On the Efficient Inference of Preconditions and Effects of Compound Tasks in Partially Ordered HTN Planning Domains

Conny Olz[1] **and Pascal Bercher**[2]
[1] Ulm University
[2] The Australian National University
conny.olz@uni-ulm.de, pascal.bercher@anu.edu.au

Abstract

Recently, preconditions and effects of compound tasks based on their possible refinements have been introduced together with an efficient inference procedure to compute a subset of them. However, they were restricted to total-order HTN planning domains. In this paper we generalize the definitions and algorithm to the scenario of partially ordered domains.

Introduction

In Hierarchical Task Network (HTN) planning we refine an initial abstract task step-by-step into a more fine-grained description until an executable sequence of actions results (Erol, Hendler, and Nau 1996; Ghallab, Nau, and Traverso 2004; Bercher, Alford, and Höller 2019).

Compound tasks together with decomposition methods govern the refinement process. In many HTN formalizations one does not model or specify concrete preconditions or effects for compound tasks like for primitive ones. Instead, they are only given implicitly via the actions deeper down in the hierarchy. Recently, Olz, Biundo, and Bercher (2021) defined preconditions and effects of compound tasks that can be inferred based on the decomposition structure. Besides analyzing the computational complexity they also introduced a procedure to compute a subset of these in polynomial time but they were restricted to totally ordered (t.o.) HTN planning domains. We extend their work by generalizing the definitions and algorithm to also work with partially ordered (p.o.) domains.

As pointed out by Olz, Biundo, and Bercher (2021) the potential applications of such inferred preconditions and effects are manifold. In the context of modeling assistance they might reveal unintended modeling effects or errors and a study indicated that they can help to better comprehend a given domain model (Olz et al. 2021). More prominently, resemblances of the preconditions and effects considered by us were already exploited to speed up several kinds of planning systems (Tsuneto, Hendler, and Nau 1998; Nau et al. 2003; Clement, Durfee, and Barrett 2007; Waisbrot, Kuter, and Könik 2008; Goldman and Kuter 2019; Schreiber, Pellier, and Fiorino 2019; Magnaguagno, Meneguzzi, and de Silva 2021; Schreiber 2021). By extending the inference to p.o. domains we make them also available for planning systems solving p.o. problems like SHOP2 (Nau et al. 2003), FAPE (Dvořák et al. 2014), PANDA$_3$-POCL (Bercher et al. 2017), PANDA$_\pi$-SAT (Behnke, Höller, and Biundo 2019), PANDA$_\pi$-pro (Höller et al. 2020), SIADEX (Fernandez-Olivares, Vellido, and Castillo 2021), pyHiPOP (Lesire and Albore 2021), and PDDL4J (Pellier and Fiorino 2021). Their exploitation in p.o. systems should however be done with care as discussed later in the paper.

One further utilization especially for the p.o. case that we would like to bring up is that inferred preconditions and effects bear useful information for turning a p.o. domain or problem into t.o. while preserving specific properties. Planners can then make use of the special case to solve such problems more efficiently.

For an overview of related work concerning the inference of preconditions and effects we would like to refer to Olz, Biundo, and Bercher and add the work by Magnaguagno, Meneguzzi, and de Silva (2021) that has been published in the meantime. Their lifted planner HyperTensioN infers preconditions of compound tasks similarly to Olz, Biundo, and Bercher but is also restricted to t.o. domains.

HTN Planning Formalism

Our formalism is based on the one by (Bercher, Alford, and Höller 2019). A partially ordered (p.o.) HTN planning domain is a tuple $\mathcal{D} = (F, A, C, M)$, where F is a finite set of facts, A are *primitive tasks*, and C the set of *compound tasks*. Primitive tasks $a = (prec, add, del) \in A$ – also called actions – are described by their *preconditions* $prec(a) \subseteq F$ and their add and delete *effects* $add(a), del(a) \subseteq F$, resp. As in STRIPS planning, an action $a \in A$ is *applicable* in a state $s \in 2^F$ if $prec(a) \subseteq s$. The application of it in s (if applicable) produces the successor state $\delta(s, a) = (s \setminus del(a)) \cup add(a)$. Accordingly, the application of a sequences of actions $\bar{a} = \langle a_0 \dots a_n \rangle$ with $a_i \in A$ is possible in a state s_0 if a_0 is applicable in s_0 and for all $1 \leq i \leq n$, a_i is applicable in $s_i = \delta(s_{i-1}, a_{i-1})$. The second type of tasks are compound tasks, which serve as a collection of primitive and/or compound tasks organized in *task networks*. A task network is a triple $tn = (T, \prec, \alpha)$, where T is a (possibly empty) set of identifiers (ids) that are mapped to tasks by a function $\alpha : T \to A \cup C$. Therefore, a single task can be contained in a task network more than once. A set of ordering constraints $\prec : T \times T$ defines a partial order on the identifiers. *Decomposition methods* M specify how exactly compound tasks were decomposed. A method $m \in M$ is a pair (c, tn) of a compound task $c \in C$ and a task network. It decomposes a task network $tn_1 = (T_1, \prec_1, \alpha_1)$ into a task network $tn_2 = (T_2, \prec_2, \alpha_2)$ if $t \in T_1$ with $\alpha_1(t) = c$ and there is a task network $tn' = (T', \prec', \alpha')$ equal to tn but using different ids, so $T_1 \cap T' = \emptyset$. The task network tn_2 is defined as

$$\begin{aligned} tn_2 &= ((T_1 \setminus \{t\}) \cup T', \prec' \cup \prec_D, (\alpha_1 \setminus \{t \mapsto c\}) \cup \alpha') \\ \prec_D &= \{(t_1, t_2) \mid (t_1, t) \in \prec_1, t_2 \in T'\} \cup \\ &\quad \{(t_1, t_2) \mid (t, t_2) \in \prec_1, t_1 \in T'\} \cup \\ &\quad \{(t_1, t_2) \mid (t_1, t_2) \in \prec_1, t_1 \neq t \wedge t_2 \neq t\} \end{aligned}$$

So if a compound task c is decomposed, it is removed from the task network and the tasks of the chosen method's subnetwork were added together with ordering constraints that hold for c. When a task network tn can be decomposed into a task network tn' by applying the method m to a task with the identifier t, we write $tn \rightarrow_{t,m} tn'$; if it is possible using several methods in sequence, we write $tn \rightarrow tn'$.

An HTN planning problem $\Pi = (\mathcal{D}, s_I, tn_I, g)$ contains additionally an initial state $s_I \in 2^F$, an initial task network tn_I, and a goal description $g \subseteq F$. A solution to a Π is a task network $tn = (T, \prec, \alpha)$ if and only if

1. it can be reached via decomposing tn_I (i.e. $tn_I \rightarrow tn$),
2. all task are primitive ($\forall t \in T : \alpha(t) \in A$), and
3. there exists a sequence $\langle t_1 t_2 \dots t_n \rangle$ of the task identifiers in T that is in line with $\prec$, and the application of $\langle \alpha(t_1)\alpha(t_2) \dots \alpha(t_n) \rangle$ in s_0 results in a goal state $s \supseteq g$.

To ease notation we additionally define the following: A task network containing only one task $c \in A \cup C$, i.e., $(\{t\}, \emptyset, \alpha(t) = c)$, is denoted $\langle c \rangle$. A *t.o. refinement* of some compound task $c \in C$ is a sequence of primitive tasks $\bar{a} = \langle a_1 \dots a_n \rangle$ if and only if there exists a task network $tn = (T, \prec, \alpha)$, $\langle c \rangle \rightarrow tn$ and there exists a sequence $\langle t_1 \dots t_n \rangle$ of the ids in T that is in line with $\prec$ so that $\langle \alpha(t_1) = a_1 \dots \alpha(t_n) = a_n \rangle$. Lastly, by $c \in tn$ for some task $c \in A \cup C$ and task network $tn = (T, \prec, \alpha)$ we abbreviate that there exists a task identifier $t \in T$ so that $\alpha(t) = c$.

Preconditions and Effects of Compound Tasks

The definitions of preconditions and effects of compound tasks for totally ordered task networks by Olz, Biundo, and Bercher (2021) were based on sets of states that enable the execution of such tasks and the states in which an application of a refinement can result. We adapt these two definitions to p.o. domains in the following. Therefore, consider a domain $\mathcal{D} = (F, A, C, M)$ and a compound task $c \in C$.

The set of *executability-enabling states* of c is

$$E(c) = \{s \in 2^F \mid \exists \text{ t.o. refinement of } c \text{ applicable in } s\}.$$

The set of *resulting states* of c w.r.t. some state $s \in 2^F$ is

$$R_s(c) = \{s' \in 2^F \mid \exists \text{ t.o. refinement appl. in } s \text{ res. in } s'\}.$$

Based on these two updated definitions for p.o. domains, the definitions of preconditions and effects of compound tasks can be used without further adaptions. Olz, Biundo, and Bercher (2021) defined several types, e.g., depending on whether they are dependent or independent of a state and they differentiate between effects and postconditions. We repeat only those that are relevant for this paper.

State-independent positive and negative effects (cf. their Def. 4) of a compound task c are facts that are added or deleted, resp., by the successful execution of a refinement of c, independent of the state in which the task is executed, i.e.,

$$\begin{aligned} \mathit{eff}^+_*(c) &:= (\bigcap_{s \in E(c)} \bigcap_{s' \in R_s(c)} s') \setminus \bigcap_{s \in E(c)} s \\ \mathit{eff}^-_*(c) &:= \bigcap_{s \in E(c)} (F \setminus \bigcup_{s' \in R_s(c)} s') \end{aligned}$$

if $E(c) \neq \emptyset$, otherwise $\mathit{eff}^{+/-}_*(c) := \mathit{undef}$.

Possible state-independent effects (cf. Def. 5) of a compound task c are not guaranteed to hold (or not hold, resp.,) after *every* refinement of c but after at least one:

$$\begin{aligned} \mathit{poss}\text{-}\mathit{eff}^+_*(c) &:= \bigcup_{s \in E(c)} (\bigcup_{s' \in R_s(c)} s' \setminus s) \\ \mathit{poss}\text{-}\mathit{eff}^-_*(c) &:= \bigcup_{s \in E(c)} ((\bigcup_{s' \in R_s(c)} (F \setminus s')) \cap s) \end{aligned}$$

if $E(c) \neq \emptyset$ and $\mathit{poss}\text{-}\mathit{eff}^{+/-}_*(c) := \mathit{undef}$ otherwise.

Mandatory preconditions (cf. Def. 6) of c are facts that hold in every state for which there exists an executable refinement. So, they are required in every state in which a refinement of c shall be executed: $\mathit{prec}(c) := \bigcap_{s \in E(c)} s$ if $E(c) \neq \emptyset$ and $\mathit{prec}(c) := \mathit{undef}$ otherwise.

Important: These definitions for p.o. domains are actually not correct in the sense that in a given p.o. problem we do not consider c on its own but rather within a task network that usually contains further tasks unordered w.r.t. c. Those (or their subtasks) can interlock with the subtasks of c to enable the execution of some refinement. So the executability-enabling or resulting states of c (leaving open how exactly they are defined in such cases) can look totally different depending on which other tasks are present in a task network.

We introduced these definitions solely to define a weaker version that can be computed in polynomial time as also done by Olz, Biundo, and Bercher (2021) in the t.o. case. They showed that determining the preconditions and effects in a t.o. domain is computationally as hard as solving the respective planning problem, basically because one needs to check whether there is at least one executable refinement[1]. For practical exploitation this can often be too costly. Therefore, a relaxation has been introduced, which allows to find a subset of the original preconditions and effects efficiently. It is done by ignoring the primitive tasks preconditions as then only the tasks' ordering relation and occurrences need

[1]Note that not all complexity results can be transferred directly to p.o. domains because some proofs exploit the fact that only deterministic complexity classes $\mathcal{C}$ were considered, where it holds $\mathcal{C} = co\mathcal{C}$. This is not the case for all p.o. domains since, e.g., the plan existence problem for acyclic p.o. problems is NEXPTIME-complete (Alford, Bercher, and Aha 2015).

to be considered. So, the *precondition-relaxation* of a domain $\mathcal{D} = (F, A, C, M)$ is the domain $\mathcal{D}' = (F, A', C, M)$ with $A' = \{(\emptyset, add, del) \mid (prec, add, del) \in A\}$. Then, the *precondition-relaxed effects* $\mathit{eff}_*^{\emptyset+}(c)$, $\mathit{eff}_*^{\emptyset-}(c)$, $\mathit{poss\text{-}eff}_*^{\emptyset+}(c)$ and $\mathit{poss\text{-}eff}_*^{\emptyset-}(c)$ (cf. Def. 9) are defined just like the original ones but based on the precondition-relaxed variant of $\mathcal{D}$.

Analogue preconditions were defined slightly differently as removing them completely does not yield the expected result. A fact $f \in F$ is an *executability-relaxed precondition* of c if and only if for all t.o. refinements (ignoring executability) $\langle a_0 \dots a_n \rangle$ of c there exists an action a_i with $f \in prec(a_i)$ and there does not exist an action a_j with $j < i$ and $f \in add(a_j)$, where $i, j \in \{0 \dots n\}$ (cf. Def. 10).

The exploitation of the relaxed preconditions and effects is possible because they possess subset properties with regard to the actual ones so that they do not contain false candidates. However, one needs to pay attention to several small details: The sets $post_*^+(c) = \bigcap_{s \in E(c)} \bigcap_{s' \in R_s(c)} s' \subseteq \mathit{eff}_*^+(c) \cup prec(c)$, $post_*^+(c) = \mathit{eff}_*^-(c)$ are called *state-independent postconditions* (cf. Def. 5 by Olz, Biundo, and Bercher (2021)), which (in the case of positive ones) additionally contain facts that hold after the execution of every refinement but were not added explicitly. For t.o. domains it was shown that $prec^{\emptyset}(c) \subseteq prec(c)$ and $\mathit{eff}_*^{\emptyset+/-}(c) \subseteq post_*^{+/-}(c)$ if $E(c) \neq \emptyset$ (Olz, Biundo, and Bercher 2021). We would like to make two remarks on this. First, the precondition-relaxed effects can contain facts, which are also preconditions and therefore can be rather interpreted as postconditions than effects. As an example, consider a compound task c with only one method $(c, \langle(\{f\}, \{f\}, \emptyset)\rangle)$. Here f is contained in $\mathit{eff}_*^{\emptyset+}(c)$ but it is also needed to execute c. We still decided to call the sets $\mathit{eff}_*^{\emptyset+/-}(c)$ effects instead of postconditions since the definition is based on the actions' effects and getting also postconditions is more a byproduct than intention. Moreover, not all postconditions are captured by the sets $\mathit{eff}_*^{\emptyset+/-}(c)$. So, the definitions imply some counter-intuitive phenomena concerning postconditions and effects, however, we did not come up with a perfect solution that fixes every interpretation issue. Thus, one should consider carefully which properties are needed for the exploitation at hand and pick the right version accordingly. Second, if c does not have an executable refinement then $\mathit{eff}_*^{+/-}(c) = prec^{\emptyset}(c) = undef$ but the relaxed versions may contain facts. This can be seen, e.g., if c has only one method containing only the two actions $(\emptyset, \emptyset, \{f_1\})$ and $(\{f_1\}, \{f_2\}, \emptyset)$, which are ordered as given. Here, $\mathit{eff}_*^{\emptyset+}(c) = \{f_2\}$ but this sequence of tasks is never executable[2]. It is a direct consequence of reducing the reasoning complexity from EXPTIME (arb. t.o. domain) to P.

In the p.o. case one needs to keep in mind one more point when it comes to exploitation: As pointed out earlier there might be other tasks in a task network that can or *even must* be interleaved with the subtasks of c, which potentially add or delete the alleged preconditions or effects of c. So the sets $\mathit{eff}_*^{\emptyset+/-}(c)$ and $prec^{\emptyset}(c)$ can only be considered as preconditions and effects of c if no other tasks are ordered within the refinement of c.

[2]We thank the anonymous reviewer for providing the two examples.

Algorithm 1: Calculates the precondition-relaxed effects for all compound tasks

Input: $\mathcal{D} = (F, A, C, M)$, an HTN planning domain.
Output: The sets of precondition-relaxed effects of all compound tasks

```
1:  $poss\text{-}eff_*^{\emptyset+}(c) = poss\text{-}eff_*^{\emptyset-}(c) = eff_*^{\emptyset+}(c) = eff_*^{\emptyset-}(c) = \emptyset$ for all $c \in C$
2:  for all $f \in F$ do
3:      $\mathcal{D}' =$ RestrictToEffects($\mathcal{D}, f$)
4:      $C_\varepsilon =$ ComputeEmptyRefinements($\mathcal{D}'$)
5:      $\mathcal{D}'' =$ ShortenMethodsFromRight($\mathcal{D}', C_\varepsilon$)
6:      $M_R =$ DecompositionReachability($\mathcal{D}''$)
7:      for all $c \in C$ do
8:          if $\exists\,(c', tn) \in M_R(c) \wedge a \in tn : f \in add(a)$ then
9:              $poss\text{-}eff_*^{\emptyset+}(c) = poss\text{-}eff_*^{\emptyset+}(c) \cup \{f\}$
10:         if $\exists\,(c', tn) \in M_R(c) \wedge a \in tn : f \in del(a)$ then
11:             $poss\text{-}eff_*^{\emptyset-}(c) = poss\text{-}eff_*^{\emptyset-}(c) \cup \{f\}$
12:         if $c \notin C_\varepsilon$ then
13:             if $f \in poss\text{-}eff_*^{\emptyset+}(c) \wedge f \notin poss\text{-}eff_*^{\emptyset-}(c)$ then
14:                 $eff_*^{\emptyset+}(c) = eff_*^{\emptyset+}(c) \cup \{f\}$
15:             if $f \notin poss\text{-}eff_*^{\emptyset+}(c) \wedge f \in poss\text{-}eff_*^{\emptyset-}(c)$ then
16:                 $eff_*^{\emptyset-}(c) = eff_*^{\emptyset-}(c) \cup \{f\}$
17: return $poss\text{-}eff_*^{\emptyset+}(c), poss\text{-}eff_*^{\emptyset-}(c), eff_*^{\emptyset+}(c), eff_*^{\emptyset-}(c)$ for all $c \in C$
```

Inference Algorithms

The proofs of Theorems 6 (on the poly-time decidability of possible effects) and Corollary 7 (guaranteed effects) as well as of Theorem 7 (on the poly-time decidability of preconditions) by Olz, Biundo, and Bercher (2021) essentially describe procedures to infer precondition-relaxed effects and executability-relaxed preconditions in t.o. domains in polynomial time. We now generalize these procedures so that they can also handle partially ordered task networks and present corresponding pseudo code.

Algorithm 1 is the main procedure to compute precondition-relaxed effects based on the textual description in the proof of Theorem 7 by Olz, Biundo, and Bercher (2021). The major modifications for p.o. domains affect solely subroutine ShortenMethodsFromRight. We consider one fact $f \in F$ after another and curtail the domain according to several subroutines, listed in Algorithm 2.

- We keep only primitive actions that add or delete f as all others are irrelevant. Therefore, the function RestrictToEffects($\mathcal{D}, f$) that takes as input a domain $\mathcal{D} = (F, A, C, M)$ and a fact $f \in F$ and outputs the domain $\mathcal{D}' = (\{f\}, A', C, M')$, where $A' = \{(prec(a) \cap \{f\}, add(a) \cap \{f\}, del(a) \cap \{f\}) \mid a \in A\} \setminus \{(\emptyset, \emptyset, \emptyset)\}$ and M' is obtained from M by restricting the task net-

Algorithm 2: Auxiliary Functions

```
1: ▷ Returns $C_\varepsilon \subseteq C$, the set of compound tasks admitting an empty refinement. ◁
2: function EmptyRefinements($\mathcal{D} = (F, A, C, M)$)
3:     $C_\varepsilon = \emptyset$; $M' = M$; $setChanged = true$
4:     for all $m = (c, tn = (T, \prec, \alpha)) \in M$ do
5:         if $T = \emptyset$ and $c \notin C_\varepsilon$ then
6:             $C_\varepsilon = C_\varepsilon \cup \{c\}$
7:             $M' = M' \setminus \{m\}$
8:         if $\exists\, t \in T : \alpha(t) \in A$ then
9:             $M' = M' \setminus \{m\}$
10:    while $setChanged$ do
11:        $setChanged = false$
12:        for all $m = (c, tn = (T, \prec, \alpha)) \in M'$ do
13:            if $c \notin C_\varepsilon$ and $\forall t \in T : \alpha(t) \in C_\varepsilon$ then
14:                $C_\varepsilon = C_\varepsilon \cup \{c\}$
15:                $M' = M' \setminus \{m\}$
16:                $setChanged = true$
17:    return $C_\varepsilon$

18: ▷ Returns an updated domain, where only the right-most relevant tasks remain in all methods. ◁
19: function ShortenMethodsFromRight($\mathcal{D}, C_\varepsilon$)
20:    we assume that $\prec$ is minimal
21:    let $\prec^+$ be the transitive closure of $\prec$
22:    $M' = \emptyset$
23:    for all $m = (c, tn = (T, \prec, \alpha)) \in M$ do
24:        $T_{rem} = T_{check} = \{t \in T \mid \nexists t' : (t, t') \in \prec\}$
25:        while $T_{check} \neq \emptyset$ do
26:            select arbitrary $t \in T_{check}$
27:            $T_{check} = T_{check} \setminus \{t\}$
28:            if $\alpha(t) \in C_\varepsilon$ then
29:                $T' = \{t' \in T \mid (t', t) \in \prec \wedge \nexists \tilde{t} \in T_{rem} : \alpha(\tilde{t}) \notin C_\varepsilon \wedge (t', \tilde{t}) \in \prec^+\}$
30:                $T_{check} = T_{check} \cup (T' \setminus (T_{rem} \cap T'))$
31:                $T_{rem} = T_{rem} \cup (T' \setminus (T_{rem} \cap T'))$
32:        $\prec' = \{(t_1, t_2) \in \prec \mid t_1 \in T_{rem} \wedge t_2 \in T_{rem}\}$
33:        $M' = M' \cup \{(c, (T_{rem}, \prec', \alpha|_{T_{rem}}))\}$
34:    return $\mathcal{D}' = (F, A, C, M')$
```

works to tasks from $A' \cup C$ instead of $A \cup C$.

- The function EmptyRefinements($\mathcal{D}'$) is called on the restricted domain that computes the set of compound tasks admitting an empty refinement, $C_\varepsilon \subseteq C$, i.e. they can be decomposed to an empty task network. If a compound task c can be refined into an empty task network, we know that f can only be a possible but not a mandatory (positive or negative) precondition-relaxed effect.
- Moreover, if the last/right-most task in a task network is primitive or does not admit an empty refinement then this task determines whether the fact gets added or deleted. In a partially ordered task network there are potentially several tasks that can be executed lastly. Therefore, the function ShortenMethodsFromRight($\mathcal{D}', C_\varepsilon$) identifies all these tasks for all decomposition methods and removes tasks that are ordered in front of them. In a t.o. task network we have a clear order of tasks and can go from right to left, stopping as soon as we encounter a task (primitive or compound) that does not admit an empty refinement. In our p.o. setting we consider initially all task that do not have a successor. If some of them admit an empty refinement we also consider their predecessors except of those that also precede another already selected task. The same applies for them until we reach a fix point.
- DecompositionReachability($\mathcal{D}$) computes for all tasks $c \in C$ the set of methods that are still reachable via decomposition from c in the restricted domain.

Finally, the effects are determined task by task by analyzing all methods that are still reachable via decomposition from that task as described from line 7 to 16: If there is a reachable method containing an action a adding f then f is a possible positive precondition-relaxed effect because then there is a refinement of c containing a such that no other action adds or deletes f afterwards according to Olz, Biundo, and Bercher (2021). We can further conclude that f is even a guaranteed positive precondition-relaxed effect if it is a possible positive effect but not a possible negative one and c can not be refined into an empty refinement. The case for negative effects follows analogously.

To sum up, we can infer precondition-relaxed effects for compound tasks in p.o. domains like in t.o. domains with the difference in how to determine the relevant tasks that can be executed at the end. We found them after performing subroutine ShortenMethodsFromRight. Instead of computing and analyzing the set of reachable methods we could also perform some fix-point algorithm to propagate the effects up the hierarchy, i.e., we could annotate to each compound tasks whether f is added or deleted in its methods based on the primitive tasks (still for $\mathcal{D}''$). Afterwards we could do this again by taking also the annotated compound tasks into account. This can be repeated until there are no further annotations.

Proposition 1. *Algorithm 1 is sound and complete, i.e., it computes all and only precondition-relaxed effects of a compound task c given a domain $\mathcal{D} = (F, A, C, M)$ and $c \in C$.*

Proof. The proof by Olz, Biundo, and Bercher (2021) is based on the argument that by curtailing the domain as described we find and keep only those tasks in the task networks that can be at the last position adding or deleting a fact f in a linearization of the task network and also in a primitive refinement of c if they are still reachable through the hierarchy. We follow this idea but mainly concentrate on the modified part.

Primitive tasks that neither add nor delete f can be neglected so we remove them to ease notation and reasoning. The set C_ε then contains all compound tasks that can be decomposed into a refinement in which no action affects f. Now we want to determine the tasks (primitive and compound) that can be ordered at the last position in a linearization of a refinement of a task network as they determine whether f is a positive or negative effect, which is done in ShortenMethodsFromRight. If a compound task in a task network admits an empty refinement, then also its predecessors need to be considered. Consider some method's task network $(c, tn = (T, \prec, \alpha))$ and a task that remained, i.e. some $t \in T_{rem}$. Either t has no successor tasks then it can clearly be ordered last or all (transitive) successors admit an empty refinement since otherwise the latter condition in line 29 would be violated. So if all those tasks were decomposed into an empty task network then t is again the last task. Therefore, for all other tasks $t' \in T \setminus T_{rem}$ it holds that they have primitive or compound successors that can not be refined into empty task networks, i.e., there is always a succeeding relevant primitive task, which makes t' irrelevant so that we delete it. Note that if t' is compound we thereby also cut off its subtasks. After ensuring this property for all methods we only need to check, which primitive tasks can be reached from c via decomposition as all of them can be inductively ordered last in some t.o. refinement. □

The procedure by Olz, Biundo, and Bercher (2021) restricted t.o. domains runs in polynomial time. As we only adapted SHORTENMETHODSFROMRIGHT($\mathcal{D}', C_\varepsilon$), which is still polynomial, we can conclude that our modified algorithms still has polynomial runtime.

The algorithm to compute executability-relaxed preconditions follows basically the same idea with small adaptions, which is why we do not include its pseudo code as well. Instead of the function RESTRICTTOEFFECTS($\mathcal{D}, f$) we now keep primitive tasks that contain f in their precondition or add effect list. Instead of SHORTENMETHODSFROMRIGHT($\mathcal{D}', C_\varepsilon$), we now shorten from left to right. Then, f is an executability-relaxed precondition of c if there does not exist a reachable method containing an action that adds f and c does not admit an empty refinement in the domain restricted just to actions that require f as precondition.

Conclusion

We defined preconditions and effects of compound tasks in p.o. domains and extended an existing inference algorithm for t.o. task networks operating in polynomial time to p.o. domains. This opens up the possibility to exploit such information for planning systems solving p.o. HTN planning problems as well as for modeling assistance or the linearization of partially ordered domains.

References

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight Bounds for HTN Planning. In *ICAPS*, 7–15. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019. Bringing Order to Chaos – A Compact Representation of Partial Order in SAT-based HTN Planning. In *AAAI*, 7520–7529. AAAI Press.

Bercher, P.; Alford, R.; and Höller, D. 2019. A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations. In *IJCAI*, 6267–6275. IJCAI.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In *IJCAI*, 480–488. IJCAI.

Clement, B. J.; Durfee, E. H.; and Barrett, A. C. 2007. Abstract Reasoning for Planning and Coordination. *Journal of Artificial Intelligence Research (JAIR)*, 28: 453–515.

Dvořák, F.; Barták, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. Planning and Acting with Temporal and Hierarchical Decomposition Models. In *ICTAI*, 115–121. IEEE.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity results for HTN planning. *Annals of Mathematics and AI (AMAI)*, 18(1): 69–93.

Fernandez-Olivares, J.; Vellido, I.; and Castillo, L. 2021. Addressing HTN Planning with Blind Depth First Search. In *10th International Planning Competition: Planner and Domain Abstracts (IPC 2020)*, 1–4.

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

Goldman, R. P.; and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: the case of SHOP3. In *Proc. of the 12th European Lisp Symposium (ELS 2019)*, 73–80. ACM.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020. HTN Planning as Heuristic Progression Search. *JAIR*, 67: 835–880.

Lesire, C.; and Albore, A. 2021. pyHiPOP – Hierarchical Partial-Order Planner. In *10th International Planning Competition: Planner and Domain Abstracts (IPC 2020)*, 13–16.

Magnaguagno, M. C.; Meneguzzi, F. R.; and de Silva, L. 2021. HyperTensioN: A three-stage compiler for planning. In *10th International Planning Competition: Planner and Domain Abstracts (IPC 2020)*, 5–8.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *JAIR*, 20: 379–404.

Olz, C.; Biundo, S.; and Bercher, P. 2021. Revealing Hidden Preconditions and Effects of Compound HTN Planning Tasks – A Complexity Analysis. In *AAAI*, 11903–11912. AAAI Press.

Olz, C.; Wierzba, E.; Bercher, P.; and Lindner, F. 2021. Towards Improving the Comprehension of HTN Planning Domains by Means of Preconditions and Effects of Compound Tasks. In *Proceedings of the 10th Workshop on Knowledge Engineering for Planning and Scheduling (KEPS 2021)*.

Pellier, D.; and Fiorino, H. 2021. Totally and Partially Ordered Hierarchical Planners in PDDL4J Library. In *10th International Planning Competition: Planner and Domain Abstracts (IPC 2020)*, 17–18.

Schreiber, D. 2021. Lilotane: A Lifted SAT-Based Approach to Hierarchical Planning. *JAIR*, 70: 1117–1181.

Schreiber, D.; Pellier, D.; and Fiorino, H. 2019. Tree-REX: SAT-Based Tree Explorationfor Efficient and High-Quality HTN Planning. In *ICAPS*, 382–390. AAAI Press.

Tsuneto, R.; Hendler, J.; and Nau, D. 1998. Analyzing External Conditions to Improve the Efficiency of HTN Planning. In *AAAI*, 913–920. AAAI Press.

Waisbrot, N.; Kuter, U.; and Könik, T. 2008. Combining Heuristic Search with Hierarchical Task-Network Planning: A Preliminary Report. In *Proc. of the 21st Int. Florida Artificial Intelligence Research Society Conference (FLAIRS 2008)*, 577–578. AAAI Press.