Efficient HTN to STRIPS Encodings for Concurrent Planning

Nicolas Cavrel, Damien Pellier, Humbert Fiorino

Efficient HTN to STRIPS Encodings for Concurrent Planning

Nicolas Cavrel, Damien Pellier and Humbert Fiorino
Univ. Grenoble Alpes - LIG
Grenoble, France
{nicolas.cavrel, damien.pellier, humbert.fiorino}@univ-grenoble-alpes.fr

***Abstract*—The Hierarchical Task Network (HTN) formalism is used to express a wide variety of planning problems and many techniques have been proposed to solve them. A particular technique is to encode hierarchical planning problems as classical STRIPS planning problems. One advantage of this technique is to benefit directly from the constant improvements made by STRIPS planners. However, there are still few effective and expressive encodings. In this paper, we present the first HTN to STRIPS encodings allowing to generate *concurrent* plans. We show experimentally that these encodings not only outperform previous approaches on hierarchical IPC benchmarks but also produce more expressive solution plans.**

***Index Terms*—Automated Planning, Hierarchical Plannning, Concurrent Planning, STRIPS Planning**

I. Introduction

The Hierarchical Task Network (HTN) formalism [1], [2] is used to express a wide variety of planning problems in terms of decompositions of tasks into subtasks. HTN planning is used in many applications as, for instance, in task allocation for robot fleets [3], video games [4] or industrial contexts such as software deployment [5]. One possible explanation for this popularity is that HTN formalism usually fits better for real-world applications and domain experts' mindset: a HTN planning problem is expressed as a set of tasks to achieve rather than an objective state to reach, and the "processes" achieving these tasks as *methods*, that is to say task decompositions into "simpler" subtasks.

Many techniques are used to solve hierarchical planning problems. The first technique is to develop ad-hoc HTN planners, e.g., [6], [7]. The second technique consists in encoding HTN problems into other classical formalisms such as SAT problems, e.g., [8]–[10], constraint programming problems, e.g., [11] or simply as STRIPS planning problems that can be solved by classical STRIPS planners. This last technique has the advantage to benefit directly from the constant improvements of STRIPS planners. To our best knowledge, only three HTN to STRIPS encodings have been published so far [12]–[14]). These translations aims at solving HTN problems by producing a *sequential plan*, meaning that the resulting plan is a sequence of singular actions performed one at the time. However, in the general case and in particular in multi-agents planning problems, having a concurrent plan where several actions can be performed simultaneously or whose order of execution can be determined at the time of the plan execution is much more efficient and flexible, if not required.

In the literature, there are two approaches producing concurrent plans. The first one is to use *plan-space planning*. This approach refines an initial task network into a solution one by resolving the *flaws* of the initial task network, e.g, [15]. The resulting plan is a set of actions partially ordered by either causal relationships or precedence constraints inherited from the HTN decomposition. This approach produces very flexible plans, however it can quickly become intractable.

The second approach is to first produce a sequential plan and to *de-order* it into a partially ordered plan [16], [17]. This approach benefits from the efficiency of sequential planners and can then de-order the plan in polynomial time. The main disbenefit of this method is the quality of the resulting plan: while in a sequential context the optimal plan is the one containing the fewer number of actions, the optimal plan in a concurrent context could contain more actions but performed simultaneously. Hence, a sequential planner returning the sequentially optimal solutions first would not necessarily obtain the optimal concurrent solutions by de-ordering. The second issue with de-ordering approaches is that they are not directly applicable to HTN problems as they only de-order the plan while only maintaining the plan executability, but do not maintain the hierarchical constraints inherited from task decomposition.

To deal with this issue, we propose in this paper two contributions: (1) a search procedure called CPFD (Concurrent Partial Forward Decomposition) that produces and guarantees concurrently optimal plans, and (2) two HTN to STRIPS encodings of the CPFD procedure based on the translations proposed in [12], [14]. The translated problems can be solved by any classical sequential planner and produce concurrently optimal plans.

The rest of this paper is organized as follows. Section 1 defines the problem statement. Section 2 presents the Concurrent Partial Forward Decomposition procedure (CPFD). Section 3 introduces the key concepts used in the encodings, and Section 4 the two concrete STRIPS encodings of CPFD, called sCTHD (Static Concurrent Task Holders Decomposition encoding) and pCTHD (Push Concurrent Task Holders Decomposition). Finally, in the last section, we compare both CTHD and pCTHD with the translations proposed in [12], [14], which are the current state-of-the-art of HTN to STRIPS translations.

II. Problem statement

A. STRIPS Planning Problems

A *STRIPS planning problem* is a tuple $P = (L, A, I, G)$ where L is a finite set of logical propositions, A is a finite set of actions, $I \subseteq L$ is the initial state, and $G \subseteq L$ is the goal.

An *action* a is a tuple $a = (name(a), pre(a), add(a), del(a))$ where $name(a)$ is the name of the action, $pre(a)$ is the action's *preconditions*, $add(a)$ is its positive *effects* and $del(a)$ its negative ones, each one being a set of propositions. Two actions (a, b) are *independent* iff $del(a) \cap (pre(b) \cup add(b)) = \emptyset$ and $del(b) \cap (pre(a) \cup add(a)) = \emptyset$. Note that action independence only depends on action definitions. In the following, for all $a \in A$, $nInd(a)$ will denote the set of actions $b \in A$ *dependent* on a.

A *state* s is a set of logical propositions. The result of applying an action a to state s is a state s' defined by the transition function $s' = \gamma(s, a) = (s - del(a)) \cup add(a)$ if $pre(a) \subseteq s$, and undefined otherwise. Let an *action layer* π be a set of pairwise independent actions, and $pre(\pi) = \bigcup_{a \in \pi} pre(a)$. $add(\pi)$ and $del(\pi)$ are defined in the same way. By extension $s' = \gamma(s, \pi) = (s - del(\pi)) \cup add(\pi)$ if $pre(\pi) \subseteq s$, and undefined otherwise. Note that the actions of π can be executed concurrently or in any sequential permutation and still yield exactly the same state s'.

A *layered plan* Π is a sequence of action layers $\langle \pi_1, \ldots, \pi_n \rangle$. Let $\gamma(s, \Pi) = \gamma(\gamma(s, \pi_1), \langle \pi_2, \ldots, \pi_n \rangle)$. π_i *precedes* π_j if $i < j$. Likewise $a_i \prec a_j$ if $a_i \in \pi_i$, $a_j \in \pi_j$, and π_i *precedes* π_j. A layered plan Π is a solution to a STRIPS planning problem $P = (L, A, I, G)$ iff $G \subseteq \gamma(s, \Pi)$ (see Fig. 1).

B. HTN Planning Problems

We build on STRIPS planning problem definition to define a *HTN planning problem* as a tuple $P = (L, \mathcal{T}, A, M, I, tn)$ where L is a finite set of logical propositions, $\mathcal{T}$ is a finite set of *tasks*, A is a finite set of actions, M is a finite set of methods, $I \subseteq L$ is the initial state and tn the initial task network. There are two kinds of tasks: *primitive* tasks that can be resolved by a STRIPS action $a = (name(a), pre(a), add(a), del(a)) \in A$, and *compound* tasks, which can be recursively decomposed into either primitive or compound tasks by a method $m \in M$.

A *task network* is a tuple $tn = (T, \prec, \alpha)$ such that T is a finite set of task symbols, $\alpha : T \mapsto \mathcal{T}$ maps task symbols to tasks in $\mathcal{T}$, and $\prec$ is a partial order over T representing precedence constraints: t *precedes* t' if $t \prec t'$, or equivalently $(t, t') \in \prec$. $\prec$ is transitive. A task $\alpha(t), t \in T$ is *trailing* if $\forall t' \in T$, $(t', t) \notin \prec$ (t has no predecessor in T). $trail(tn)$ will denote the set of trailing tasks in T. Symmetrically, a task $\alpha(t)(t \in T)$ is a *last task* if $\forall t' \in T, (t, t') \notin \prec$ (t has no successor in T).

A *method* is a tuple $m = (task(m), pre(m), tn(m))$ where $task(m)$ is the compound task *decomposed by the method* m, $pre(m)$ is the method's *preconditions* and $tn(m)$ is a task network. A method m is a *resolver* of a compound task τ if $task(m) = \tau$. Note that a given compound task can have various methods to resolve it.

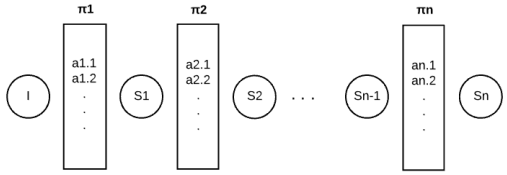

Fig. 1. A layered plan with the successive states resulting from the action layer application. Circles represent states, and rectangles action layers.

An action $a = (name(a), pre(a), add(a), del(a))$ can be applied to resolve a primitive task $\alpha(t)$ of the initial task network tn if t is trailing, $name(a) = \alpha(t)$ and $pre(a) \subseteq I$. The result is a new problem $P' = (L, \mathcal{T}, A, M, I', tn')$ where $I' = \gamma(I, a)$ and $tn' = (T \setminus \{t\}, \{(t', t'') \in \prec \mid t' \neq t\}, \alpha \setminus (t, \alpha(t)))$. In a symmetrical manner, a method $m = (task(m), pre(m), tn(m))$ can be applied to resolve a compound task $\alpha(t)$ of the task network tn if t is trailing, $task(m) = \alpha(t)$ and $pre(m) \subseteq I$. The result of applying the method m with $tn(m) = (T_m, \prec_m, \alpha_m)$ is a new problem $P' = (L, \mathcal{T}, A, M, I, tn')$ where $tn' = (T', \prec', \alpha')$ and:

$$\begin{aligned} T' &= (T \setminus \{t\}) \cup T_m \\ \prec' &= \{(t', t'') \in \prec \mid t' \neq t\} \cup \prec_m \cup \\ &\quad \{(t'', t') \in T_m \times T \mid (t, t') \in \prec\} \\ \alpha' &= \{(t', \alpha(t')) \mid t' \in T \setminus \{t\}\} \cup \alpha_m \end{aligned}$$

In other words, in $\prec'$ we keep all the precedence constraints of $\prec$ that do not involve t, add all the precedence constraints in $\prec_m$, and propagate precedence transitivity between $\prec$ and $\prec_m$ through t.

Applying either an action a or a method m to resolve a task in a planning problem P is called a *progression*. If $t_p \in \mathcal{T}$ is a primitive task of P, resolving t_p by a is a *progression* denoted $P \mapsto^a_{t_p} P'$. Similarly, if t_c is a compound task of P, decomposing t_p using m is a *progression* denoted $P \mapsto^m_{t_c} P'$.

To conclude, a layered plan $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$ is a solution for a HTN planning problem $P = (L, \mathcal{T}, A, M, I, tn)$ if (1) a sequence of progressions exists that transforms P into $P' = (L, \mathcal{T}, A, M, I', (\emptyset, \prec', \alpha'))$ (i.e. all the tasks of P have been resolved), and (2) $a_i \prec a_j \Leftrightarrow task(a_i) \prec' task(a_j)$ (i.e. action precedence constraints in the layered plan Π are equivalent to the primitive task precedence constraints in P'). In section 2, we show how CPFD builds a layered solution plan by applying progressions on P.

C. HTN to STRIPS Encoding Problems

The Hierarchical Task Network (HTN) formalism has been shown to be more expressive than STRIPS [1]. This means that any STRIPS problem can be formulated as a HTN problem but not the other way round. Therefore, the translation of a HTN problem into a STRIPS problem is not always possible. However, it has been proven by [12] that this translation is possible if the size of the solution task network can be

bounded. Given a HTN problem and a sequential solution plan, the minimum (respectively maximum) bound is the smallest (respectively largest) number of tasks in any task network visited by the sequence of progressions carried out to find this solution plan.

In practice, not all problems have a maximum bound, but all solvable problems have a minimum bound. These bounds are not directly related to the length of a problem solution, though the minimum progression bound is smaller than the optimal plan length[1].

Our encoding also assumes the bound existence. In addition and as [12] we make three other assumptions on the HTN problem to encode:

1) In the initial HTN problem, T is singleton. Otherwise, it is always possible to add a root dummy-task and a dummy-method to decompose it.
2) Every method of the HTN problem has a task network with a *single last task*. If a method does not have it, a dummy-task with no successor is added to the task network.
3) Methods have no preconditions. Otherwise, a trailing dummy action is added to the method task network with no effects and the method's preconditions.

These assumptions are made without loss of generality and will simplify the notations in the following.

III. Concurrent Partial-order Forward Decomposition

In this section, we introduce the procedure called CPFD (Concurrent Partial-order Forward Decomposition) (see Alg. 1) on which our encoding is based. The objective is to give on overview of our encoding and its properties. The encoding of this procedure as a STRIPS planning problem is presented in the next section.

CPFD is an adaptation of PFD (Partial-order Forward Decomposition) [18] procedure to output layered plans (see Figure 1). It takes as input a HTN problems and generates layered plans. Keep in mind that a layered plan is a solution of a HTN problem if actions resolve all the tasks of the task network, and if the ordering constraints of the actions in the layered plan satisfy the precedence constraints in this task network. Therefore, CPFD has to solve recursively the trailing tasks as in the PFD procedure. The difference lies on the resolution of the primitive tasks: while PFD adds actions to a sequential plan, CPFD adds them to layers of independent actions.

More precisely, CPFD takes as input four parameters: a HTN problem $P = (L, \mathcal{T}, A, M, I, (T, \prec, \alpha))$, a layered plan Π, the index i of the current layer π_i and τ the set of primitive tasks resolved by the actions in π_i. At the first call of CPFD, Π is an empty layered plan, $i = 0$ and $\tau = \emptyset$. At each recursive call, CPFD checks if the list of tasks T of the task network is empty, i.e., no more tasks have to be resolved. If this condition is satisfied, the layered plan Π is a solution to P and Π is returned. Otherwise, a task $t \in T$ is non deterministically selected among the trailing tasks (tasks without predecessors with respect to precedence constraints), and a resolver is non deterministically chosen. As in the PFD procedure, there are two ways to resolve t depending on whether t is primitive or compound.

- **Case 1. (Primitive task)** The resolvers of t are actions a whose preconditions are satisfied in the current state I and that are independent of all the actions already planned in the current layer π_i. Two cases are possible: either t has no resolvers and the current layer π_i is empty, meaning no action can solve t in the current state I, and CPFD returns FAILURE. Or t has a resolver but this resolver is not an independent action in the current layer: then CPFD moves to the next layer by applying to the current state all the actions already committed in the current layer. The idea is that t could be resolved by an action in a next state concurrently with other independent actions. Obviously, if t has an independent resolver a, a is added to the current layer π_i and t is added to the set of resolved primitive tasks τ.
- **Case 2. (Compound task)** CPFD computes all the methods resolving the compound task t, i.e., whose preconditions are satisfied in the current state I. If there is no method, then t cannot be solved, and CPFD returns FAILURE. Otherwise, CPFD non deterministically chooses a method m decomposing t, updates the task set and the ordering constraints accordingly.

CPFD is then called recursively until the tasks to solve in P are emptied ($T = \emptyset$, line 2 in Algo. 1) or a failure condition is met (line 9 and 24 in Algo. 1).

CPFD *is sound and complete*.

a) Proof sketch (Soundness): All produced plans come from a progression of the initial task network, thus there is a sequence of task decompositions that produces the primitive tasks in the solution plan. Furthermore, since a primitive task can be added to a layer if the corresponding node is trailing, all tasks preceding the one added have been planned on previous layers. Thus, the ordering constraints in $\prec$ are satisfied in the solution plan. Thus output plans are sound.

b) Proof sketch (Completeness): We will show that CPFD is complete based on the demonstration that PFD is complete. Let $P = (L, \mathcal{T}, A, M, I, tn)$ be a HTN problem and $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$ a layered solution plan of P. Let us show that there is a sequence of recursive calls of CPFD outputting Π. First, let us note that given a concurrent layer $\pi = \{a_1, \ldots, a_k\}$, any linearization of that layer $\langle a_{\gamma(1)}, \ldots, a_{\gamma(k)} \rangle$ where γ is a permutation function of $\{1, \ldots, k\}$, is a sequence of actions that can be applied to the same states as π. This is due to the *mutual independence* property of the actions within a concurrent layer. From there, any sequential plan produced by linearizing every layer of Π (by taking any permutation function on the layers) is a sound plan that also solves P. Let us consider the linearization Π_l defined by the n permutation functions $\gamma_1, \ldots, \gamma_n$. Since PFD is a complete algorithm, there

[1]For more details about the method to compute the bounds of the solution task network, see [12]

Algorithm 1: CPFD(P, Π, i, τ)

Let $P = (L, \mathcal{T}, A, M, I, (T, \prec, \alpha)$
Let π_i the i^{th} layer of Π
Let τ set of tasks already resolved at layer i
if $T = \emptyset$ **then return** Π
tasks $\leftarrow trail(tn) \setminus \tau$
nondeterministically choose $t \in$ tasks
if t *is primitive* **then**
  resolvers $\leftarrow \{a \in A \mid task(a) = t, pre(a) \subseteq I \text{ and } (\forall b \in \pi_i, a \text{ independent of } b)\}$
  if resolvers $= \emptyset$ **then**
    if $\pi_i = \emptyset$ **then return** Failure
    else
      $I \leftarrow \gamma(I, \pi_i)$ // Apply the layer effects
      $\Pi \leftarrow \Pi + []$ // Add a new empty layer
      $i \leftarrow i + 1$ // Update the layer index
      $T \leftarrow T \setminus \tau$ // Update the task network
      $\prec' = \{(t', t'') \in \prec \mid t'' \neq t\} \cup \prec_m \cup \{(t'', t') \in T_m \times T \mid (t, t') \in \prec\}$
      $\tau \leftarrow \emptyset$ // Reset the resolved tasks set
  else
    nondeterministically choose $a \in$ resolvers
    $\pi_i \leftarrow \pi_i \cup \{a\}$ // Add a to the current layer
    $\tau \leftarrow \tau \cup \{t\}$ // Add t to the resolved tasks
else
  resolvers $\leftarrow \{m \in M \mid task(m) = t\}$
  if resolvers $= \emptyset$ **then return** Failure
  nondeterministically choose $m \in resolvers$
  $\{m = (T_m, \prec_m)\}$
  $T \leftarrow (T \setminus \{t\}) \cup T_m$ // Decomposing tn with m
  $\prec \leftarrow \{(t', t'') \in \prec \mid t'' \neq t\} \cup \prec_m \cup \{(t'', t') \in T_m \times T \mid (t, t') \in \prec\}$
return CPFD(P, Π, i, τ)

is a sequence of recursions of PFD that outputs Π_l. At each recursion, PFD and CPFD either solve a trailing abstract task, or a trailing primitive task. While they solve abstract tasks the same way, PFD solves a primitive task by adding an action resolving it to the head of the plan, meanwhile CPFD adds the action resolving the task to the concurrent layer at the head of the plan. If the task cannot be resolved, PFD returns FAILURE while CPFD tries to add a new concurrent layer to the plan. Thus, for each recursive PFD call resolving an abstract task, the analogous call of CPFD is to solve the same abstract task. Each recursive call of PFD resolving a primitive task is analogous to a CPFD call adding the action to the current concurrent layer. However, CPFD requires extra recursive calls to switch layers.

IV. PLANNING THE PLANNING

In the following section, we present the basic concepts used to encode the CPFD procedure into STRIPS planning problems.

A. Taskholder and Current layer Encoding

The CPFD procedure resolves recursively trailing primitive tasks and compound tasks by modifying the initial task network of the problem until the task network contains an empty set of tasks. To encode this procedure, we need first to model a task network with STRIPS. To achieve this, we use the concept of *taskholder* introduced first by [12]. A taskholder is a STRIPS object that will act as a container for a task. By way of extension, a task network is modeled by a stack of taskholders and by fluent propositions modelling the ordering constraints between the tasks contained in the taskholders. The number of taskholders has to be fixed before translating the HTN problem. Similarly to HTN2STRIPS, our encodings require at least as many taskholders as there are tasks in the largest explored task network, thus the number of taskholders in our encodings can be estimated as described in HTN2STRIPS [12].

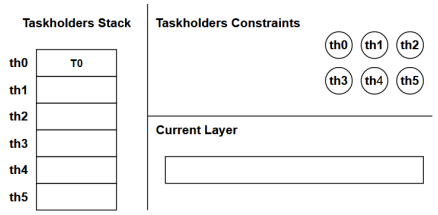

Fig. 2. The problem is initialized by setting the initial task into the first taskholder.

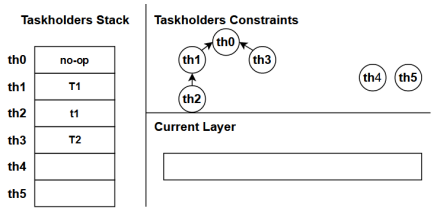

Fig. 3. *T0* is decomposed into four tasks, two compound ones *T1* and *T2*, and two primitive ones *t1* and *no-op*. Since neither *T1*, *T2* or *t1* is a last task, a *no-op* action is inserted instead of *T0*. The constraint over the taskholders are represented on the top right graph: each directed edge represents a precedence constraint. So for instance, the task in $th2$ should be planned before the one in $th1$.

In addition, we model the current layer from CPFD by a set of fluent propositions. Each proposition denotes whether or not an action $a \in A$ is planned in the current layer. The main idea is to allow an action to be added to the current layer only if every action contained in the current layer is independent with the newly added one.

A graphical representation of the initial structure of a problem, i.e., taskholders and current layer, is given in Figure 2.

B. Encoding CPFD as STRIPS Actions

To encode the dynamic of the CPFD procedure (Alg . 1) we need to define three types of STRIPS actions: (1) the first type of actions resolves a trailing compound task and update the current task network according to a method decomposition, (2) the second type of actions resolves a primitive task and add an action to the current layer, and (3) the last type of actions is the one switching layers, making the algorithm build the next layer of the solution plan. The planning process will choose one action among the three types. Applying one of

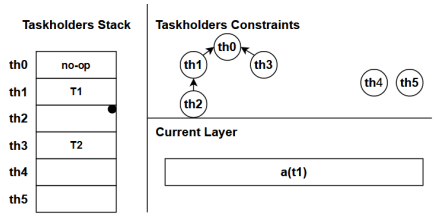

Fig. 4. $th2$ is unconstrained, and contains a primitive task. It is added to the plan step by removing the task from the taskholder, adding the action $a(t1)$ resolving $t1$ to the plan step, and marking the taskholder as resolved (represented by the black dot).

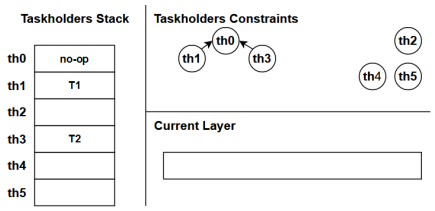

Fig. 5. The plan step is terminated by emptying it, the constraints implied by the resolved taskholders are removed.

these actions is equivalent to one recursive call of the CPFD procedure:

1) **Actions for resolving compound tasks:** These actions reflect the decomposition of a compound task into subtasks according to a method. It corresponds to the lines 27 to 30 of the CPFD procedure. In order to apply these actions, the taskholder must be unconstrained and there must be enough empty taskholders in the stack. An example of taskholder usage is represented on Figure 3. The substasks of the method M_1 decomposing T_0 are added to the stack of taskholders and the task T_0 is replaced by the last task of the current task network. As M_1 has no *last task*, the no-op action is used. Finally, the *ordering* constraints are set over the taskholders.
2) **Actions for resolving primitive tasks:** In CPFD, trailing primitive tasks are resolved by adding their corresponding action into the current layer (lines 21 to 23 of Alg. 1). In our encodings, we define corresponding actions that take as parameters an unconstrained taskholder containing a primitive task which add its corresponding action to the current layer if no concurrent action is already in it. An example of application is displayed on Figure 4. The taskholder $th2$ is unconstrained and contains a primitive task $t1$ that can be resolved by the action $a(t1)$. The task is resolved by adding $a(t1)$ to the current layer, $th2$ is emptied and marked as resolved. As in Alg. 1, the constraints implied by the resolved taskholder are not removed yet. They will be removed when moving to the next layer.
3) **Action for switching of plan layer:** This action corresponds to switching to the next layer. This action empties the *current layer*, setting up the next one in the solution plan. It also removes the constraints implying the resolved taskholders. This action can be applied with no precondition and works as displayed in Figure 5. In this example, only $th2$ is marked as resolved. After the application of the action, the constraints implying $th2$ are removed, so the constraint between $th2$ and $th1$ is deleted, additionally $th2$ is unmarked as resolved and set as empty. Then the current layer is emptied by removing all actions in it.

V. Concurrent TaskHolder Decomposition Encodings

In this section, we present two encodings of the CPFD procedure based on the previous introduced concepts, called respectively Static Concurrent TaskHolder Decomposition (sCTHD) and Push Concurrent TaskHolder Decomposition (pCTHD). The main difference of these encodings relies in the *taskholders usage*. The first encoding called sCTHD constrains the planner to use the taskholders in an increasing order according to the static relationship defining the taskholders stack. The second encoding called pCTHD (Push CTHD) always sets the newly added tasks into the *last* taskholders and uses the concept of *Push* action defined in [14]. Similarly to HTN2STRIPS, both proposed encodings depend on an integer parameter, the *progression bound* denoted b in the rest of the paper representing the number of taskholders.

In the remainder of this section, we will first present the predicates used to model the STRIPS problem that are common to both encodings. Secondly, we will present the encoding of the initial state and the goal, which are also common to both encodings. Finally, we will present the actions that encode the CPFD procedure that are specific to the encodings.

A. Predicates Definition

Our two encodings, sCTHD and pCTHD, model the CPFD procedure. In that regard, they share the predicates related to the taskholders stack, to the ordering constraints and to the current layer modelling. They also have specific predicates that are related to their taskholders management. In the following, every predicate is used by both sCTHD and pCTHD unless specified otherwise.

- $(not\_constraint\ ?th1\ ?th2 - taskholder)$ represents the fluent constraints over the taskholders. The predicate is inverted for convenience, so when the proposition $(not\_constraint\ th1\ th2)$ is false, it means that the task contained in $th1$ must be planned before the task contained in $th2$.
- $(empty\ ?th - taskholder)$ represents the fact that a taskholder is empty. The proposition $(empty\ th)$ is true if the taskholder th does not contain a task.
- $(in\ ?t - task\ ?th - taskholder)$ is the predicate representing whether or not the task $?t \in \mathcal{T}$ is set in the taskholder. The proposition $(in\ t\ th)$ is true if t is set in th.
- $(not\_planned\ ?a - action)$ is true if a is not planned in the current plan step.

- $(resolved\ ?th - taskholder)$ is a predicate representing whether or not a taskholder has been resolved. So the proposition $(resolved\ th)$ is true if the taskholder th has been resolved.
- $(prec\_th\ ?th1\ ?th2 - taskholder)$ is a predicate defining the static relationship between the taskholders. The proposition $(prec\_th\ th1\ th2)$ is true if $th1$ is preceding $th2$ in the stack. In the following, this order will be fixed, and for all $0 \leq i, j < b$ the proposition $(prec\_th\ th_i\ th_j)$ will be true if and only if $i \leq j$. This predicate is specific to sCTHD.
- $(next\ ?th\ -\ taskholder)$ is a predicate used to denote the taskholder prioritized for the next *push* action. This predicate is specific to pCTHD.

B. Initial and Goal State Encoding

The initial state of the translated problems is defined by setting the initial task into the first taskholder. The remaining taskholders are set as empty and ordered in a stack. As there is no constraint over the taskholder yet, all constraint predicates are initialized accordingly.

$I' = I \wedge (in\ task_0\ th_0) \wedge \bigwedge_{1 \leq i < b}(empty\ th_i)$
$\bigwedge_{0 \leq i,j < b}(not\_constraint\ th_i\ th_j) \wedge$
$\bigwedge_{1 \leq i < b}(prec_t h\ th_{i-1}\ th_i) \wedge$
$\bigwedge_{a \in A}(not\_planned\ a)$

For the goal state, the problem is solved when all taskholders are empty, meaning that all tasks are planned, and when the goal state is reached. Then we have:

$G' = G \wedge \bigwedge_{i=0}^{p-1}(empty\ th_i)$

C. sCTHD Actions Encoding

sCTHD generates the set of actions $A' = A_c \cup A_p \cup A_l$ where A_c is the set of actions resolving compound tasks, A_p the set of actions resolving primitive tasks and A_l the set of actions switching plan layers:

- **Actions resolving compound tasks:**
  Let $m = (task(m), pre(m), tn(m))$ and $(task_1, task_2, \ldots, task_k)$ the subtasks in tn. We assume that $task_k$ is the *last task* of tn. For all methods $m \in M$ there is an action $a_m \in A_c$ with k parameters $(?th_1, ?th_2, \ldots, ?th_k)$ defined as follows:
  - $pre(a_m) = (in\ task(m)\ ?th_1) \wedge$
    $\bigwedge_{i=0}^{b-1}(not\_constraint\ th_i\ ?th_1) \wedge$
    $\bigwedge_{i=2}^{k}(prec\_th\ ?th_i\ ?th_{i+1}) \wedge$
    $\bigwedge_{i=2}^{k}(empty\ ?th_i)$
  - $add(a_m) = \bigwedge_{i=2}^{k}(in\ task_i\ ?th_i) \wedge (in\ task_k\ ?th_1)$
  - $del(a_m) = \bigwedge_{i=2}^{k}(empty\ ?th_i) \wedge$
    $\bigwedge_{task_i \prec task_j}(not\_constraint\ ?th_i\ ?th_j) \wedge$
    $\bigwedge_{i=2}^{k}(not\_constraint\ ?th_i\ ?th_0)$

  Note that the $k-1$ last taskholder parameters are required to be *ordered* according to the static relationship defined by the $prec\_th$ predicate. For instance, if three new taskholders are required to decompose the task in $th6$, $(th6\ th2\ th4\ th7)$ is a valid combination of parameters, while $(th6\ th2\ th5\ th3)$ is not.
- **Actions resolving primitive tasks:**
  For all actions $a \in A$ there is an action $a_p \in A_p$ with one parameter: a taskholder containing p and denoted $?th$. The action is defined as follows:
  - $pre(a_p) = pre(a) \wedge (in\ task(a)\ ?th) \wedge$
    $\bigwedge_{i=0}^{b-1}(not\_constraint\ th_i\ ?th) \wedge$
    $\bigwedge_{t \in nInd(p)}(not\_planned\ ?a)$
  - $add(a_p) = add(a) \wedge (empty\ ?th) \wedge (resolved\ ?th)$
  - $del(a_p) = del(a) \wedge (not\_planned\ ?a)$
- **Action switching layers:**
  A_l is composed of one conditional action a_l with no parameter. This action has a non conditional part: emptying the current layer, and a conditional part: unconstraining and making the resolved taskholders available for a new use. It is defined as follows:
  - $pre(a_l) = \emptyset$
  - $add(a_l) = \bigwedge_{a \in A}(not\_planned\ ?a) \wedge$
    $\forall(?th\ -\ taskholder)$ when $(resolved\ ?th)$,
    $\bigwedge_{i=0}^{b-1}(not\_constraint\ ?th\ \ th_i)$
  - $del(a_l) = \emptyset$

D. pCTHD Actions Encoding

The pCTHD encoding generates the set of action $A' = A_c \cup A_p \cup A_l \cup A_{Push}$. pCTHD encodes the CPFD procedure the same way as CTHD does, in that regard the sets of actions encoding the *primitive tasks* resolving A_p and action for the *layer switch* A_l are defined the same way as in CTHD. pCTHD manages the taskholders as the Push version of HTN2SAS does [14]. Then we have:

- **Actions resolving compound tasks:**
  Let $m = (task(m), pre(m), tn(m))$ and $(task_1, task_2, \ldots, task_k)$ the subtasks in tn, $task_k$ denotes the *last task* of tn. For every method $m \in M$ there is an action $a_m \in A_c$ with one parameter $(?th - taskholder)$ defined as follows:
  - $pre(a_m) = (in\ task(m)\ ?th) \wedge$
    $\bigwedge_{i=1}^{b}(not\_constraint\ th_i\ ?th_1) \wedge$
    $\bigwedge_{i=1}^{k-1}(empty\ th_{b-i+1})$
  - $add(a_m) = \bigwedge_{i=1}^{k-1}(in\ \ task_i\ \ th_{b-i+1}) \wedge$
    $(in\ task_k\ ?th)$
  - $del(a_m) = \bigwedge_{i=1}^{k-1}(empty\ th_{b-i+1}) \wedge$
    $\bigwedge_{task_i \prec task_j}(not\_constraint\ ?th_{b-i+1}\ th_{b-j+1}) \wedge$
    $\bigwedge_{i=1}^{k-1}(not\_constraint\ th_{b-i+1}\ ?th)$

  In pCTHD, the subtasks inherited from a method decomposition are always set in the same taskholders (the last ones). This allows the method actions to only have one parameter, thus reducing the number of operators generated.
- **Push Actions:**
  For every task $t \in \mathcal{T}$, and for every taskholder index $2 \leq k \leq b$ there is a push action $p_t^k \in A_p$ defined as follows:
  - $pre(p_t^k) = (in\ t\ th_k) \wedge$
    $(empty\ th_{k-1}) \wedge$
    $\bigwedge_{i=1, i \neq k}^{b} \neg(next\ th_i)$

- $add(p_t^k) = (in\ t\ th_{k-1}) \wedge (empty\ th_k)$
  $\bigwedge_{i=1}^{b}(not_precedes\ th_k\ th_i)$
  when $(empty\ th_{k-2}), (next\ th_{k-1})$
- $del(p_t^k) = (next\ th_k) \wedge (empty\ th_{k-1}) \wedge$
  $(in\ t\ th_k) \wedge$
  $\forall(?th\ -\ taskholder):$
  when $\neg(not_precedes\ ?th\ th_k)$,
  $\bigwedge_{i=0}^{b-1} \neg(not\_precedes\ ?th\ \ th_{k-1})$
  when $\neg(not_precedes\ th_k\ ?th)$,
  $\bigwedge_{i=0}^{b-1} \neg(not\_precedes\ th_{k-1}\ \ ?th)$

A *Push action* moves a task and its constraints from a taskholder to the preceding one in the stack. Note that the push actions rely on conditional effects, allowing for a much more compact encoding.

VI. Experimentations and Results

In this section we present the experiments and results obtained to evaluate sCTHD and pCTHD efficiency. We compare our encodings with the two other known encodings: HTN2STRIPS and HTN2SAS encodings [12], [14]. Note that these encodings are not able to generate concurrent plans unlike our encodings. Our aim is to show how a concurrent procedure can perform against non concurrent ones, both on problems with totally ordered solution plans and problems with concurrent plans. Hence we selected 7 domains from the IPC partially ordered benchmarks that can have concurrent solution plans, and, in addition, a domain that is susceptible to concurrent actions, Miconic. We also added concurrent version of sequential domains where additional agents can perform the tasks simultaneously ($nameConc$ represents the concurrent version of the domain $name$).

A. Experimental Setup

We ran all experiments on a single core of a Intel Core i7-9850H CPU using the Fast Downward library [19] with enforced hill climbing and then lazy greedy search with Fast Forward heuristics [20]. All experiments were set a limit of 8GB of RAM over 600 seconds. We compared all four encodings according to two criteria:

1) **Solving time:** We measured the time spent by Fast Downard to find a solution to the translated problems. We perform an iterated search over the *progression bound* until a solution is found.
2) **Plan quality:** As we want to demonstrate the efficiency of our approach at producing concurrent plans, we measure the *makespan* of the solution output by the solving of the translated problems.

We compared these encodings by scoring them relatively to each other. The score of an encoding k over the domain d with a set of instances P_d is defined as:

$$s_d(k) = \frac{1}{|P_d|} \sum_{i \in P_d} \frac{min_{e \in encodings}(score(e,i))}{score(k,i)}$$

where $score(k,i)$ is the score measured of the encoding k on the instance i. The results are displayed on Tables I and II.

TABLE I
Solving Time

	HTN2STRIPS	HTN2SAS	sCTHD	pCTHD
Satellite	0,20	0,83	**0,95**	0,71
SatelliteConc	0,21	0,79	**0,93**	0,6
Rover	0,02	**1**	0,26	0,85
Transport	0,19	0,58	**0,82**	0,45
Blocksworld	0,29	**0,77**	0,62	0,65
BlowksworldConc	0,34	**0,94**	0,69	**0,94**
Miconic	0,72	**0,84**	0,44	0,82
MiconicConc	0,24	**1**	0,45	0,43
Total	1,56	**6,74**	5,28	5,56

TABLE II
Solution Makespan

	HTN2STRIPS	HTN2SAS	sCTHD	pCTHD
Satellite	**1**	**1**	**1**	**1**
SatelliteConc	0,94	0,94	**1**	0,95
Rover	0,97	0,97	**1**	0,98
Transport	0,72	0,91	**1**	0,91
Blocksworld	**1**	**1**	**1**	**1**
BlowksworldConc	0,82	0,84	**1**	0,90
Miconic	**1**	**1**	**1**	**1**
MiconicConc	0,76	0,76	**1**	0,91
Total	7,21	7,42	**8**	7,65

B. Solving Time

Table I represents the results of our experimentations with regards to the *solving time*. On the *Satellite* and *Transport* domains sCTHD is the most efficient. On the *Rover* and *Blocksworld* domains, HTN2SAS Push is dominating the other encodings. This discrepancy displays the main difference between the taskholder management and the *Push* management. As Push encodings decompose subtasks in the *last* taskholders, these encodings perform worst on *tail-recursive* problems such as Satellite and Transport. On the other hand, sCTHD does not suffer from this and has the same performances whatever the problem structure. However, Push encodings are very efficient on *head-recursive* problems or problems with low recursivity as the conditionnal Push actions are rarely needed, hence representing the problem in a very compact and almost non conditional way. This efficiency is apparent on the Rover, Blocksworld and Miconic domains.

C. Plan Quality

Table II represents the results concerning the makespan of the solution plans. As we can see it on this table, our encodings obtain the best results on every domain, followed by the pCTHD encoding on every domain too. Obviously this is due to the fact that these encodings are able to produce concurrent plans, and thus to reduce the makespan of the solution. Note however that sCTHD and pCTHD can increase the concurrency of the solution plan by increasing the number of taskholders. This is not apparent on our experimentations as the tests end at the first valid solution (hence at the lowest valid number of taskholder). The effect of increasing the number of taskholders in our encodings over the resulting makespan is displayed in Figure 6 on a typical problem from Satellite. As displayed on the figure, sCTHD cannot find a solution until it reaches the minimal taskholder number to

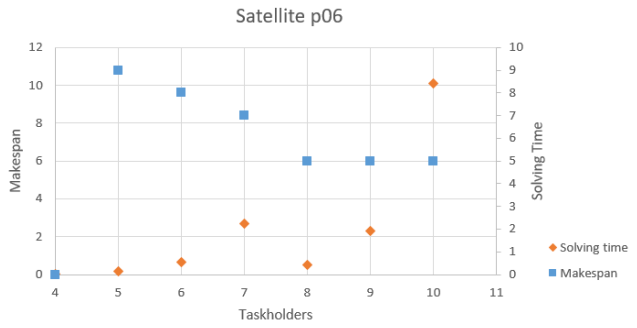

Fig. 6. The makespan of the solution plan returned by sCTHD with regards to the number of parameters on the instance 6 from Satellite p06.

form a solution. Then the makespan of the solution plan decreases as the number of taskholders increases. Indeed, the more taskholders are available, the more concurrent tasks can live simultaneously in the progression network, and the more possibilities are offered to the STRIPS planner. However, increasing the number of taskholders has a negative effect on the solving time. The more taskholders, the more complex is the translated problem. Note that increasing the number of taskholders cannot have a negative impact on plan quality: the more taskholders there are, the more reachable through progression the task networks are, hence the more concurrent actions added simultaneously to the current layer there are.

Finally, on some instances, the maximum size of progression can be bounded. Using this bound as progression bound will lead sCTHD and pCTHD to explore and output the solution plans with the smallest makespan as every progression network can be represented in the taskholder stack. An algorithmic method to estimate this bound has been proposed in [12].

D. Discussion on CTHD optimality

Two factors affect the optimality of the CTHD encoding: the first is the number of taskholders used in the encoding and the second is the cost associated to each translated action. Having more taskholders allows for more primitive tasks to live simultaneously within the taskholder stack. Thus there are more opportunities for concurrent tasks to be resolved within the same layer. The makespan-optimal solution can be obtained only if enough taskholders have been encoded. Figure 6 shows the effect of increasing the number of taskholder on the makespan of the solution. The cost of the translated actions allows to guide the search procedure towards a better quality solution: finding the makespan-optimal solution means finding the plan with the smallest amount of *layers* within it. By increasing the cost of the actions switching layers, the search procedure is guided towards the better plans.

VII. Conclusion and Future Works

In this article, we have presented a new HTN procedure, CPFD, solving HTN problems with layered solution plans. We have proposed two HTN to STRIPS encodings of this procedure that can be solved by any STRIPS planner. Then we have showed experimentally that our translations perform well comparatively to the state of the art approaches, and more importantly produce concurrent solution plans. However, our representation of the concurrency still remains limited. There are stronger ways to represent concurrency by returning a partial order of the planned actions. This concurrency allows to have solution plans that are more resilient and flexible, and thus are more applicable in real world applications. This kind of partially-ordered plans will be addressed in our future works.

References

[1] K. Erol, J. Hendler, and D. Nau, 'Complexity Results for HTN Planning', Annals of Mathematics and Artificial Intelligence, vol. 18, pp. 69–93, 04 2003.
[2] P. Bercher, R. Alford, and D. Höller, 'A Survey on Hierarchical Planning - One Abstract Idea, Many Concrete Realizations', in IJCAI, 2019, pp. 6267–6275.
[3] A. Milot, E. Chauveau, S. Lacroix, and C. Lesire, 'Solving Hierarchical Auctions with HTN Planning', in ICAPS workshop on Hierarchical Planning, 2021.
[4] A. Menif, E. Jacopin, and T. Cazenave, 'SHPE: HTN Planning for Video Games', in Workshop on Computer Games, 2014, pp. 119–132.
[5] I. Georgievski, F. Nizamic, A. Lazovik, and M. Aiello, 'Cloud Ready Applications Composed via HTN Planning', in IEEE Conference on Service-Oriented Computing and Applications, 2017, pp. 81–89.
[6] P. Bercher, S. Keen, and S. Biundo, 'Hybrid Planning Heuristics Based on Task Decomposition Graphs', in SoCS, 2014, pp. 35–43.
[7] D. Nau et al., 'SHOP2: An HTN planning system', J. Artif. Intell. Res., vol. 20, pp. 379–404, 2003.
[8] D. Schreiber, D. Pellier, H. Fiorino, and T. Balyo, 'Tree-REX: SAT-Based Tree Exploration for Efficient and High-Quality HTN Planning', in ICAPS, 2019, 2019, pp. 382–390.
[9] R. Huang, Y. Chen, and W. Zhang, 'SAS+ Planning as Satisfiability', J. Artif. Intell. Res., vol. 43, pp. 293–328, Mar. 2012.
[10] D. Schreiber, 'Lilotane: A Lifted SAT-based Approach to Hierarchical Planning', J. Artif. Intell. Res., vol. 70, pp. 1117–1181, 2021.
[11] V. Vidal and H. Geffner, 'Branching and pruning: An optimal temporal POCL planner based on constraint programming', Artificial Intelligence, vol. 170, no. 3, pp. 298–335, 2006
[12] R. Alford, G. Behnke, D. Höller, P. Bercher, S. Biundo, and D. W. Aha, 'Bound to Plan: Exploiting Classical Heuristics via Automatic Translations of Tail-Recursive HTN Problems', in ICAPS, 2016, pp. 20–28.
[13] R. Alford, U. Kuter, and D. Nau, 'Translating HTNs to PDDL: A Small Amount of Domain Knowledge Can Go a Long Way', in IJCAI, 2009, pp. 1629–1634.
[14] G. Behnke, F. Pollitt, D. Höller, P. Bercher, and R. Alford, 'Making Translations to Classical Planning Competitive With Other HTN Planners', in AAAI, 2022, pp. 11744–11754.
[15] P. Bercher, S. Keen, and S. Biundo, 'Hybrid Planning Heuristics Based on Task Decomposition Graphs', in Proceedings of the Annual Symposium on Combinatorial Search, 2014.
[16] M. Waters, B. Nebel, L. Padgham, and S. Sardina, 'Plan Relaxation via Action Debinding and Deordering', Proceedings of the International Conference on Automated Planning and Scheduling, vol. 28, no. 1, pp. 278–287, Jun. 2018.
[17] C. Bäckström, 'Computational Aspects of Reordering Plans', J. Artif. Int. Res., vol. 9, no. 1, pp. 99–137, Sep. 1998.
[18] M. Ghallab, D. Nau, and P. Traverso, Automated Planning: Theory and Practice. 2004.
[19] M. Helmert, 'The Fast Downward Planning System', J. Artif. Intell. Res., vol. 26, pp. 191–246, 2006.
[20] J. Hoffmann and B. Nebel, 'The FF Planning System: Fast Plan Generation Through Heuristic Search', CoRR, vol. abs/1106.0675, 2011.
[21] M. Katz, S. Sohrabi, H. Samulowitz, and S. Sievers, 'Delfi: Online planner selection for cost-optimal planning', in International Planning Competition, 2018, pp. 55–62.