Revealing Hidden Preconditions and Effects of Compound HTN Planning Tasks — A Complexity Analysis

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

Abstract

In Hierarchical Task Network (HTN) planning, compound tasks need to be refined into executable (primitive) action sequences. In contrast to their primitive counterparts, compound tasks do not show preconditions or effects. Thus, their implications on the states in which they are applied are not explicitly known: they are "hidden" in and depending on the decomposition structure. We formalize several kinds of preconditions and effects that can be inferred for compound tasks in totally ordered HTN domains. As relevant special case we introduce a problem relaxation which admits reasoning about preconditions and effects in polynomial time. We provide procedures for doing so, thereby extending previous work, which could only deal with acyclic models. We prove our procedures to be correct and complete for any totally ordered input domain. The results are embedded into an encompassing complexity analysis of the inference of preconditions and effects of compound tasks, an investigation that has not been made so far.

Introduction

Hierarchical Task Network (HTN) planning is a hierarchical approach to planning, where compound tasks are step-wise refined into more primitive tasks, until an executable plan of action is obtained (Erol, Hendler, and Nau 1996; Ghallab, Nau, and Traverso 2004). When modeling such planning problems, often a planning language is used in which compound tasks do neither show preconditions nor effects like their primitive action counterparts. This is for example the case in HDDL (Höller et al. 2020a), a standard description language for HTN planning problems, which was also used to describe the benchmark set of the International Planning Competition 2020 (Behnke et al. 2019)[1] on HTN planning. This is also in line with one of the standard formalizations of HTN planning (Geier and Bercher 2011; Alford, Bercher, and Aha 2015) that HDDL bases upon, which is also used, among others, for many of the more recent theoretical investigations (Bercher, Alford, and Höller 2019). Instead of describing state transitions, the decomposition of a compound task introduces other primitive and/or compound tasks until eventually a completely primitive plan is achieved featuring all necessary preconditions and effects. So, while – due to the eventual introduction of primitive tasks – compound tasks do have an impact on states, it is not directly visible or even specified in the model explicitly (Goldman 2009).

In contrast, a vast range of hierarchical planning formalisms *do* feature and exploit preconditions and effects of compound tasks (Bercher et al. 2016), e.g., the hybrid planning formalisms by Kambhampati, Mali, and Srivastava (1998) and Biundo and Schattenberg (2001). For example they are used to insert compound tasks and reason about executability in plan space-based hierarchical planning such as PANDA (Biundo and Schattenberg 2001; Schattenberg 2009), FAPE (Dvořák et al. 2014; Bit-Monnot, Smith, and Do 2016) or UMCP (Tsuneto, Hendler, and Nau 1998); or to generate solution plans that still contain compound tasks (Clement, Durfee, and Barrett 2007; Marthi, Russell, and Wolfe 2008; de Silva, Sardina, and Padgham 2009). Another prominent example is the established HTN formalism by Erol, Hendler, and Nau (1996). It features a rich set of constraints that can be expressed, such as the (immediately) *before* constraint, which enables one to define preconditions of compound tasks, even preconditions which are not required by actions of its primitive refinements. Similarly, it also allows to use (immediately) *after* constraints constraining the search to choose an refinement after which the specified state properties hold (even if the chosen refinement itself did not have them as effect, if some held already prior its execution).

In most of these works, the compound tasks' preconditions and effects are provided by the domain modeler, so it is up to him or her to ensure completeness and correctness of this information. Thus, the benefit of automated inference of this information is twofold: it reduces modelling effort and prevents modelling errors. Moreover, it provides information that could be exploited for search, e.g., for the *Upward* and *Downward* properties (Yang 1990) (which are essential for generating abstract solutions), or heuristics.

There is a huge range of state-based heuristics for classical (i.e., non-hierarchical) planning, which all center around properties of *states*. In HTN planning, however, it is unclear which impact a compound task's hierarchy has on states when not knowing its pre- and postconditions. Our paper bridges the gap between both, so it can serve as a basis to compute such state properties. We are optimistic that

[1] See ipc2020.hierarchical-task.net.

they can then be exploited by various kinds of HTN planners, such as heuristic state-based progression search (Höller et al. 2018, 2020b), heuristic plan space search (Dvořák et al. 2014; Bit-Monnot, Smith, and Do 2016; Bercher et al. 2017) or SAT-based approaches (Behnke, Höller, and Biundo 2018; Schreiber, Pellier, and Fiorino 2019) to compute effective heuristics or to prune the search space. Past works already show the benefit of incorporating some sort of preconditions and/or effects in progression search (Nau et al. 2003; Waisbrot, Kuter, and Könik 2008; Goldman and Kuter 2019) (via the exploitation of method preconditions), plan space search (Tsuneto, Hendler, and Nau 1998), plan space-like search and plan coordination in a multi-agent setting (Clement, Durfee, and Barrett 2007), as well as in a compilation to SAT (Schreiber, Pellier, and Fiorino 2019) thus showing the potential benefit of preconditions and effects of compound tasks (or their methods).

Related Work Despite the potential of compound tasks' preconditions and effects, only little work was done in automatically inferring them. Clement, Durfee, and Barrett (2007) provide algorithms to compute such information for partially ordered, acyclic HTN models that feature time. Based on their and Thangarajah and Padgham (2011)'s work de Silva, Sardina, and Padgham (2016) provide such algorithms running in polynomial time for a Belief-Desire-Intention (BDI) agent programming language, which is a generalization of totally ordered HTN problems. Yao, de Silva, and Logan (2016) extend this work to a non-deterministic action model. Tsuneto, Hendler, and Nau (1998) automatically extract what they call *external conditions*, which can be regarded as preconditions of methods. However, these are based on conditions that are *predefined* by a domain modeler, i.e., their methods include specifications of several conditions given in the domain model upon which these external conditions are defined. Moreover, their algorithm is not complete, so they can not find all external conditions. Ilghami et al. (2005) learn preconditions of methods from user-provided training data.

Core Contributions We define various kinds of (inferred) preconditions and effects of a compound task. These include *guaranteed* effects, i.e., effects induced by *all* refinements of the task as well as *possible* effects generated just by *some* of them. We analyze the computational complexity of reasoning about these preconditions and effects, which range from **PSPACE**- to **EXPTIME**-completeness depending on task hierarchy properties. We furthermore introduce a new problem relaxation – *executability-relaxed HTN problems* – and show that here reasoning about preconditions and effects can be done in polynomial time. For this we describe procedures that extend previous work from the literature in that they are complete and can deal with arbitrary recursion.

Formal Framework

HTN Planning Formalism

We consider totally ordered (t.o.) HTN planning domains and problems. We base our formalization on the one by Geier and Bercher (2011), simplified by Behnke, Höller, and Biundo (2018) to totally ordered domains/problems.

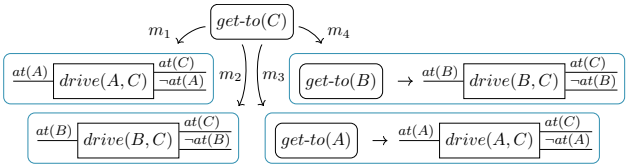

Figure 1: A compound task *get-to*(C) together with its methods as part of a transportation domain, which models the task to deliver packages to different locations.

A *STRIPS planning domain* $\mathcal{D} = (F, A)$ consists of a finite set of facts F and actions A, respectively. An action $a = (prec, add, del) \in A$ is described by its *preconditions* $prec(a) \subseteq F$ and its *effects* $add(a), del(a) \subseteq F$ (the *add, resp. delete effects*). An action $a \in A$ is *applicable* in a state $s \in 2^F$ if $prec(a) \subseteq s$. If applicable to s and applied to it, it produces the successor state $s' = (s \setminus del(a)) \cup add(a)$. A sequence of actions $\bar{a} = \langle a_0 \dots a_n \rangle$ with $a_i \in A$ for $0 \leq i \leq n$ is applicable in a state s_0 if and only if for all $0 \leq i < n$ a_i is applicable in s_i, where s_{i+1} results from applying a_i in s_i. In addition to a planning domain, a *STRIPS planning problem* $\Pi = (\mathcal{D}, s_I, g)$ contains an *initial state* $s_I \in 2^F$ and a *goal description* $g \subseteq F$ that implicitly specifies a set of goal states $s \supseteq g$. A sequence of actions $\langle a_0 \dots a_n \rangle$ is a solution to Π if and only if it is applicable in s_I and results in a goal state.

In a *t.o. HTN planning domain* $\mathcal{D} = (F, A, C, M)$ the sets F and A are defined as before, but the actions A are now referred to as *primitive tasks* as well. $\mathcal{D}$ also contains a finite set C of *compound tasks*, symbols that serve as abstractions for a collection of primitive and compound tasks. Together, they form the set of *tasks* $T = A \cup C$. A *t.o. task network* tn is a (possibly empty) finite sequence of tasks $\bar{t} = \langle t_0 \dots t_n \rangle \in T^*$. Compound tasks can be refined by a sequence of primitive and/or compound tasks according to the set of *decomposition methods* $M \subseteq C \times T^*$. A method $m = (c, \bar{t}) \in M$ *decomposes* a compound task $c \in C$ within a task network $tn_1 = \langle \bar{t}_1\, c\, \bar{t}_2 \rangle$ into a task network $tn_2 = \langle \bar{t}_1\, \bar{t}\, \bar{t}_2 \rangle$, written $tn_1 \rightarrow_{c,m} tn_2$. We write $tn \rightarrow tn'$ if there is a (possible empty) sequence of methods transforming tn into tn'. We then call tn' a refinement of tn. A *t.o. HTN planning problem* $\Pi = (\mathcal{D}, s_I, tn_I, g)$ contains the domain $\mathcal{D} = (F, A, C, M)$, an *initial state* $s_I \in 2^F$, an *initial task network* $tn_I \in T^*$, and a goal description $g \subseteq F$. Then, a sequence of actions $tn = \langle a_0 \dots a_n \rangle \in A^*$ is a solution to Π if and only if $tn_I \rightarrow tn$, tn is applicable in s_I, and results in a goal state.

As a running example we consider the part of a small transportation domain, which plans the tour a truck needs to take in order to deliver packages to different locations. A fraction of the domain is shown in Fig. 1. We have three locations A, B, and C and therefore our set of facts is $F = \{at(A), at(B), at(C)\}$. One can drive from one location to another given by the primitive actions $drive(x, y) = (\{at(x)\}, \{at(y)\}, \{at(x)\})$ for every com-

bination of $x, y \in \{A, B, C\}$, $x \neq y$. Note that we illustrate delete effects in pictures by a preceding $\neg$. Moreover, there are three compound tasks *get-to*(A), *get-to*(B), and *get-to*(C), each of which can be refined according to four different decomposition methods. In Fig. 1 this is exemplarily shown for the task *get-to*(C): Either we can drive directly to location C (modeled by the two primitive actions in m_1 and m_2) or drive via a different location to C. The compound tasks *get-to*(A) and *get-to*(B) can be decomposed likewise, which causes cycles in the domain model.

Effects of Compound Tasks

Before we begin to define effects of compound task we need to define the set of all states in which a given compound task is executable and in which an execution can result:

Definition 1 (Executability-Enabling States)**.** *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain and $c \in C$ a compound task. Then $E(c) \subseteq 2^F$ is the set of all states $s \in 2^F$ for which there exists a sequence of actions $\bar{a}$, such that $c \rightarrow \bar{a}$ and $\bar{a}$ is applicable in s.*

Definition 2 (Resulting States)**.** *Let $\mathcal{D} = (F, A, C, M)$ be a domain, $c \in C$ a compound task, and $s \in 2^F$ a state. Then $R_s(c) \subseteq 2^F$ is the set of all states into which the execution of c in s can result, i.e., $R_s(c)$ is the set of all* (resulting*) states $s' \in 2^F$ for which there exists a sequence of actions $\bar{a}$, such that $c \rightarrow \bar{a}$, $\bar{a}$ is applicable in s, and $\bar{a}$ executed in s results in s'.*

Now we start with defining *state-dependent postconditions*, which are state properties (true or false facts) that are guaranteed to hold directly after the execution of a compound task in a given state. Our definition roughly resembles Def. 4 by de Silva, Sardina, and Padgham (2016) defining what they call *Must Literals*.

Definition 3 (State-Dependent Postconditions)**.** *Let $\mathcal{D} = (F, A, C, M)$ be a domain, $c \in C$ a compound task, and $s \in 2^F$ a state. Then,*

- *the set of* state-dependent positive postconditions *of c w.r.t. s is $post_s^+(c) := \bigcap_{s' \in R_s(c)} s'$ and*
- *the set of* state-dependent negative postconditions *of c w.r.t. s is $post_s^-(c) := F \setminus \bigcup_{s' \in R_s(c)} s'$*

if $R_s(c) \neq \emptyset$ and, else $post_s^+(c) = post_s^-(c) := undef$.

According to this definition, given some state $s \in 2^F$, and a compound task $c \in C$, the set of *positive* postconditions is simply the set of all facts that hold in any state after the successful execution of any primitive decomposition of c. Similarly, the set of *negative* postconditions is the set of all facts that do not hold in any such successor state. By defining them as *undefined* in case there does not exist an executable refinement we can express the semantic difference to the case where the sets are empty, i.e., where there do not exist common facts in the resulting states. Let us consider the state $s = \{at(A)\}$ and the task *get-to*(C) in our running example. In every executable refinement the truck drives from location A to C possibly via B or by passing some locations more than once. However, it will always result in the state $s' = \{at(C)\}$, so $post_s^+(\textit{get-to}(C)) = \{at(C)\}$ and $post_s^-(\textit{get-to}(C)) = \{at(A), at(B)\}$.

Now we obtain state-*independent* postconditions by considering those facts that hold independent of the chosen refinement and state as long as the refinement is executable, i.e., $post_*^+(c) := \bigcap_{s \in E(c)} post_s^+(c)$ and $post_*^-(c) := \bigcap_{s \in E(c)} post_s^-(c)$ if $R_s(c) \neq \emptyset$ and $post_*^{+/-}(c) := undef$ otherwise.

Some postcondition might only hold because it was already true in the state prior to execution and was not changed by the compound task. In order to exclude such cases, we differentiate between *postconditions* and *effects*.

Definition 4 (State-Independent Effects)**.** *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain and $c \in C$ a compound task. Then,*

- *the set of* state-independent positive effects *of c is $\mathit{eff}^+_{E(c)}(c) := \bigcap_{s \in E(c)} post_s^+(c) \setminus \bigcap_{s \in E(c)} s$ and*
- *the set of* state-independent negative effects *of c is $\mathit{eff}^-_{E(c)}(c) := \bigcap_{s \in E(c)} post_s^-(c)$.*

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

Since we will later also introduce effects that hold only in *some* refinements, we sometimes write *guaranteed effects* to prevent confusion. To ease notation, we will write $\mathit{eff}^+_*(c)$ and $\mathit{eff}^-_*(c)$ as abbreviations for $\mathit{eff}^+_{E(c)}(c)$ and $\mathit{eff}^-_{E(c)}(c)$, respectively. We write $\mathit{eff}^+_s(c)$ and $\mathit{eff}^-_s(c)$ when $E(c)$ is cut down to a singleton $\{s\}$ in the above equations to describe state-*dependent* effects in analogy to the respective postconditions of Def. 3.

In case of *get-to*(C) in our running example $at(C)$ is even a state-independent positive effect and postcondition since it doesn't matter at which location we start, we will always end up at location C. Note that $at(A)$ and $at(B)$ are *not* state-independent negative effects or postconditions because executing the refinement using m_1 (i.e., $drive(A, C)$) in the state $\{at(A), at(B)\}$ results in the state $\{at(B), at(C)\}$ in which $at(B)$ does still hold. Since the state $\{at(A), at(B)\}$ would normally be considered inconsistent one can consider restricting the set $E(c) \subseteq 2^F$ further, e.g., by exploiting mutex relations.

So far, we were only concerned with definitions for postconditions and effects that are *guaranteed* to hold independently of the chosen decomposition methods. We now consider *possible effects*, effects the occurrence of which depends on the chosen refinement.

Our definition will consider all effects that could possibly hold after the execution of a suitable primitive refinement of the compound task. This resembles the idea behind *may conditions* as introduced by Clement, Durfee, and Barrett (2007).

Definition 5 (Possible State-Independent Effects)**.** *Let $\mathcal{D} = (F, A, C, M)$ be a domain, $c \in C$ a compound task, and $s \in 2^F$ a state. We first define possible postconditions:*

- *$\mathit{poss\text{-}post}_s^+(c) := \bigcup_{s' \in R_s(c)} s'$ and*
- *$\mathit{poss\text{-}post}_s^-(c) := \bigcup_{s' \in R_s(c)} (F \setminus s')$.*

Now we can define possible effects:

- *The set of* possible state-independent positive effects *of c is $\mathit{poss\text{-}eff}^+_{E(c)}(c) := \bigcup_{s \in E(c)} (\mathit{poss\text{-}post}_s^+(c) \setminus s)$ and*

- *the set of* possible state-independent negative effects *of* c *is* $\textit{poss-eff}^-_{E(c)}(c) := \bigcup_{s \in E(c)} (\textit{poss-post}^-_s(c) \cap s)$

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

We write $\textit{poss-eff}^+_*(c)$ and $\textit{poss-eff}^-_*(c)$ when we use $E(c)$ as defined and we write $\textit{poss-eff}^+_s(c)$ and $\textit{poss-eff}^-_s(c)$ as a shorthand for using $\{s\}$ in the subscript instead of $E(c)$ in order to refer to *possible state-dependent* effects.

We have seen that $at(A)$ and $at(B)$ are no state-independent negative effects of *get-to*(C) in our running example but now we can say that they are at least possible state-independent negative effects.

We also would like to make aware of a difference in the semantics of *guaranteed effects* and *possible effects*. In both definitions, effects are sets of facts. In case of guaranteed effects, the respective set can be interpreted as a conjunction, i.e., all of them hold in the same state. For *possible effects*, however, our definition only provides the needs to state for every *single* effect fact e that there is a state right after c's execution in which it holds. But two effects e_1 and e_2 might only hold in two different states.

Preconditions of Compound Tasks

Similar to the effects of a compound task we can also define their preconditions. We start with a basic definition in correspondence to the state-independent (guaranteed) effects of a compound task.

Definition 6 (Mandatory Preconditions)**.** *Let* $\mathcal{D} = (F, A, C, M)$ *be a planning domain and* $c \in C$ *a compound task. Then, the set of* mandatory preconditions *of* c *is* $prec(c) := \bigcap_{s \in E(c)} s$ *if* $E(c) \neq \emptyset$ *and* $prec(c) := \textit{undef}$ *otherwise.*

Mandatory preconditions are the facts that are required to hold in order to execute a compound task, i.e., none of them may not hold. However, it is not guaranteed that there exists an executable refinement in a state in which only these preconditions hold: They are necessary but not a sufficient criteria for executability. We introduce a sufficient criterion at the end of the subsection. The set of mandatory preconditions of the compound task in Fig. 1, e.g., is empty as the tasks in m_1 and m_2 do not have preconditions in common.

Our definition of preconditions is closely related to the concept of *method preconditions* known from the SHOP family (Nau et al. 2003; Goldman and Kuter 2019). These systems attach each individual method their own (hand-modeled) precondition, meaning that the respective method can be used for decomposition only if this precondition holds. We can *infer* these preconditions by introducing a new artificial compound task for the method in question and compute its precondition. This seems especially useful when methods are learned in the first place (Lotinac and Jonsson 2016; Gopalakrishnan, Muñoz-Avila, and Kuter 2018; Xiao et al. 2020).

As argued before, the preconditions are just a necessary condition for executability. Sufficient conditions can be defined in the following way, which is similar to what de Silva, Sardina, and Padgham (2016) define as precondition of a compound task:

Definition 7 (Executability-Enabling Precondition)**.** *Let* $\mathcal{D} = (F, A, C, M)$ *be a domain and* $c \in C$ *a compound task. Then we call a set of facts* executability-enabling precondition *of* c*,* $exEn(c) \subseteq F$*, if and only if for all states* s *with* $exEn(c) \subseteq s$ *there exists an executable refinement of* c*.*

In our running example, the set $\{at(A)\}$ would be an executability-enabling precondition of *get-to*(C) because it can be decomposed by using method m_1 to a refinement executable in $s = \{at(A)\}$. For the same reason, $\{at(B)\}$ would also be such a precondition, because it allows an executable refinement in $s = \{at(B)\}$ using m_2.

From the examples given, it is easy to see that there is not just one unique executability-enabling precondition since different refinements can be executable in arbitrarily different states. We also note the property that F is an executability-enabling precondition if and only if there exists an initial state such that the planning problem is solvable. This, however, does not tell us anything about which facts are mandatory preconditions (if any), simply because more actions can be executed in larger states due to restricting to positive preconditions.

General Complexity Results

Checking whether a fact is a precondition or effect implies solving the induced plan existence problem for the respective compound task, i.e., checking whether there is a solution. The hardness of doing so depends on properties of the available methods and their tasks. We thus briefly recap these structural properties.

Deciding arbitrary recursive HTN problems is undecidable (Erol, Hendler, and Nau 1996), yet it becomes decidable in **EXPTIME** when restricting the initial task network and all decomposition methods to total order (Erol, Hendler, and Nau 1996). **EXPTIME**-hardness was shown by Alford, Bercher, and Aha (2015). Erol, Hendler, and Nau (1996) proved **PSPACE**-completeness for both totally and partially ordered *regular* HTN problems, which are problems where in each decomposition method there is at most one compound task, which has to be the last one. A generalization thereof was given by Alford et al. (2012), called *tail-recursive problems*. Here, task networks may have more than one compound task, but recursion is only allowed through the very last task in any task network. In the total order setting these problems remain **PSPACE**-complete (Alford, Bercher, and Aha 2015). Also t.o. acyclic problems were shown to be **PSPACE**-complete (Alford, Bercher, and Aha 2015). Remember that all those (deterministic) complexity classes are closed under complement. This means that deciding whether a problem does not have a solution is as hard as deciding whether it has a solution. With *structural properties* we refer to these properties required for an HTN planning problem to be considered *acyclic*, *regular*, *tail-recursive*, or *arbitrarily recursive*.

With $\mathcal{D}|_c$ we denote the domain $\mathcal{D}$ restricted to all subtasks and their methods contained in some refinement of c, thereby eliminating all parts irrelevant for c.

Effects

Due to the way in which the respective proofs build upon each other, we start by investigating *possible* effects.

Theorem 1. *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain, $c \in C$ a compound task, and $f \in F$ a fact. Let $\Pi' = (\mathcal{D}', s'_I, tn'_I, g')$ be a planning problem, where $\mathcal{D}|_c$ and $\mathcal{D}'$ have the same structural properties. Deciding $f \in poss\text{-}eff^+_*(c)$ is as hard as (i.e., with matching lower and upper bounds) deciding whether Π' has a solution.*

Proof. We show membership by transforming the decision problem $f \in poss\text{-}eff^+_*(c)$ for $\mathcal{D}$ into a plan existence problem with the same structural properties as $\mathcal{D}$. Therefore, we construct a planning problem $\Pi' = (\mathcal{D}', s'_I, tn'_I, g')$, where $\mathcal{D}' := \mathcal{D}$, $s'_I := F \setminus \{f\}$ and $g' = \{f\}$. As we do not consider negative preconditions we know that if there *exists* a state (in which f does not hold) in which c is applicable then c is also applicable in s'_I. Thus, it holds $f \in poss\text{-}eff^+_*(c)$ if and only if Π' is solvable.

Hardness is obvious since deciding whether f is a possible effect contains the question whether there exists an executable refinement of c in at least one state, i.e., c can be seen as the initial task of a planning problem. □

As a special case we get the same result for one specific state s by simply using $s'_I = s$ instead of $s'_I = F \setminus \{f\}$ in the proof assuming that $f \notin s$ since otherwise f is clearly at most a postcondition but not an effect.

Corollary 1. *Thm. 1 also holds for $f \in poss\text{-}eff^+_s(c)$.*

We can also decide $f \in poss\text{-}eff^-_*(c)$ by adding a further fact $not\text{-}f$ ($not\text{-}f$ encodes that f is false) to F and modifying actions in the previous proof. All actions that delete f additionally add $not\text{-}f$ and all actions that add f delete $not\text{-}f$. Setting the initial state to $s'_I = F$ and the goal to $g' = \{not\text{-}f\}$ ensures that a plan deletes f.

Corollary 2. *Thm. 1 also holds for $f \in poss\text{-}eff^-_*(c)$ and, consequently, for $f \in poss\text{-}eff^-_s(c)$.*

Theorem 2. *Let $\mathcal{D} = (F, A, C, M)$ be a domain, $c \in C$ a compound task, and $f \in F$ a fact. Let $\Pi' = (\mathcal{D}', s'_I, tn'_I, g')$ be a planning problem, where $\mathcal{D}|_c$ and $\mathcal{D}'$ have the same structural properties mentioned earlier[2]. Deciding $f \in eff^+_*(c)$ is as hard as (i.e., with matching lower and upper bounds) deciding whether Π' has a solution.*

Proof. This time we construct *two* planning problems to show membership[3]. Let $\mathcal{D} = (F, A, C, M)$ be a planning domain, $c \in C$ a compound task, and $f \in F$ a fact. First, we check whether there actually exists an executable refinement which has f as an effect, which we can do as shown in the proof of Thm. 1. If the respective planning problem is not solvable stop with $f \notin eff^+_*(c)$. Otherwise, we need to verify that executing c can never result in a state in which f does not hold.

Therefore, we construct again a planning problem $\Pi' = (\mathcal{D}', s'_I, tn'_I, g')$ with $\mathcal{D}' = (V', A', C', M') := \mathcal{D}$ so that we can find out whether the original problem can lead to a state in which f does not hold, i.e. we check for unsolvability. First set $F' = F \cup \{not\text{-}f\}$, $s'_I = F \setminus \{f\}$, and set $g' = \{not\text{-}f\}$. Then, modify every action $a \in A'$ with $f \in add(a)$ by adding $not\text{-}f$ to $del(a)$. Analogously, add $not\text{-}f$ to $add(a)$ if $f \in del(a)$. Moreover, we need to guarantee that exactly either f or $not\text{-}f$ hold in the "initial" state. This can be encoded by introducing a further compound task c' with two methods $m_1 = (c', \langle(\emptyset, \{f\}, \emptyset)c\rangle)$ and $m_2 = (c', \langle(\emptyset, \{not\text{-}f\}, \emptyset)c\rangle)$. So we need to set $tn'_I = \langle c'\rangle$. Now, there are two kinds of plans solving the problem Π' depending on whether m_1 or m_2 is chosen to decompose c'. All executable plans resulting from using m_1 (which lead to f holding in the "initial" state) will *delete* f as $not\text{-}f$ must be added. In all executable plans resulting from decomposing with m_2 either f is never added (although it did not hold initially) or f is added but deleted later again to ensure that $not\text{-}f$ holds at the end. Thus, if Π' is solvable we get $f \notin eff^+_*(c)$ and otherwise $f \in eff^+_*(c)$.

Hardness follows without adjustments to the hardness proof of Thm. 1. □

The case $f \in eff^+_s(c)$ can be shown again by using $tn'_I = \langle c\rangle$ and $s'_I = s \cup \{not\text{-}f\}$ instead of $s'_I = F \setminus \{f\}$ in the last proof assuming that $f \notin s$. For similar reasons as for Cor. 2 the results also hold for the respective negative effects.

Corollary 3. *Thm. 2 also holds for $f \in eff^+_s(c)$, $f \in eff^-_s(c)$, and $f \in eff^-_*(c)$.*

Finally, we can state the complexity:

Corollary 4. *Let $c \in C$ and $X \in \{E(c)\} \cup \{s \mid s \in 2^F\}$. Deciding $f \in eff^{+/-}_X(c)$ and $f \in poss\text{-}eff^{+/-}_X(c)$ is:*

- **PSPACE**-complete *if c is acyclic, regular, or tail-recursive, and it is*
- **EXPTIME**-complete *in general.*

Preconditions

Analogously to the effects, the computational complexity of finding the set of mandatory preconditions of a compound task is also as hard as the respective planning problem.

Theorem 3. *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain, $c \in C$ a compound task, $s \in 2^F$ a state, and $f \in F$ a fact. Let $\Pi' = (\mathcal{D}', s'_I, tn'_I, g')$ be a planning problem, where $\mathcal{D}|_c$ and $\mathcal{D}'$ have the same structural properties mentioned earlier[2]. Deciding whether $f \in prec(c)$ is as hard as (i.e., with matching lower and upper bounds) deciding whether Π' has a solution.*

Proof. Like previously we show membership by transforming the decision problem $f \in prec(c)$ for $\mathcal{D}$ into two[3] plan existence problems with the same structural properties as $\mathcal{D}$. Therefore, let $\mathcal{D} = (F, A, C, M)$ be the given planning domain, $c \in C$ a compound task, and $f \in F$ a fact. First, we construct the planning problem $\Pi' = (\mathcal{D}, s'_I, tn'_I, g')$, where $s'_I = F \setminus \{f\}$, tn'_I is the task network containing just c and

[2]For which the plan existence problem lies in a deterministic complexity class X, where it holds that $X =$ co-X.

[3]Calling two subroutines is valid since we consider only deterministic complexity classes.

$g' = \emptyset$. If Π' is solvable we know that $f \notin prec(c)$. Otherwise, i.e. if Π' is unsolvable, we need to construct a second planning problem Π'' in order to check that Π' was unsolvable because of f being absent from the initial state. So let $\Pi'' = (\mathcal{D}, s''_I, tn'_I, g')$ but this time $s''_I = F$. Now, if Π'' is solvable then $f \in prec(c)$, otherwise there does not exist any executable refinement of c, so $prec(c)$ is undefined in this case.

Hardness: Deciding whether a fact f is included in the set of preconditions of a compound task involves the question whether there exists an executable refinement in some state in which f holds. So there must at least exist an executable refinement in the state $s = F$, which needs to be solved by the respective plan existence problem. Note that taking the entire set F as initial state does not change the hardness. □

There arise two decision problems regarding executability-enabling preconditions: Deciding whether a fact f is contained in at least one of those precondition sets and deciding whether a set of facts is an executability-enabling precondition. Clearly, the latter is simply the plan existence problem. For the former, every fact $f \in F$ is contained in at least one executability-enabling precondition if there exists an executable refinement (due to missing negative preconditions). Thus:

Proposition 1. *Let $\mathcal{D} = (F, A, C, M)$ be a domain, $c \in C$ a compound task, and $f \in F$ a fact. Let $\Pi' = (\mathcal{D}', s'_I, tn'_I, g')$ be a planning problem, where $\mathcal{D}|_c$ and $\mathcal{D}'$ have the same structural properties. Deciding whether there exists an executability-enabling precondition $exEn(c) \subseteq F$ with $f \in exEn(c)$ is as hard as (i.e., with matching lower and upper bounds) deciding whether Π' has a solution.*

In Thms. 2 and 3 we are explicitly restricted to domains where the plan existence problem lies in a deterministic complexity class. Here, in Prop. 1, as well as in Thm. 1 this is not necessary because we perform just a many-one reduction and do not consider the complement problem.

The last proposition also shows us that deciding whether there exists an initial state such that a planning domain together with an initial task network admits a solution is as hard as the respective plan existence problem where the initial state is already given, simply by checking whether the problem with F being the initial state is solvable.

Corollary 5. *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain, tn_I a task network and $g \subseteq F$ a goal description. Moreover, let $\Pi' = (\mathcal{D}', s'_I, tn'_I, g')$ be a planning problem, where $\mathcal{D}$ and $\mathcal{D}'$ have the same structural properties. Deciding whether there exists a state s_I such that $(\mathcal{D}, s_I, tn_I, g)$ is solvable is as hard as deciding whether Π' has a solution.*

Again, we sum up the complexity:

Corollary 6. *Let $c \in C$. Deciding $f \in prec(c)$ and $\exists\, exEn(c) : f \in exEn(c)$ is:*
- **PSPACE**-complete *if c is acyclic, regular, or tail-recursive, and it is*
- **EXPTIME**-complete *in general.*

We would like to mention an interesting relationship of Cor. 4 and Cor. 6 to a recent result about landmarks. Various kinds of landmarks have been analyzed, one of them are *fact landmarks* (Höller and Bercher 2021). Such a landmark is a fact that has to be true at some point in every HTN solution. Thus, our guaranteed effects are clearly landmarks. Höller and Bercher proved that checking whether a fact (or a task, or a method) is a landmark of an HTN planning problem is exactly as hard as checking whether the respective planning problem has *no* solution. This is in line with our results, since the hierarchical restrictions we studied can all be decided in a deterministic complexity class, where the complement has the same complexity, so checking for landmarks is as hard as checking for guaranteed effects in the total order setting. This shows the close relationship to our results. However, while guaranteed effects are landmarks, a landmark must neither be a mandatory precondition nor a guaranteed effect, since it could also be required or produced "within" all refinements, but neither hold at its beginning nor at its end.

Tractable Complexity Results

The main result from the previous section is that, unfortunately, computing preconditions and effects of compound tasks is as expensive as solving the planning problem in the first place. To reduce this complexity we investigate novel problem relaxations resulting in tractable algorithms.

Precondition-Relaxation (Basic Definitions)

We start by providing the necessary definitions for *precondition relaxation*.

Definition 8 (Precondition Relaxation)**.** *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain. Its* precondition-relaxation *is the domain $\mathcal{D}' = (F, A', C, M)$ with $A' = \{(\emptyset, add, del) \mid (prec, add, del) \in A\}$.*

Definition 9 (Effects of Precondition-Relaxed Tasks)**.** *Let $\mathcal{D}$ be a planning domain and $c \in C$ a compound task. We define $\mathit{eff}_*^{\emptyset+}(c)$, $\mathit{eff}_*^{\emptyset-}(c)$, $\mathit{poss\text{-}eff}_*^{\emptyset+}(c)$ and $\mathit{poss\text{-}eff}_*^{\emptyset-}(c)$ as shorthand for $\mathit{eff}_*^{+}(c)$, $\mathit{eff}_*^{-}(c)$, $\mathit{poss\text{-}eff}_*^{+}(c)$ and $\mathit{poss\text{-}eff}_*^{-}(c)$ that are based on the precondition-relaxed variant of $\mathcal{D}$, respectively.*

We do not need to differentiate between state-*dependent* and state-*independent* effects when ignoring preconditions, since all refinements are always applicable. That is, $\mathit{eff}_*^{\emptyset+}(c)$ ($\mathit{eff}_*^{\emptyset-}(c)$) equals $\mathit{eff}_s^{\emptyset+}(c)$ ($\mathit{eff}_s^{\emptyset-}(c)$) for any $s \in 2^F$.

The next theorem is of major importance as it relates the effects of relaxed problems to the original postconditions.

Theorem 4. *Let $\mathcal{D} = (F, A, C, M)$ be a domain and $c \in C$ a compound task. Then, $\mathit{eff}_*^{\emptyset+}(c) \subseteq post_*^{+}(c)$ and $\mathit{eff}_*^{\emptyset-}(c) \subseteq post_*^{-}(c)$ if $post_*^{+}(c)$ and $post_*^{-}(c)$ are defined.*

Proof. We need to show that every $f \in \mathit{eff}_*^{\emptyset+}(c)$ ($\mathit{eff}_*^{\emptyset-}(c)$) is also in $post_*^{+}(c)$ ($post_*^{-}(c)$), so we fix some $f \in F$. If $f \in \mathit{eff}_*^{\emptyset+}(c) \cup \mathit{eff}_*^{\emptyset-}(c)$ then for all states s in which the refinements of c are executed, f must (in case of $f \in \mathit{eff}_*^{\emptyset+}(c)$ – and must not in case of $f \in \mathit{eff}_*^{\emptyset-}(c)$) be contained in the final state produced. In the non-relaxed domain, the set of executable refinements in these states s can only be a

(not necessarily strict) subset. Thus, also the set of states resulting from their application is a subset and hence the guaranteed effect f in the relaxed domain is also one in the non-relaxed domain, as long as there exists at least one executable refinement, i.e., if $post^+_*(c)$ and $post^-_*(c)$ are not undefined. □

The set of effects of the relaxed tasks might contain facts that are also mandatory preconditions, so in this case they are not considered state-independent effects according to Def. 4. So the property $\mathit{eff}^{\emptyset+}_*(c) \subseteq \mathit{eff}^+_*(c)$ does not hold.

Unfortunately, the last theorem does not apply to *possible* effects. This is easy to see because the precondition relaxation can make further refinements executable that produce effects that can not be achieved in the non-relaxed domain.

Proposition 2. *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain and $c \in C$ a compound task. Then, $\mathit{poss\text{-}eff}^{\emptyset+}_*(c) \supseteq \mathit{poss\text{-}eff}^+_*(c)$ and $\mathit{poss\text{-}eff}^{\emptyset-}_*(c) \supseteq \mathit{poss\text{-}eff}^-_*(c)$.*

This limits the usefulness of computing precondition-relaxed possible effects. However, they can still be used to rule out some of the effects that are simply "mentioned" in a task's sub hierarchy. That is, any fact *not* appearing in $\mathit{poss\text{-}eff}^{\emptyset+}_*(c)$ or $\mathit{poss\text{-}eff}^{\emptyset-}_*(c)$ cannot be a possible effect in the original domain.

We now investigate checking for *preconditions*. If we relaxed (i.e., removed) the preconditions, similar to Def. 8, we could not infer them anymore – since they were gone. So instead we introduce *executability-relaxed* preconditions, which ignores preconditions regarding executability and just looks at *all* primitive refinements. Note that the two relaxations are semantically equivalent, so executability-relaxation (defined next) is just a different formal tool to express the same idea as for the effects. So we could also have defined precondition-relaxation (Def. 8) in the same way as in the following definition, but we decided not to, because Def. 8 is more intuitive (but just doesn't work for identifying certain preconditions as mentioned before).

We regard a fact as precondition if there is an action in every primitive refinement that requires this fact and none of the other actions in the same refinement adds it beforehand. This kind of precondition also resembles the idea behind a *must precondition* by Clement, Durfee, and Barrett (2007).

Definition 10 (Executability-Relaxed Precondition)**.** *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain and $c \in C$ a compound task. We call a fact f an* executability-relaxed precondition *of c if and only if for all primitive refinements (i.e., ignoring executability) $\langle a_0 \dots a_n \rangle$ of c there exists an action a_i with $f \in \mathit{prec}(a_i)$ and there does not exist an action a_j with $j < i$ and $f \in \mathit{add}(a_j)$, where $i, j \in \{0 \dots n\}$.*

Definition 11. *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain and $c \in C$ a compound task. Then the set of* executability-relaxed preconditions *of c is $\mathit{prec}^{\emptyset}(c) := \{f \in F \mid f$ is an executability-relaxed precondition of $c\}$.*

Again, we ensure that this relaxation is of practical usefulness as all relaxed preconditions are also preconditions in the original sense (provided the respective task is solvable at all, as otherwise its precondition would be undefined).

Theorem 5. *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain and $c \in C$ a compound task. Then, $\mathit{prec}^{\emptyset}(c) \subseteq \mathit{prec}(c)$ if $\mathit{prec}(c)$ is not undefined.*

Proof. If $f \in \mathit{prec}^{\emptyset}(c)$ then we know that in every refinement of c there is an action a with $f \in \mathit{prec}(a)$ and no other action adds f before a is executed. This means f must hold in a state in which the refinement is executed in order to be executable at all. Therefore $f \in \mathit{prec}(c)$ if there is at least one executable refinement in some state as otherwise $\mathit{prec}(c)$ would be undefined. □

Note that in some cases $\mathit{prec}^{\emptyset}(c) = \mathit{prec}(c)$ might hold (just take any problem without preconditions, then the sets coincide), whereas it does not always as can be seen by the following example: A compound task c might have two primitive refinements but only one of them contains an action having fact f in its preconditions. So in this case $f \notin \mathit{prec}^{\emptyset}(c)$. But if the second refinement is not executable, then $f \in \mathit{prec}(c)$.

Investigating the Computational Complexity

We now investigate the computational complexity of computing preconditions and effects in precondition-relaxed HTN planning (with goal condition). Closely related to this, de Silva, Sardina, and Padgham (2016, Alg. 1) provided an algorithm that computes a subset of their *must literals* for *acyclic* domains in polynomial time. They specify reasons for not being complete – one being not taking precondition formulae into account – which makes the underlying domain for which they implicitly compute postconditions closely related to precondition-relaxed domains. However, they do not state formal properties of the subset of must literals that can be found by their algorithm. Clement, Durfee, and Barrett (2007) provide algorithms to compute pre- and postconditions of compound tasks for *partially ordered temporal* but still *acyclic* HTN problems in polynomial time. While their algorithm of extracting them *does* take the interactions of preconditions and effects into account, they do not do so to a full extent. They consider only the immediate methods of tasks instead of the complete decomposition hierarchy, such that information about the interplay between preconditions and effects deeper in the hierarchy gets lost.

We provide proofs describing poly time-bounded procedures that decide whether a given fact is a (possible or guaranteed) effect or precondition of a compound task in a precondition-relaxed HTN domain. It can be used to compute a *complete* set of preconditions and effects for a precondition-relaxed HTN model (by running it for each fact) thereby generalizing the work by de Silva, Sardina, and Padgham (2016) and Clement, Durfee, and Barrett (2007) in that it is provably complete and it works for all t.o. HTN domains, including cyclic ones.

Effects We begin with checking for possible effects as they can be used to compute the guaranteed effects.

Theorem 6. *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain, $c \in C$, and $f \in F$. Deciding whether $f \in \mathit{poss\text{-}eff}^{\emptyset+}_*(c)$ ($f \in \mathit{poss\text{-}eff}^{\emptyset-}_*(c)$) holds can be done in polynomial time.*

Proof. Let $\mathcal{D} = (F, A, C, M)$ be some planning domain, $\mathcal{D}' = (F', A', C', M')$ its precondition-relaxation, $c \in C$ a compound task, and $f \in F$ a fact. We will curtail this domain in several steps such that we can gather our desired information from the remaining domain.

Primitive tasks that do not affect f can be neglected in the process of deciding whether f is a possible effect of c. We construct a new domain $\mathcal{D}'' = (F'', A'', C'', M'')$ to account for this. We delete all actions $a \in A'$ except of those with $f \in add(a) \cup del(a)$ from $\mathcal{D}'$. Furthermore, we set $F'' := F' \cap \{f\}$, which includes restricting $add(a)$ and $del(a)$ for each $a \in A''$ to $add(a) \cap \{f\}$ and $del(a) \cap \{f\}$, respectively. $C'' = C'$ remains unaffected, but M'' differs from M' due to restricting the action set to A''.

Based on this model our main observation is: If it were not possible to refine a compound task into an empty task network, then it would always be the very last task in the method that determines the outcome of a method, and the possible outcomes of a compound task would simply be the union of all outcomes of all its methods. Overall, we could simply propagate these effects in a bottom-up manner starting with the primitive task networks. However, tasks that *can* be refined into empty task networks are more complicated. So, instead, we need to *identify* which task could be responsible for adding or deleting f: If the last task of a method is primitive, the method behaves according to this action's effects. Otherwise, there are two cases that we need to consider: (1) It cannot be refined into an empty task network and thus results into a non-empty primitive plan. (2) It *can* be refined into an empty task network (which is likely due to our task elimination). Case (1) is analogous to the last task being primitive, as in each sub method there will be a last task adding an effect. We just have to check all the possibilities. In case (2), however, we need to investigate both the refinement of the task itself and its predecessor because of the possible refinement into an empty task sequence.

Therefore, we need to know whether a compound task can be decomposed into an empty task network ε. We do so by a simple bottom-up propagation that runs in polynomial time – an adaptation of Alford et al.'s (2014) procedure for deciding TIHTN problems without negative effects (Thm. 3.1). We start with all methods (c', tn') that have an empty task network $tn' = \varepsilon$ and mark their compound task c' as admitting an empty refinement. We continue marking all tasks c'' as admitting an empty refinement that have a method (c'', tn'') with a task network tn'' that consists only of compound tasks, which all must admit an empty refinement. We continue doing so until no more propagations are possible. In the worst case we mark just one compound task in each iteration resulting into a runtime of $\mathcal{O}(|M| \cdot |C|)$.

We can use this information to restrict the method set M'' further (to a set M'''). Then, we can simply infer the the possible effects by checking which primitive tasks remain reachable from c. So, for every method $m_1 = (c_1, tn_1 = \langle t_1 \dots t_n \rangle) \in M''$ we identify the right-most task t_i with $n \leq i \leq 1$ that is primitive or compound *without* admitting an empty refinement (if such a t_i exists). Then we remove all remaining tasks $t_1 \dots t_{i-1}$ from the task network tn_1 as only t_i to t_n can be responsible for the outcome of the given method m_1 (add the resulting method to M''', also add methods for which no t_i exists). This ensures that every remaining primitive task in a task network that can be reached from c could be the last task of a refinement of c which allows us to infer the possible effects later on. To prove this claim assume we restricted the set M''' to M'''', where only methods from M''' reside that are reachable from c (cutting off some prefixes can result in various tasks becoming unreachable). Now let $m_2 = (c_2, tn_2 = \langle a \bar{t}_2 \rangle) \in M''''$ be a method with a primitive task. We show that a can be a last task in some refinement of c. By construction, no tasks can occur before a, and $\bar{t}_2$ must be a sequence of compound tasks that admit an empty refinement (or they are ε, i.e., $tn_2 = \langle a \rangle$). Because m_2 is reachable by assumption c_2 must be contained in a number of task networks (at least one) of the form $tn_3 = \langle \bar{t}_{31} c_2 \bar{t}_{32} \rangle$ of some method(s) $m_3 = (c_3, tn_3) \in M''''$, where $\bar{t}_{31}$ and $\bar{t}_{32}$ are again (possibly empty) sequences of tasks and $\bar{t}_{32}$ can also be decomposed to ε by construction. We can repeat this argument inductively until we reach a method $m = (c, tn)$, where w.l.o.g. $tn = \langle \bar{t}_1 c_3 \bar{t}_2 \rangle$. Since $\bar{t}_2$ can also be decomposed to ε, there is a refinement of c with a being its last task.

We can conclude $f \in \mathit{poss\text{-}eff}_*^{\emptyset+}(c)$ ($f \in \mathit{poss\text{-}eff}_*^{\emptyset-}(c)$) if there is a method in M'''' that contains an action a with $f \in add(a)$ ($f \in del(a)$). All operations (including checking reachability of methods and compound tasks) can clearly be done in polynomial time, which proves the claim. □

Corollary 7. *Given a planning domain $\mathcal{D} = (F, A, C, M)$ and a compound task $c \in C$ we can decide $f \in \mathit{eff}_*^{\emptyset+}(c)$ ($f \in \mathit{eff}_*^{\emptyset-}(c)$) in polynomial time.*

Proof. It holds $f \in \mathit{eff}_*^{\emptyset+}(c)$ if and only if (1) $f \in \mathit{poss\text{-}eff}_*^{\emptyset+}(c)$, (2) $f \notin \mathit{poss\text{-}eff}_*^{\emptyset-}(c)$, and (3) c does not admit an empty refinement. Analogously, it holds $f \in \mathit{eff}_*^{\emptyset-}(c)$ if and only if (1) $f \in \mathit{poss\text{-}eff}_*^{\emptyset-}(c)$, (2) $f \notin \mathit{poss\text{-}eff}_*^{\emptyset+}(c)$, and (3) c does not admit an empty refinement. (1) and (2) can be decided in polynomial time (**P**) according to Thm. 6, and checking for an empty refinement is also in **P** as outlined in the proof of Thm. 6. □

Note that we can use the decision procedures described in the proofs of Thm. 6 and Cor. 7 to compute a complete set of possible and guaranteed effects simply by applying them to all $f \in F$ and adding them to the respective effect sets.

Regarding our example in Fig. 1 we illustrate how one can check whether $at(C)$ is a relaxed positive effect of *get-to*(C). First we delete all primitive actions that do not affect $at(C)$, which would be $drive(A, B)$ and $drive(B, A)$. The two compound tasks *get-to*(A) and *get-to*(B) admit an empty refinement afterwards since the two methods' task networks containing only $drive(A, B)$ and $drive(B, A)$, respectively, become empty. However, *get-to*(C) still does not admit an empty refinement, as the primitive tasks in the methods decomposing it are still there. Now we restrict the method set further (to M''' in the proof of Thm. 6). Therefore, we delete *get-to*(A) and *get-to*(B) from m_3 and m_4 because the rightmost tasks in those methods' task networks

are already primitive. We do not need to proceed with the remaining methods as we can see that no more task networks other than those of m_1 to m_4 are reachable from *get-to*(C) now. So we can conclude that indeed $at(C)$ is a possible and even guaranteed positive relaxed effect since every task network still reachable from *get-to*(C) contains an action adding it.

We can use the described polynomial-time procedure to find *the same effects* in the relaxed domain of the running example as in the original one. As future work it would be interesting to analyze to which extent this does also hold in benchmark and real world planning problems.

Preconditions Calculating the set of executability-relaxed preconditions can be done quite similarly.

Theorem 7. *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain, $c \in C$, and $f \in F$. It can be decided in polynomial time whether $f \in prec^{\emptyset}(c)$ holds.*

Proof sketch. In contrast to the proof of Thm. 6 we now analyze the *beginning* of the refinements whether a primitive action adds f before another action requires f as precondition. By verifying that c does not admit an empty refinement after deleting all actions $a \in A$ except of those with $f \in prec(a)$ we ensure that in every refinement of c there must exist an action that requires f as precondition. Afterwards we again take the original domain and delete all actions $a \in A$ except of those with $f \in prec(a)$ or $f \in add(a)$. Moreover, we curtail the methods like in the proof of Thm. 6 but instead of going from right to left we curtail them from left to right. Then we check whether there exists a method containing a primitive action adding f that is reachable from c via decomposition. If not (and only then) f is an executability-relaxed precondition of c. A more detailed proof can be found in the appendix. □

Conclusion

For totally ordered HTN planning, we provided formal definitions for a compound task's preconditions and effects to enable reasoning about them. We provided tight complexity bounds for computing these, which are **PSPACE**- or **EXPTIME**-complete depending on the hierarchical decomposition structure of the respective task. This means computing preconditions and effects is exactly as hard as solving the respective plan existence problem for the structural properties analyzed. We introduced *executability-relaxation* as a means to compute preconditions and effects in **P**. The respective algorithms extend work from the literature in the sense that they are provably complete and are not restricted to acyclic HTN models.

Appendix

Full Proof of Thm. 7. Let $\mathcal{D} = (F, A, C, M)$ be some planning domain, $c \in C$ a compound task and $f \in F$ a fact. In the previous proof of Thm. 6 we analyzed the refinements of c in terms of which tasks are the last ones affecting f. Now we have to look at the beginning of the refinements whether a primitive action adds f before another action requires f as precondition. Therefore, we can make use of the same procedure as described before but instead of looking at methods from right to left we curtail them from left to right.

We have to verify two things. Firstly, in every refinement of c there must exist an action that requires f as precondition. Secondly, there must not exist a refinement of c in which f is added by a primitive task before another action requires it.

We can check the first by considering the domain $\mathcal{D}' = (F', A', C', M')$ that is obtained from $\mathcal{D}$ by deleting all actions $a \in A$ except of those with $f \in prec(a)$ and setting $F' := F \cap \{f\}$, which again includes restricting $add(a)$ and $del(a)$ for each $a \in A'$ to $add(a) \cap \{f\}$, $del(a) \cap \{f\}$ and $prec(a) \cap \{f\}$, respectively. $C' = C$ remains unaffected, but M' differs from M due to restricting the action set to A'. If c admits an empty refinement in $\mathcal{D}'$ (the procedure is described in the last proof) we can conclude that there exists an refinement that does not require f for executability. In this case f is not an executability-relaxed precondition.

So assume c does not admit an empty refinement in $\mathcal{D}'$. We construct a new domain $\mathcal{D}'' = (F'', A'', C'', M'')$. It is again an adaption of $\mathcal{D}$ but this time we delete all actions $a \in A$ except of those with $f \in prec(a)$ or $f \in add(a)$. The remaining sets are restricted like before to $F' := F \cap \{f\}$ and $a \in A''$. Like in the proof of Thm. 6 we calculate all compound tasks that admit an empty refinement. Moreover, for every method $m_1 = (c_1, tn_1 = \langle t_1 \dots t_n \rangle) \in M''$ we identify the *left-most* task t_i with $1 \leq i \leq n$ that is primitive or compound *without* admitting an empty refinement (if such a t_i exists) and we remove all remaining tasks $t_{i+1} \dots t_n$ from the task network tn_1. With the same arguments like before we can conclude that there exists a refinement of c that contains a primitive task adding f before it is required by another task if and only if there still exists a method's task network in M'' containing a primitive action adding f that is reachable from c via decomposition. Thus, f is an executability-relaxed precondition of c if and only if no such method exists. □

References

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

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. 2012. HTN Problem Spaces: Structure, Algorithms, Termination. In *SoCS*, 2–9. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. 2014. On the Feasibility of Planning Graph Style Heuristics for HTN Planning. In *ICAPS*, 2–10. AAAI Press.

Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; Pellier, D.; Fiorino, H.; and Alford, R. 2019. Hierarchical Planning in the IPC. In *Proc. of the Workshop on the International Planning Competition*.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – Totally-Ordered Hierarchical Planning through SAT. In *AAAI*, 6110–6118. 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.

Bercher, P.; Höller, D.; Behnke, G.; and Biundo, S. 2016. More than a Name? On Implications of Preconditions and Effects of Compound HTN Planning Tasks. In *ECAI*, 225–233. IOS Press.

Bit-Monnot, A.; Smith, D. E.; and Do, M. 2016. Delete-Free Reachability Analysis for Temporal and Hierarchical Planning. In *ECAI*, 1698–1699. IOS Press.

Biundo, S.; and Schattenberg, B. 2001. From Abstract Crisis to Concrete Relief (A Preliminary Report on Combining State Abstraction and HTN Planning). In *ECP*, 157–168. AAAI Press.

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.

de Silva, L.; Sardina, S.; and Padgham, L. 2009. First Principles Planning in BDI Systems. In *AAMAS*, 1105–1112. IFAAMAS.

de Silva, L.; Sardina, S.; and Padgham, L. 2016. Summary Information for Reasoning About Hierarchical Plans. In *ECAI*, 1300–1308. IOS Press.

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.

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *IJCAI*, 1955–1961. AAAI Press.

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

Goldman, R. P. 2009. A Semantics for HTN Methods. In *ICAPS*, 146–153. AAAI Press.

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.

Gopalakrishnan, S.; Muñoz-Avila, H.; and Kuter, U. 2018. Learning Task Hierarchies Using Statistical Semantics and Goal Reasoning. *AI Communications* 31(2): 167–180.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020a. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *AAAI*, 9883–9891. AAAI Press.

Höller, D.; and Bercher, P. 2021. Landmark Generation in HTN Planning. In *AAAI*. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A Generic Method to Guide HTN Progression Search with Classical Heuristics. In *ICAPS*, 114–122. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020b. HTN Planning as Heuristic Progression Search. *Journal of Artificial Intelligence Research (JAIR)* 67: 835–880.

Ilghami, O.; Muñoz-Avila, H.; Nau, D. S.; and Aha, D. W. 2005. Learning Approximate Preconditions for Methods in Hierarchical Plans. In *Proc. of the 22nd Int. Conference on Machine Learning (ICML)*, 337–344. ACM.

Kambhampati, S.; Mali, A.; and Srivastava, B. 1998. Hybrid Planning for Partially Hierarchical Domains. In *AAAI*, 882–888. AAAI Press.

Lotinac, D.; and Jonsson, A. 2016. Constructing Hierarchical Task Models Using Invariance Analysis. In *ECAI*, 1274–1282. IOS Press.

Marthi, B.; Russell, S.; and Wolfe, J. 2008. Angelic Hierarchical Planning: Optimal and Online Algorithms. In *ICAPS*, 222–231. AAAI Press.

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

Schattenberg, B. 2009. *Hybrid Planning & Scheduling*. Ph.D. thesis, Ulm University, Germany.

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.

Thangarajah, J.; and Padgham, L. 2011. Computationally Effective Reasoning About Goal Interactions. *Journal of Automated Reasoning* 47(1): 17–56.

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.

Xiao, Z.; Wan, H.; Zhuo, H. H.; Herzig, A.; Perrussel, L.; and Chen, P. 2020. Refining HTN Methods via Task Insertion with Preferences. In *AAAI*, 10009–10016. AAAI Press.

Yang, Q. 1990. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence* 6(1): 12–24.

Yao, Y.; de Silva, L.; and Logan, B. 2016. Reasoning about the Executability of Goal-Plan Trees. In *Proc. of the 4th Int. Workshop on Engineering Multi-Agent Systems (EMAS 2016)*, 176–191. Springer.