling it to a different problem class like $STRIPS$ in which it is then solved.

3 STRIPS

$STRIPS$ problems can be expressed as a quadruple (P, O, I, G). P is a finite set of ground atomic formulas, O is a finite set of operators, I is the initial state and therefore a subset of P. Finally, there is the goal state G. The elements in P are also called conditions. O has the form $Preconditions \rightarrow Postconditions$ where the pre- and postconditions are a 'satisfiable conjunction of positive and negative conditions'.[2] The same goes for G which is also a 'satisfiable conjunction of positive and negative conditions'.[3]

4 Compiling HTN to STRIPS

4.1 General approach

Now we want to translate the HTN problem to one in $STRIPS$. To do so, we basically only have to rewrite all the methods and subtasks to $STRIPS$-actions. The methods consist of a unique name, a finite list of paramters, another finite list for the subtasks and a task. Since there might be a subtask with the same name as a method, we add a

[1] See: Daniel (2019): A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations

[2] See: The Computational Complexity of Propositional STRIPS Planning

[3] See: The Computational Complexity of Propositional STRIPS Planning

uppercase M to the methods name at the beginning to guarantee that the name is unique in the list of strips-actions. The list of parameters can be taken over with the condition, that special characters have to be skipped and keywords have to be resolved. This will be discussed in detail later on. Each element of the list of subtasks must be resolved to an effect of the corresponding action and the task is the postconidition of the $STRIPS$-action.
The HTN-actions, which are a subset of the subtasks, also have to be translated to $STRIPS$-actions. The name of each action is kept, this time without adding anything to it. The list of paramters again is processed to remove all special characters and to resolve all the keywords. Same goes for the preconditions where the keywords are resolved as well. They are set as the $STRIPS$-action's preconditions. Finally, the set effect is set as the postconditions.
Since we assume that the given HTN problem is formulated correctly, the following step is unnecessery. Yet, it would be necessery if we want to check whether the problem is formulated correctly or not. Due to our assumption, we do not validate the problem. Nevertheless, we set the basics for a validation if we want to implement it in the future. The tasks are saved in a list, containing the name, which has a leading uppercase T, and the list of parameters.
The last two things missing are the inital and the goal state. We get the inital and the goal state from the problem.hddl file. $tasks$ is set as the goal. It can contain multiple elements, but our translation only works with one task. Therefore, every HTN problem that should be translated must contain exactly one element $task \in tasks$. On the other hand, the elements of $init$ are used as the elements of the initial state of the $STRIPS$ problem and can contain multiple elements.

4.2 Special characters and keywords

The given HTN problem might contain different kinds of special characters. Since the notation in kebab-case is pretty common when setting up a HTN problem, we have to remove all dashes as our $STRIPS$ would not work otherwise. Additionally, the file contains special keywords like *not* and *and*. They are meant to be logical operators. The $STRIPS$ planner expects these operators in a different notation, i.e. every *not* has to be rewritten to an ! and instead of *and*, we use commas to indicate the logical *and*.
So, $(not(and(statement - a\ param - a)(statement - b\ param - b)))$ would be rewritten to $(!((statemenetA\ paramA), (statementB\ paramB)))$. The parameters and variables in HTN have a leading questionmark, which has to be removed as well as the typing. I.e., a statement like $parameters(?l - location)$ would be restructured to $parameters(l)$.

4.3 Example by hand

In order to have a feeling about the problem of compiling from HTN to $STRIPS$ and to see how a result should look like, we decided to translate a HTN problem to a $STRIPS$ problem by hand. The following tea brewing example was created together with the htn2sat group.

Listing 1: domain.hddl

```
(define (domain tea)
        (:requirements :negative-preconditions :typing :hierarchy)
        (:types cup location kettle water teabag - object)
        (:predicates
                (hot ?w - water)
                (is-in ?o1 - object ?o2 - object)
                (at-location ?lk - location)
                (in-hand ?o - object))
        (:task brew-tea
                :parameters (?lk - location ?c - cup ?k - kettle ?w - water
                    ?t - teabag))
        (:task boil-water
                :parameters (?k - kettle ?w - water ?lk - location))

        (:method m-brew-tea
                :parameters (
                        ?lk - location
                        ?c - cup
                        ?k - kettle
                        ?w - water
                        ?t - teabag)
                :task (brew-tea ?lk ?c ?k ?w ?t)
                :ordered-subtasks (and
                        (get-to ?lk)
                        (boil-water ?k ?w ?lk)
                        (pick-up ?t)
                        (put-in ?t ?c)
                        (pour-water ?w ?c)))

        (:method m-boil-water
                :parameters (?k - kettle ?w - water ?lk - location)
                :task (boil-water ?k ?w ?lk)
                :ordered-subtasks (and
                        (pick-up ?k)
                        (pick-up ?w)
                        (put-in ?w ?k)
                        (turn-on ?k)))

        (:action get-to
                :parameters (?lk ?lz  - location)
                :precondition (and
                        (not (at-location ?lk))
                        (at-location ?lz))
                :effect (and
                        (at-location ?lk)
                        (not (at-location ?lz))))

        (:action pick-up
                :parameters (?o1 - object ?lk - location)
                :precondition (and
                        (not (in-hand ?o1))
```

```
                        (at-location ?lk))
                :effect (in-hand ?o1))

        (:action turn-on
                :parameters (?k - kettle ?w - water)
                :precondition (is-in ?w ?k)
                :effect (hot ?w))

        (:action put-in
                :parameters (?o1 ?o2 - object)
                :precondition (and
                        (not (is-in ?o1 ?o2))
                        (in-hand ?o1))
                :effect (and
                        (is-in ?o1 ?o2)
                        (not(in-hand ?o1))))

        (:action pour-water
                :parameters (?w - water ?c - cup ?k - kettle)
                :precondition (and
                        (hot ?w)
                        (in-hand ?k))
                :effect (is-in ?w ?c)))
```

Listing 2: problem.hddl

```
(define (problem p)
        (:domain tea)
        (:objects c - cup lk lh - location k - kettle w - water t - teabag)
        (:htn
                :tasks (brew-tea lk c k w t)
                :ordering ()
                :constraints ())
        (:init
                (at-location lh)))
```

The first step was picking the *init* and the *tasks* from problem.hddl.

Listing 3: Creation of initial and goal state

```
        :tasks (brew-tea lk c k w t)
        :init (at-location lh)
```

As described in chapter 4.2, special characters have to be removed.

Listing 4: Creation of initial and goal state cont'd

```
        :tasks (brewtea lk c k w t)
        :init (atlocation lh)
```

Now we replace : *tasks* with *Goal state* and : *init* with *Initial state*. Furthermore, we put the parameters into brackets and seperate them with commata such that it looks like this:

Listing 5: Creation of initial and goal state cont'd

```
        Initial state: atlocation(lh)
        Goal state: tbrewtea(lk,c,k,w,t)
```

After setting up the initial and the goal state, we move on to the actions. First, we remove all the special characters and the typing and rewrite the keywords.

Listing 6: Creation of the actions

```
        (:method mbrewtea
                :parameters (lk c k w t)
                :task (brewtea lk c k w t)
                :ordered-subtasks (
                        (getto lk),
                        (boilwater k w lk),
                        (pickup t),
                        (putin t c),
                        (pourwater w c)))

        (:method mboilwater
                :parameters (k w lk)
                :task (boilwater k w lk)
                :ordered-subtasks (
                        (pickup k),
                        (pickup w),
                        (putin w k),
                        (turnon k)))

        (:action getto
                :parameters (lk lz)
                :precondition (!(atlocation lk), (atlocation lz))
                :effect ((atlocation lk), !(atlocation lz)))

        (:action pickup
                :parameters (o1 lk)
                :precondition (!(inhand o1), (atlocation lk))
                :effect (inhand o1))

        (:action turnon
                :parameters (k w)
                :precondition (isin w k)
                :effect (hot w))

        (:action putin
                :parameters (o1 o2)
                :precondition ( !(isin o1 o2), (inhand o1))
                :effect ((isin o1 o2),  !(inhand o1)))

        (:action pourwater
                :parameters (w c k)
                :precondition ((hot w), (inhand k))
                :effect (isin w c))
```

The final step is to rewrite the actions and methods, so they fit the *STRIPS* syntax.

Listing 7: Creation of the actions cont'd

```
Initial state:      Atlocation(hall)
Goal state:      Tbrewtea(kitchen,cup,kettle,water,tea)

Actions:
        Mbrewtea(lk,c,k,w,t)
        Preconditions: Atlocation(lk), Tboilwater(k,w,lk), Isin(t,c), Isin(
            w,c)
        Postconditions: Tbrewtea(lk,c,k,w,t)

        Mboilwater(k,w,lk)
        Preconditions: Inhand(k), Isin(w,k), Hot(w)
        Postconditions: Tboilwater(k,w,lk)
```

```
12
13          Getto(lk, lz)
14          Preconditions: !Atlocation(lk), Atlocation(lz)
15          Postconditions: Atlocation(lk), !Atlocation(lz)
16
17          Pickup(o1, lk)
18          Preconditions: !Inhand(o1), Atlocation(lk)
19          Postconditions: Inhand(o1)
20
21          Turnon(k,w)
22          Preconditions: Isin(w,k)
23          Postconditions: Hot(w)
24
25          Putin(o1, o2)
26          Preconditions: !Isin(o1, o2), Inhand(o1)
27          Postconditions: Isin(o1, o2), !Inhand(o1)
28
29          Pourwater(w, c, k)
30          Preconditions: Hot(w), Inhand(k)
31          Postconditions: Isin(w, c)
```

Note: The resulting $STRIPS$ plan differs from the resulting HTN plan. To solve this, one would had to remove $inhand(t)$ in line 6 and $inhand(w)$ in line 10. The reason for this will be discussed later on.

4.4 The actual code

4.4.1 htn2strips

In order to work in a simple way with the read and preprocessed data, we created the structs $strips-action$, $strips-task$, $strips-init$ and $strips-goal$.
The main method *htn2strips* takes the domain and the problem file as parameters, calls the functions which process them, saves them aferwards and prints them to *output.txt*.

Listing 8: htn2strips

```
1  (defun htn2strips (domain-file problem-file)
2          "Main method: takes a domain-file and problem-file in HTN format"
3          (let ((htn-task '())
4                  (htn-method '())
5                  (htn-action '())
6                  (htn-init '())
7                  (htn-goal '())
8                  (strips-task '())
9                  (strips-method '())
10                 (strips-action '())
11                 (strips-init '())
12                 (strips-goal '())
13                 (domain '())
14                 (problem '()))
15
16                 (declare (ignorable strips-task))
17                 ;; reader makro to remove '?'
18                 (let ((*readtable* (copy-readtable)))
19                 (set-macro-character #\? #'question-reader)
20                 ;read and save domain/problem hddl file
21                         (setq domain (subst '! 'NOT (read-file domain-file)
                               ))
22                         (setq problem (subst '! 'NOT (read-file
                               problem-file))))
23
24                 ;get and save tasks, methods and actions in lists
25                 (setq htn-task (get-from-htn domain ':task))
26                 (setq htn-method (get-from-htn domain ':method))
27                 (setq htn-action (get-from-htn domain ':action))
28                 (setq htn-init (cdr (first (get-from-htn problem ':init))))
29                 (setq htn-goal (parse-it :tasks (first (get-from-htn
                       problem ':htn))))
30
31                 ;transfrom htn to strips
32                 (setq strips-action (parse-actions htn-action))
33                 (setq strips-method (parse-methods htn-method strips-action
                       ))
34                 (setq strips-task (parse-tasks htn-task))
35                 (setq strips-init (parse-init htn-init))
36                 (setq strips-goal (parse-goal htn-goal))
37
38                 (write-file strips-method strips-init strips-goal
                       strips-action))
39
40         "Complete")
```

In line 18 to 22, the files are read, questionmarks are removed with the help of an reader macro and every occurence of NOT has been substituted with an '!'.
The needed parts of the HTN problem gets extracted in the lines 24 to 29 and are saved

in the respective local variables. From this point on, the pieces from the HTN problem are parsed to the desired form and the returning structs are saved in lists.
Finally, the $write - file$ function, which constructs the *output.txt*, is called. If the programm terminates, it returns 'Complete'.

4.4.2 parse-actions

Listing 9: parse-actions

```
(defun parse-actions (htn-actions)
        "Takes htn-action and turns it into a strips-action struct"
        (loop for element in htn-actions
                collect
                (make-strips-action
                        :parameters (remove-item-and-next '- (parse-it :
                            parameters element))
                        :name (remove-hyphen (second element))
                        :preconditions (remove-hyphen (parse-it :
                            precondition element))
                        :postconditions (empty-postconditions (
                            remove-hyphen (parse-it :effect element))
                            element))))
```

The function $parse - actions$ takes the list of all $htn - action$ as a parameter, loops for each element and creates a struct of the type $strips - action$. The action's name is the old name and the parameters are kept as well but the typing is removed using the function $remove - item - and - next$. This helper function takes a symbol and a list. It scans the list and removes every occurence of the symbol together with the following element. The resulting list is then returned. The preconditions are also kept after the hyphens have been removed and the preconditions are the elements of the effect set from the HTN problem.
If the list of postconditions is empty, then a new postcondition is created by adding the prefix 'E' to the action's name. By this we make sure that there is always a postcondition which can be resolved by the $STRIPS$ implementation.

4.4.3 parse-method

Listing 10: parse-methods

```
(defun parse-methods (htn-methods strips-action)
        "Takes htn-method and turns it into a strips-method struct"
        (loop for element in htn-methods
                collect
                (make-strips-action
                :parameters (remove-item-and-next '- (parse-it :parameters
                    element))
                :name (add-prefix-to-element "M" (list (remove-hyphen (
                    second element))))
                :preconditions (substitute-name (remove-hyphen (parse-it :
                    ordered-subtasks element)) strips-action)
                :postconditions (append (list (add-prefix-to-element "T" (
                    list (remove-hyphen (first (parse-it :task element))))))
                     (cdr (remove-hyphen (parse-it :task element)))))))
```

$parse - methods$ accepts the list of all HTN methods and the list containing the action-structs as parameters. It then loops over all methods and rewrites them to $STRIPS$

actions. Parameters and postconditions are handled as in $parse-actions$. An uppercase M is added to the name, to indicate that this action has been a method. In order to convert the subtasks to the precondition, each element has to be resolved to an positive postcondition of an already existing action. To achieve this, the method $substitute-name$ substitutes the subtask with the name of the first positive postcondition from the corresponding action.

4.4.4 parse-tasks

Listing 11: parse-task

```
(defun parse-tasks (htn-tasks)
        "Takes htn-task and turns it into a strips-task struct"
        (loop for element in htn-tasks
        collect
                (make-strips-task
                        :parameters (remove-item-and-next '- (parse-it :
                            parameters element))
                        :name (add-prefix-to-element "T" (list (
                            remove-hyphen (second element)))))))
```

This function accepts the list of all HTN tasks, creates a struct with their data, where a leading T is added to the name, and returns them.

4.4.5 parse-init

Listing 12: parse-init

```
(defun parse-init (htn-init)
        "Takes htn-init and turns it into a strips-init struct"
        (loop for element in htn-init
        collect
                (make-strips-init
                        :parameters (remove-hyphen (cdr element))
                        :name (remove-hyphen (first element)))))
```

The $parse-init$ function takes the $htn-init$, which has been retrieved from the problem file in the $htn2strips$ function. The name and parameters are then saved in a struct to make the printing later on more easy.

4.4.6 parse-goal

Listing 13: parse-goal

```
(defun parse-goal (htn-goal)
        "Takes htn-goal and turns it into a strips-goal struct"
        (make-strips-goal
                :parameters (remove-hyphen (cdr htn-goal))
                :name (add-prefix-to-element "T" (list (remove-hyphen (
                    first htn-goal))))))
```

$parse-goal$ saves the name and the parameters from the input $htn-goal$ in the respective struct. We don't need a loop here, as we expect that there is only one task given in the problem file. The reasons behind this decision will be discussed later on.

4.4.7 write-file

Listing 14: write-file

```
(defun write-file (strips-init strips-goal strips-action)
        "Formats and writes the resulting data to output.txt"
        (with-open-file (file #P"output.txt" :direction :output :if-exists
                              :supersede :if-does-not-exist :create)

                (format file "Initial state: ")
                (loop for init in (butlast strips-init)
                        finally
                        (format file "~12,0T~a" (strips-init-name (first (
                           last strips-init))))
                        (format file "(~{~a~^,~})" (strips-init-parameters
                           (first (last strips-init))))
                        do      (format file "~12,0T~a" (strips-init-name
                           init))
                                (format file "(~{~a~^,~})" (
                                   strips-init-parameters init))
                                (format file ","))
                (fresh-line file)

                (format file "Goal state: ")

                (format file "~12,0T~a" (strips-goal-name strips-goal))
                (format file "(~{~a~^,~})" (strips-goal-parameters
                   strips-goal))

                (fresh-line file)
                (format file "Actions:")
                (fresh-line file)

                (loop for action in strips-action do
                        (format file "~12,0T~a" (strips-action-name action)
                           )
                        (format file "(~{~a~^,~})~%" (
                           strips-action-parameters action))
                        (format file "~12,0TPreconditions: ")
                        (format file "~a ~%" (process-conditions (
                           strips-action-preconditions action) (make-array
                           0 :element-type 'character :fill-pointer 0)))
                        (format file "~12,0TPostconditions: ")
                        (format file "~a ~%" (process-conditions (
                           strips-action-postconditions action) (make-array
                            0 :element-type 'character :fill-pointer 0)))
                        (terpri file))))
```

In the $write-file$ function, the processed data is formatted and brought into a $STRIPS$ compliant output. To achieve the needed form, the header is printed first. It consists of the inital and the goal state. Therefore, the function prints the key words ('Initial state: ', Goal state :') followed by the data from the corresponding structs which is brought in the form $Name(param1, param2, ...)$. The initial state can contain multiple states whereas the goal state contains only one state.
After that, the list of the action begins. The beginning is indicated by the keyword 'Actions: ' followed by the individual actions which have the following form.

```
1          Name(param1, param2, ...)
2          Preconditions: Name(param1, param2, ...), Name(param1, param2, ...)
               ,...
3          Postconditions: Name(param1, param2, ...), Name(param1, param2,
               ...) ,...
```

Everything is printed in uppercase.

4.5 More examples

4.5.1 Tea example translated by the program

Feeding the algorithm with the domain.hddl and problem.hddl from Listing 1 and 2 prints the following output.

Listing 15: STRIPS problem

```
1
2  Initial state: ATLOCATION(LH)
3  Initial state: ATLOCATION(LH)
4  Goal state: TBREWTEA(LK,C,K,W,T)
5  Actions:
6          MMBREWTEA(LK,C,K,W,T)
7          Preconditions: ATLOCATION(LK) ,TBOILWATER(K,W,LK) ,INHAND(T) ,ISIN(T,C
               ) ,ISIN(W,C)
8          Postconditions: TBREWTEA(LK,C,K,W,T)
9
10         MMBOILWATER(K,W,LK)
11         Preconditions: INHAND(K) ,INHAND(W) ,ISIN(W,K) ,HOT(K)
12         Postconditions: TBOILWATER(K,W,LK)
13
14         GETTO(LK,LZ)
15         Preconditions: !ATLOCATION(LK) ,ATLOCATION(LZ)
16         Postconditions: ATLOCATION(LK) ,!ATLOCATION(LZ)
17
18         PICKUP(O1,LK)
19         Preconditions: !INHAND(O1) ,ATLOCATION(LK)
20         Postconditions: INHAND(O1)
21
22         TURNON(K,W)
23         Preconditions: ISIN(W,K)
24         Postconditions: HOT(W)
25
26         PUTIN(O1,O2)
27         Preconditions: !ISIN(O1,O2) ,INHAND(O1)
28         Postconditions: ISIN(O1,O2) ,!INHAND(O1)
29
30         POURWATER(W,C,K)
31         Preconditions: HOT(W) ,INHAND(K)
32         Postconditions: ISIN(W,C)
```

To check whether there exists a valid plan to this problem, the output was fed to a *STRIPS* planner which printed this plan:

Listing 16: STRIPS plan

```
1          Plan:
2          GETTO(LK, LH) -> PICKUP(W, LK) -> PUTIN(W, C) ->
3          TURNON(C, W) -> PICKUP(C, LK) -> POURWATER(W, K, C) ->
4          PICKUP(K, LK) -> PUTIN(K, C) -> TURNON(C, K) ->
5          PICKUP(K, LK) -> PICKUP(W, LK) -> MMBOILWATER(K, W, LK) ->
```

```
6           PICKUP(T, LK) -> PUTIN(T, C) -> TURNON(C, T) ->
7           POURWATER(T, C, K) -> PICKUP(T, LK) -> MMBREWTEA(LK, C, K, W, T)
```

However, the *HTN* planner prints this:

Listing 17: HTN plan

```
1           SOLUTION SEQUENCE
2           0: get-to(lk,lh)
3           1: pick-up(k,lk)
4           2: pick-up(w,lk)
5           3: put-in(w,k)
6           4: turn-on(k,w)
7           5: pick-up(t,lk)
8           6: put-in(t,c)
9           7: pour-water(w,c,k)
```

The *STRIPS* plan still has some errors. They are caused by resolving INHAND(T) in line 6 and INHAND(W) line 10. Also, HOT(K) in line 11 must be HOT(W) in the *STRIPS* problem file. While the error with HOT(W) occurs due to the missing typing in *STRIPS*, we are not sure if the error in the plans are due to errors in the *STRIPS* planner or due to mistakes in the problem.hddl itself.

4.5.2 Truck example translated by the program

The second example, which was published on the VC.

Listing 18: Truck HTN problem.hddl

```
1 (define (problem p)
2           (:domain  domain_htn)
3           (:objects
4                   city-loc-0 city-loc-1 city-loc-2 - location
5                   truck-0 - vehicle
6           )
7           (:htn
8           :tasks (and
9                   (get-to truck-0 city-loc-0)
10                  (get-to truck-0 city-loc-1)
11          )
12          :ordering ( )
13          :constraints ( ))
14          (:init
15                  (road city-loc-0 city-loc-1)
16                  (road city-loc-1 city-loc-0)
17                  (road city-loc-1 city-loc-2)
18                  (road city-loc-2 city-loc-1)
19                  (at truck-0 city-loc-2)
20          )
21          )
```

Listing 19: Truck HTN domain.hddl

```
1 (define (domain transport)
2           (:requirements :negative-preconditions :hierarchy :typing)
3           (:types
4           location vehicle - object
5           )
6
7           (:predicates
8                   (road ?l1 ?l2 - location)
```

```
9                 (at ?x - vehicle ?v - location)
10        )
11
12        (:task get-to :parameters (?v - vehicle ?l - location))
13
14        (:method m-drive-to
15                :parameters (?v - vehicle ?l1 ?l2 - location)
16                :task (get-to ?v ?l2)
17                :subtasks (and
18                (drive ?v ?l1 ?l2))
19        )
20
21        (:method m-drive-to-via
22                :parameters (?v - vehicle ?l2 ?l3 - location)
23                :task (get-to  ?v ?l3)
24                :ordered-subtasks (and
25                (get-to ?v ?l2)
26                (drive ?v ?l2 ?l3))
27        )
28
29        (:method m-i-am-there
30                :parameters (?v - vehicle ?l - location)
31                :task (get-to  ?v ?l)
32                :subtasks (and
33                (noop ?v ?l))
34        )
35
36        (:action drive
37                :parameters (?v - vehicle ?l1 ?l2 - location)
38                :precondition (and
39                (at ?v ?l1)
40                (road ?l1 ?l2))
41                :effect (and
42                (not (at ?v ?l1))
43                (at ?v ?l2))
44        )
45
46        (:action noop
47                :parameters (?v - vehicle ?l2 - location)
48                :precondition (at ?v ?l2)
49                :effect ()
50        )
51
52        )
```

The corresponding problem in *STRIPS*:

Listing 20: STRIPS problem

```
1 Initial state:ROAD(CITYLOC0,CITYLOC1),ROAD(CITYLOC1,CITYLOC0),
2         ROAD(CITYLOC1,CITYLOC2),ROAD(CITYLOC2,CITYLOC1),AT(TRUCK0,CITYLOC2)
3 Goal state: TGETTO(TRUCK0,CITYLOC0)
4 Actions:
5         MMDRIVETO(V,L1,L2)
6         Preconditions: AT(V,L1,L2)
7         Postconditions: TGETTO(V,L2)
8
9         MMDRIVETOVIA(V,L2,L3)
10        Preconditions: TGETTO(V,L2),AT(V,L2,L3)
11        Postconditions: TGETTO(V,L3)
12
```

```
13          MMIAMTHERE(V,L)
14           Preconditions: ENOOP(V,L)
15           Postconditions: TGETTO(V,L)
16
17          DRIVE(V,L1,L2)
18           Preconditions: AT(V,L1),ROAD(L1,L2)
19           Postconditions: !AT(V,L1),AT(V,L2)
20
21          NOOP(V,L2)
22           Preconditions: AT(V,L2)
23           Postconditions: ENOOP(V,L2)
```

The *STRIPS* planner gives out the following plan:

Listing 21: STRIPS plan

```
1 Plan:   DRIVE(TRUCK0, CITYLOC2, CITYLOC1) ->
2                 DRIVE(TRUCK0, CITYLOC1, CITYLOC0) ->
3                 NOOP(TRUCK0, CITYLOC0) ->
4                 MMIAMTHERE(TRUCK0, CITYLOC0)
```

While the *HTN* plan looks like this:

Listing 22: HTN plan

```
1 SOLUTION SEQUENCE
2         0: drive(truck-0,city-loc-2,city-loc-1)
3         1: drive(truck-0,city-loc-1,city-loc-0)
```

This time, the translation worked like intended.

4.6 Allowed input and restrictions

For a correct output, it is necessary that a problem.hddl and a domain.hddl file meet the following criteria. First of all, the two file have to be in a valid hddl format. Furthermore, the problem file has to containt exactly one task element but can contan multiple init elements. This is needed to make sure, that the resulting *STRIPS* problem has only one goal state and can be solved.[4]. Also, subtasks have to be ordered as well.

4.7 Covered edge cases

As already seen in the example with the truck, it can happen that HTN has actions that have no effect. This is not a problem with HTN, because here the actions are identified by their name. With Strips, however, this is different, here the required actions are identified by their postconditions. In order to sovle this, empty postconditions, NOOP, are created.

4.8 Uncovered edge cases

Several edge cases remain uncovered. For example, the constraints and the ordering in the problem.hddl file are not covered. As seen in one of the previous chapters, the tea example has some problems with resolving HOT and INHAND. The problem with HOT is, that in the hddl file a action with two parameters is given and a postcondition with a predicate with only one parameter. This leads to the circumstance, that when the action name is resolved to the postcondition, we have to decide which parameter to keep. In this

[4]See: Bound to Plan: Exploiting Classical Heuristics via Automatic Translations of Tail-Recursive HTN Problem

example, when there are multiple paramters, the first parameter is used. Due to that, it is possible that the wrong parameter is passed on. Sadly, we haven't found a solution that solves this problem yet. A possible solution would be, to have additional constraint for the domain file so that those effected parameters are always in the first position.
The second edge case occurs with INHAND in lines 7 and 11. With HTN these instructions are needed so that the preconditions for PUTIN are fulfilled. With $STRIPS$ however this leads to an error, since here the parameter O1 is replaced with a parameter, in our example with C, which fulfills the postconditions. A solution for this would be to delete the INHAND statements, which are used in connection with the parameter O1 in PUTIN. This can also be seen in the working example of the tea example.

5 Conclusion

As can be seen from our examples, the algorithm can translate simpler plans very well from HTN to Strips. Problems occur with more complex plans or plans that make greater use of the properties of HTN. However, to make a better statement about the accuracy of our algorithm, it would have to be tested with more examples. Also, with further testing it should be achievable to reliably fix the edge cases that are not yet covered.

6 Bibliography

1. Alford, Ron / Behnke, Gregor / Höller, Daniel / Bercher, Pascal / Biundo, Susanne / Aha, David W.: Bound to Plan: Exploiting Classical Heuristics via Automatic Translations of Tail-Recursive HTN Problems:
https://gki.informatik.uni-freiburg.de/papers/alford-etal-icaps16.pdf

2. Bercher, Pascal / Alford, Ron / Höller, Daniel (2019): A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations
https://www.ijcai.org/proceedings/2019/0875.pdf

3. Tansey, Wesley / Karpov, Igor: STRIPS planner implementation in python2:
https://github.com/tansey/strips

4. Bylander, Tom (1994): The Computational Complexity of Propositional STRIPS Planning
https://www.cs.utsa.edu/~bylander/pubs/artificial-intelligence94.ps.gz

5. The used string-to-list method:
https://stackoverflow.com/questions/7459501/how-to-convert-a-string-to-list-using-cli
13832673