Journal of Artificial Intelligence Research 82 (2025) 1407-1444 Submitted 09/2024; published 03/2025

An Extensive Empirical Evaluation of Inferring Preconditions and Effects of Compound Tasks in Ground HTN Planning Problems

Conny Olz CONNY.OLZ@ALUMNI.UNI-ULM.DE
Alexander Lodemann ALEXANDER.LODEMANN@UNI-ULM.DE
Benedikt Jutz BENEDIKT.JUTZ@UNI-ULM.DE
Mario Schmautz MARIO.SCHMAUTZ@ALUMNI.UNI-ULM.DE
Maximilian Borowiec MAXIMILIAN.BOROWIEC@ALUMNI.UNI-ULM.DE
Susanne Biundo SUSANNE.BIUNDO@UNI-ULM.DE
Institute of Artificial Intelligence, Ulm University
89081 Ulm, Germany

Pascal Bercher PASCAL.BERCHER@ANU.EDU.AU
School of Computing, The Australian National University
Canberra, Australia

Abstract

HTN planning requires the decomposition of compound tasks into primitive and executable actions. In the currently most frequently used formalism, compound tasks lack explicit preconditions and effects. Those are, however, useful, e.g., for pruning techniques, heuristics, or the comprehension of domains. Previously, we introduced and formalized different kinds of inferred preconditions and effects of compound tasks based on their decomposition methods together with a complexity analysis. In this paper, we present an empirical evaluation of computing these inferred preconditions and effects using the IPC benchmark sets. Specifically, we analyze their frequency of occurrence and compare the performance of an approximation to the exact preconditions and effects. Our goal is to provide a comprehensive overview of the proposed techniques, enabling researchers to determine the extent to which they can be utilized in their given application.

1. Introduction

Hierarchical Task Network (HTN) Planning is a subfield of Automated Planning, with the overarching goal of decomposing complex tasks into a sequence of primitive and executable actions (Erol, Hendler, & Nau, 1996; Ghallab, Nau, & Traverso, 2004; Bercher, Alford, & Höller, 2019). While primitive actions are explicitly defined by preconditions and effects that describe their interaction with states, compound tasks lack such explicit definitions in the currently most frequently used formalism and HDDL (Höller, Behnke, Bercher, Biundo, Fiorino, Pellier, & Alford, 2020a), the domain description language of the International Planning Competition (IPC) in 2020 and 2023 (Taitler, Alford, Espasa, Behnke, Fišer, Gimelfarb, Pommerening, Sanner, Scala, Schreiber, Segovia-Aguas, & Seipp, 2024). This makes their influence and dependencies on states rather opaque. To shed more light on their state implications, Olz, Biundo, and Bercher (2021) proposed inferred preconditions and effects for compound tasks in total-order domains based on the actions in their possi-

ble refinements, as opposed to hand-crafted preconditions and effects provided by domain engineers (Biundo & Schattenberg, 2001; Bercher, Höller, Behnke, & Biundo, 2016).

The proposed inferred preconditions and effects have already proven useful in diverse applications, ranging from assistance in the process of writing or understanding planning domains (Olz, Wierzba, Bercher, & Lindner, 2021) to generally improving the scalability and efficiency of HTN planning systems. For the latter, a total-order HTN planning heuristic based on Integer Linear Programming (Olz, Lodemann, & Bercher, 2024) utilizes inferred negative effects to restrict the set of possible solutions. Moreover, a look-ahead technique (Olz & Bercher, 2023b) exploits preconditions and effects to successfully prune the search space, leading to winning all total-order tracks at the latest IPC in 2023 (Olz, Höller, & Bercher, 2023; Taitler et al., 2024). Closely related, a version of inferred preconditions and effects generalized to partial-order domains (Olz & Bercher, 2022) has been used to convert partial-order problems into total-order ones, enabling the use of total-order planning systems to solve (some of) these problems, which currently outperform the general partial-order planners (Wu, Olz, Lin, & Bercher, 2023).

Other systems also already incorporate related or similar information (Nau, Au, Ilghami, Kuter, Murdock, Wu, & Yaman, 2003; Waisbrot, Kuter, & Könik, 2008; de Silva, Sardina, & Padgham, 2009; Bit-Monnot, Smith, & Do, 2016; Goldman & Kuter, 2019; Magnaguagno, Meneguzzi, & de Silva, 2021; Schreiber, 2021), some of which could potentially be further improved by integrating the preconditions and effects considered here. Additionally, this inferred information might be used to represent HTN planning solutions on an abstract level (Marthi, Russell, & Wolfe, 2007; de Silva et al., 2009), or closely related, to decide whether a sequence of (abstract) tasks can or can not be refined into a solution without finding a concrete plan (Yang, 1990). Those questions are interesting to the field for a long time as can also be seen by investigations on the so-called Upward and Downward properties (Tenenberg, 1988). However, little results exist so far.

Despite the benefits and existing approaches to *exploit* the proposed preconditions and effects, the algorithms for *computing* them are currently hidden in computational complexity proofs, and their concrete outcomes on available domains are not reported. For example, it is unclear how many preconditions and effects can be found and whether significant differences exist across different domains. These limitations make their application challenging.

This article complements the previous work by Olz et al. (2021) by making the algorithms for computing inferred preconditions and effects more accessible through the provision of pseudocode. Our main contribution is a comprehensive evaluation of the proposed techniques to examine their runtime, report the occurrence frequency of the inferred preconditions and effects, and assess the accuracy of an approximation on concrete domains. Our results aim to help researchers decide which of the proposed techniques, and to what extent they may be beneficial for their specific use cases beyond the already existing ones.

2. Theoretical Background

We begin with the theoretical basics, i.e., we introduce totally ordered (t.o.) HTN planning and the required definitions of preconditions and effects of compound tasks before we state algorithms to compute them.

2.1 HTN Planning Formalism

Our used formalism is based on the one by Geier and Bercher (2011), adapted by Behnke, Höller, and Biundo (2018) to totally ordered domains and problems, respectively.

As a basis, we introduce a *STRIPS planning domain* $\mathcal{D} = (F, A)$, which consists of a finite set of facts F and actions A. An action $a = (prec, add, del) \in A$ is a tuple consisting of its *preconditions* $prec(a) \subseteq F$ and its *add* and *delete effects*: $add(a), del(a) \subseteq F$. If it holds $prec(a) \subseteq s$ for an action $a \in A$ and a state $s \in 2^F$, then we say that a is applicable in s. In this case a can be applied to s producing the successor state $s' = (s \setminus del(a)) \cup add(a)$. We also say the application *results* in s'. 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 $0 \leq i \leq n$ a_i is applicable in s_i, where s_{i+1} results from applying a_i in s_i. A *STRIPS planning problem* $\Pi = (\mathcal{D}, s_I, g)$ additionally consists of an *initial state* $s_I \in 2^F$ and a *goal description* $g \subseteq F$, where the latter represents a set of goal states s with $s \supseteq g$. 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.

Now, we can define a *t.o. HTN planning domain* $\mathcal{D} = (F, A, C, M)$. It contains the sets F and A as before but the actions A are also called *primitive tasks* now. The reason is that we additionally consider the set C of *compound tasks*, and we define $A \cup C =: T$. Compound tasks represent sequences of primitive and further compound tasks, i.e., *t.o. task networks* tn, which are (possibly empty) finite sequences of tasks $\bar{t} = \langle t_0 \dots t_n \rangle \in T^*$. The set of *decomposition methods* $M \subseteq C \times T^*$ defines how exactly compound tasks can be refined, i.e., 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 (possibly empty) sequence of methods transforming tn into tn'. We call tn' a refinement of tn. In contrast to a STRIPS planning problem, a *t.o. HTN planning problem* $\Pi = (\mathcal{D}, s_I, tn_I, g)$ also contains an *initial task network* $tn_I \in T^*$, which needs to be refined. So, 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 primitive, applicable in s_I, and results in a goal state.

2.2 Preconditions and Effects of Compound Tasks

Now, we briefly recap the necessary definitions of preconditions and effects of compound tasks, taken from the publication we build upon (Olz et al., 2021).

First, for each compound task c, we need to define the set of states for which there exists an executable refinement (i.e., the application in these states is possible) as well as the set of states in which the execution of such a refinement can result:

Definition 1 (Executability-Enabling and 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.*

- *$E(c) \subseteq 2^F$ is the set of all states $\tilde{s} \in 2^F$ for which there exists a sequence of actions $\bar{a}$, such that $c \rightarrow \bar{a}$ and $\bar{a}$ is applicable in $\tilde{s}$.*
- *$R_s(c) \subseteq 2^F$ is the set of all states into which the execution of refinements of c 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'.*

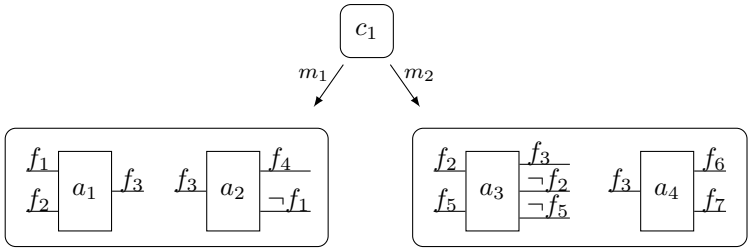


Figure 1: A compound task c_1 and its decomposition methods m_1 and m_2. Rectangles are primitive tasks with preconditions on the left-hand and effects on the right-hand side.

When we talk about effects of compound tasks we classify them according to three properties. We illustrate them using the compound task c_1 in Figure 1, which has two refinements (using either m_1 or m_2) with two primitive tasks each.

First, sometimes we are given a concrete state in which a compound task (or more precisely, its refinements) is executed (e.g., during search), but sometimes we want to know the preconditions and effects independent of a concrete state (e.g., as being part of the model). Therefore, there are state-dependent and state-independent effects. The latter are defined based on the former by considering all executability-enabling states.

Second, we differentiate between postconditions and effects. Postconditions can hold after the execution of a task just because they hold initially and were not changed, whereas effects are facts that were explicitly added (or deleted) by a task in the refinement and did not (did) hold in advance. For example, executing $\langle a_1, a_2 \rangle$ from Figure 1 in the state $s = \{f_1, f_2\}$ results in the state $\{f_2, f_3, f_4\}$. The facts f_3 and f_4 are explicitly added, so they are considered to be effects, whereas f_2 held before, so it is a postcondition.[1]

Lastly, we distinguish between possible and guaranteed effects. The former were added (or deleted) just by *some* refinements whereas the latter hold for *all* executable refinements. In Figure 1, the facts f_3, f_4, f_6 and f_7 are possible positive effects of c_1, f_1, f_2, and f_5 are possible negative effects. Since only f_3 is added by both refinements, it is the only guaranteed effect.

Based on these properties, we summarize only the most important definitions for this article and refer to Olz et al. (2021) for the remaining ones, which can also be easily inferred from the definitions provided.

Definition 2. *Let $\mathcal{D} = (F, A, C, M)$ be a domain, $c \in C$ a compound task, and $s \in 2^F$ a state. We define*

- $post_s^+(c) := \bigcap_{s' \in R_s(c)} s'$ *(state-dependent postconditions)*
  $post_s^-(c) := F \setminus \bigcup_{s' \in R_s(c)} s'$
- $post_*^+(c) := \bigcap_{s \in E(c)} post_s^+(c)$ *(state-independent postconditions)*
  $post_*^-(c) := \bigcap_{s \in E(c)} post_s^-(c)$
- $eff_*^+(c) := (\bigcap_{s \in E(c)} post_s^+(c)) \setminus (\bigcap_{s \in E(c)} s)$ *(state-independent effects)*
  $eff_*^-(c) := \bigcap_{s \in E(c)} post_s^-(c)$

1. There are some pitfalls concerning the difference between effects and postconditions, which we further illustrate with examples in the appendix.

- $\mathit{poss\text{-}post}_s^+(c) := \bigcup_{s' \in R_s(c)} s'$ (possible state-dependent postconditions)
  $\mathit{poss\text{-}post}_s^-(c) := \bigcup_{s' \in R_s(c)} (F \setminus s')$
- $\mathit{poss\text{-}eff}_*^+(c) := \bigcup_{s \in E(c)} (\mathit{poss\text{-}post}_s^+(c) \setminus s)$ (possible state-independent effects)
  $\mathit{poss\text{-}eff}_*^-(c) := \bigcup_{s \in E(c)} (\mathit{poss\text{-}post}_s^-(c) \cap s)$

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

The state-dependent postconditions are defined by iterating over all resulting states. The state-independent ones can be obtained by additionally considering all executability-enabling states. Depending on whether we want possible or guaranteed ones the intersection or union is taken. To obtain positive effects instead of postconditions we need to subtract the facts that held before execution. Note that there is an asymmetry between positive and negative guaranteed effects. This is due to the absence of negative preconditions. For this reason, all supersets of a state $s \in E(c)$ are also executability-enabling. Thus, to define the negative effects, we do not need to check that the facts held before since such a state always exist if there is at least one executability-enabling state. Therefore, the guaranteed negative effects are identical to the guaranteed negative postconditions.

Besides effects, we can also define preconditions of a compound task. Intuitively, preconditions are facts that are needed by every executable refinement to be executable. For c_1 in Figure 1 this would be f_2. So a precondition is the set of facts that are contained in every executability-enabling state of a compound task.

Definition 3 (Mandatory Preconditions)**.** *Let* $\mathcal{D} = (F, A, C, M)$ *be a t.o. HTN planning domain and* $c \in C$ *a compound task. Then, the set of* mandatory preconditions *of* c *is* $\mathit{prec}(c) := \bigcap_{s \in E(c)} s$ *if* $E(c) \neq \emptyset$ *and* $\mathit{prec}(c) := \mathit{undef}$ *otherwise.*

2.2.1 Computational Complexity

When developing algorithms it is always a good start to know the computational complexity of the underlying problem since it determines worst-case estimates and (partially) rules out the existence of certain algorithms. Thus, we now repeat some complexity results by Olz et al. (2021) concerning the inference of preconditions and effects of compound tasks before we state the algorithms. It turned out that it is always as hard as solving a planning problem, which in turn depends on the structure of the hierarchy.[2] Planning problems are referred to as *acyclic* when the number of possible decompositions is finite, meaning that there are no recursive methods involved (the respective plan existence problem is **PSPACE**-complete (Alford, Bercher, & Aha, 2015)). A problem is considered *regular* if, in both the initial task network and all the task networks of methods, there is only one compound task, if any, and it must always be the last one (**PSPACE**-complete (Erol et al., 1996)). They can be generalized to *tail-recursive* problems (**PSPACE**-complete (Alford et al., 2015)). Lastly, arbitrary problems do not have any restrictions on the hierarchy (**EXPTIME**-complete (Alford et al., 2015)).

Theorem 1. *Let* $c \in C$ *be a compound task,* $f \in F$ *a fact, and* $X \in \{s, *\}$*, where* $s \in 2^F$*. Deciding the following problems:*

2. Strictly speaking, this is only true for problems with deterministic complexity classes X (for which it holds $X = coX$) since the proof is partially based on the complementary problem.

- $f \in prec(c)$
- $f \in eff_X^{+/-}(c)$
- $f \in poss\text{-}eff_X^{+/-}(c)$

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

2.2.2 Precondition-Relaxations

For many practical applications, solving such **PSPACE**-complete or even **EXPTIME**-complete problems as part of a preprocessing step may not be a viable option due to the inherent complexity of the actual problem. Therefore, Olz et al. (2021) introduced a problem relaxation which admits the inference of preconditions and effects in polynomial time. It is based on ignoring whether primitive tasks are executable in a refinement and thus just takes the appearance and order of actions into account.

Definition 4 (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 5 (Effects of Precondition-Relaxed Tasks)**.** *Let $\mathcal{D}$ be a planning domain and $c \in C$ a compound task. We define $eff_*^{\emptyset+}(c)$, $eff_*^{\emptyset-}(c)$, $poss\text{-}eff_*^{\emptyset+}(c)$ and $poss\text{-}eff_*^{\emptyset-}(c)$ as shorthand for $eff_*^{+}(c)$, $eff_*^{-}(c)$, $poss\text{-}eff_*^{+}(c)$ and $poss\text{-}eff_*^{-}(c)$ that are based on the precondition-relaxed variant of $\mathcal{D}$, respectively.*

For preconditions, the relaxation is slightly adapted formally since we can not reason about preconditions if they are gone. However, semantically it is the same as for effects.

Definition 6 (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 $f \in prec(a_i)$ and there does not exist an action a_j with $j < i$ and $f \in add(a_j)$, where $i, j \in \{0 \dots n\}$. Then the set of* executability-relaxed preconditions *of c is $prec^{\emptyset}(c) := \{f \in F \mid f$ is an executability-relaxed precondition of $c\}$.*

In order to better distinguish the different preconditions and effects verbally we will use the term *exact* preconditions and effects when we refer to Definitions 2 and 3 in contrast to the *relaxed* ones (implying precondition-relaxation or executability-relaxation).

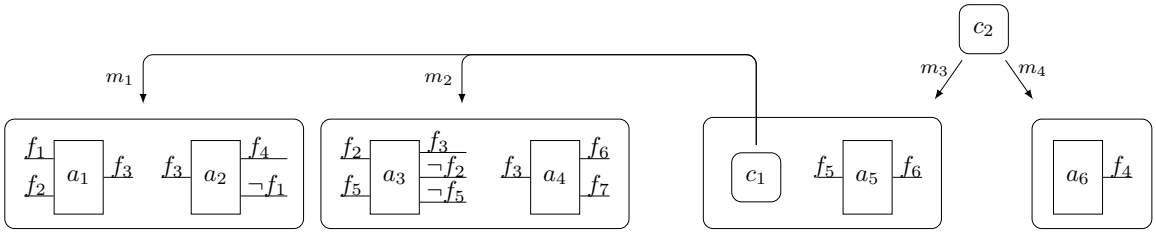


Figure 2: A compound task c_2 and its decomposition methods. The compound task c_1 is the same as before.

Consider the compound task c_2 in Figure 2 for an illustration. It has three refinements $\langle a_1, a_2, a_5 \rangle$, $\langle a_3, a_4, a_5 \rangle$, and $\langle a_6 \rangle$. The compound task c_1 is the one we considered before. We

can observe that $\langle a_3, a_4, a_5 \rangle$ will never be executable since a_3 deletes f_5, the precondition of a_5. So we disregard this refinement for exact effects. We can conclude that the guaranteed positive effects of c_2 is $\mathit{eff}^+_*(c_2) = \{f_4\}$, the possible ones are $\mathit{poss\text{-}eff}^+_*(c_2) = \{f_3, f_4, f_6\}$. Now, for the relaxed version we also take the refinement $\langle a_3, a_4, a_5 \rangle$ into consideration so that we end up with $\mathit{eff}^{\emptyset+}_*(c_2) = \emptyset$ and $\mathit{poss\text{-}eff}^{\emptyset+}_*(c_2) = \{f_3, f_4, f_6, f_7\}$. The set of (relaxed) preconditions of c_2 is empty because of m_4.

Despite the loss of information due to admitting the computation in polynomial time, the sets of relaxed preconditions and effects are still useful. The relaxed possible effects overestimate the exact ones, while the relaxed guaranteed effects and preconditions underestimate them. The intuition behind this is as follows: The relaxation increases the set of resulting states since more refinements are considered. Therefore, the set of facts common across all refinements is potentially reduced. On the other hand, the union of all facts in the resulting states is increased. Thus, we can state the following subset relations proven formally by Olz et al. (2021, Thm. 4 and 5):

Proposition 1. *Let $\mathcal{D} = (F, A, C, M)$ be a planning domain and $c \in C$ a compound task. Then it holds that:*

- $\mathit{poss\text{-}eff}^{\emptyset+}_*(c) \supseteq \mathit{poss\text{-}eff}^+_*(c)$ *and* $\mathit{poss\text{-}eff}^{\emptyset-}_*(c) \supseteq \mathit{poss\text{-}eff}^-_*(c)$,
- $\mathit{eff}^{\emptyset+}_*(c) \subseteq \mathit{post}^+_*(c)$ *and* $\mathit{eff}^{\emptyset-}_*(c) \subseteq \mathit{eff}^-_*(c)$,
- $\mathit{prec}^{\emptyset}(c) \subseteq \mathit{prec}(c)$

if they are not undefined.

It does *not* hold that $\mathit{eff}^{\emptyset+}(c) \subseteq \mathit{eff}^+(c)$ because the set $\mathit{eff}^{\emptyset+}(c)$ can contain facts that are also exact preconditions, which are specifically excluded for $\mathit{eff}^+_*(c)$. An example is given in the appendix. To make the sets of effects comparable, we define *adapted* (or *corrected*) *positive relaxed* effects as $\mathit{cor\text{-}eff}^{\emptyset+}_*(c) = \mathit{eff}^{\emptyset+}_*(c) \setminus \mathit{prec}(c)$, so that the relation $\mathit{cor\text{-}eff}^{\emptyset+}_*(c) \subseteq \mathit{eff}^+_*(c)$ holds. Note that this issue does not exist for negative effects since there are no negative preconditions.

In the evaluation section, we will assess the quality of the relaxation by calculating and reporting the size of the gap between these sets for many benchmark domains.

For practical applications, this is exactly the direction of approximation that is often needed, as can be seen, for instance, in the look-ahead technique by Olz and Bercher (2023b). Since the possible effects are an overapproximation, none of the effects is missing. For guaranteed effects, we can be sure that every relaxed effect is indeed correct, though some may be potentially missing.

2.2.3 Conjunctive Possible Effects

The possible effects of a compound task are often an overapproximation of the facts that any specific refinement actually produces, as they are derived by combining the effects of all possible refinements into a single set. To better distinguish the outcomes of different refinements, we previously introduced a more refined concept known as *conjunctive possible effects* (Olz & Bercher, 2023a). This approach considers sets of facts (of size k) that are added or deleted by a single refinement. Specifically, a set F_k is called a *positive-size-k* effect of c if and only if there exist a state $s \in E(c)$ and a successor state $s' \in R_s(c)$ such

that $F_k \subseteq s'$ and $s \cap F_k = \emptyset$. However, evaluating conjunctive possible effects is beyond the scope of this article. The time to compute the exact preconditions and effects for the conducted evaluation was already extensive (or hardly possible). The exact conjunctive possible effects are even more expensive to compute.

2.3 Related Work on Inference

There has been prior research focused on inferring preconditions and effects in HTN planning and related frameworks.

Tsuneto, Hendler, and Nau (1998) consider a lifted HTN planning formalism, where methods have different kinds of constraints predefined by domain modelers. From these they automatically extract "external conditions", which are state constraints and can be understood as method preconditions that need to be made true by other parts of the plan. According to the authors their algorithm to compute the external conditions is not complete and only described in prose.

Similar to our concept of resulting states ($R_s(c)$), Marthi et al. (2007) propose the *exact description* of a compound task. Since they are hard to compute, they define approximations through supersets (*complete descriptions*) and subsets (*sound descriptions*) and describe their potential applications in planning algorithms. However, they do not present procedures to create or verify these descriptions, leaving room for future research on utilizing our inferred preconditions and effects in this context.

Clement, Durfee, and Barrett (2007) introduce "summary information" within a kind of non-recursive partial-order HTN formalism with extensions for time and resource management, aiming to coordinate multi-agent plans at abstract levels. These summaries include temporal and modal information and are defined via logical formulae. To maintain an efficient computation, they focus on immediate subplans rather than the full hierarchy.

Extending this, de Silva, Sardina, and Padgham (2016) define summary information for a Belief-Desire-Intention (BDI) agent programming language (de Silva, Meneguzzi, & Logan, 2020), generalizing lifted totally ordered HTN problems. Their polynomial-time inference algorithms are limited to acyclic domains and remain incomplete, as supported by our complexity analysis. These algorithms, based on executable refinements like our exact ones, do not fully characterize the subset of inferred preconditions and effects, with examples illustrating their limitations.

Closer to our formalism are two approaches that compute method and task preconditions similarly to our executability-relaxed preconditions but in a lifted manner. Both approaches were developed simultaneously and independently of the work by Olz et al. (2021).

The lifted total-order planner HyperTensioN (Magnaguagno et al., 2021) computes these preconditions using a fixed-point algorithm during their preprocessing "pullup" phase, employing mentioned effects (de Silva et al., 2016). Similarly, the SAT-based planner Lilotane (Schreiber, Pellier, & Fiorino, 2019; Schreiber, 2021) "mines" method and task preconditions in polynomial time by recursively considering each method once. If a method is encountered twice due to cycles, the preconditions are approximated. These approaches are not directly comparable to ours as they operate on a lifted model. Nonetheless, we compare them empirically to our inferred preconditions to some extent in the evaluation section. The lifted approach has the advantage of being more general and compact and the results

might be usable across multiple domain instances. However, this generality can also lead to fewer specific decisions being made.

More loosely related again, there is also work on learning method preconditions from user-provided training data (Ilghami, Nau, Muñoz-Avila, & Aha, 2002; Ilghami, Muñoz-Avila, Nau, & Aha, 2005), as well as from decomposition trees with partially observed intermediate states (Zhuo, Hu, Hogg, Yang, & Munoz-Avila, 2009).

2.4 Formalisms Incorporating Preconditions and Effects

The whole article is based on the assumption that the preconditions and effects of compound tasks are not explicitly stated in the domain model. This is currently the predominant standard, as reflected in HDDL (Höller et al., 2020a), the official domain modeling language for HTN planning used in the 2020 and 2023 IPC on hierarchical planning (Behnke, Höller, Bercher, Biundo, Pellier, Fiorino, & Alford, 2019; Taitler et al., 2024). Conversely, alternative formalisms do specify preconditions and effects for compound tasks, such as the hybrid planning formalisms proposed by Kambhampati, Mali, and Srivastava (1998), Biundo and Schattenberg (2001), or Bercher, Lin, and Alford (2022). Additionally, planning systems like SHOP2 allow method preconditions (Nau et al., 2003). Typically, in these formalisms, the domain modeler defines the preconditions and effects of compound tasks, which come with varying semantics. Bercher et al. (2016, 2022) provide a comprehensive review of these. Nonetheless, it remains the responsibility of the domain modeler to ensure that the specified preconditions and effects meet all desired properties, which can be challenging (Bercher et al., 2016). Our proposed inference algorithms actually *verify* whether a fact is a precondition or effect and are turned to inference algorithms by testing all possible candidates. Consequently, even if there exist predefined preconditions or effects, our algorithms can be valuable for verifying and correcting them, or identifying additional ones.

3. Inference Algorithms

In this section, we present algorithms to compute the previously defined preconditions and effects of compound tasks. Olz et al. (2021) described all procedures in their proofs of the computational complexity results purely textual. Thus, the basic procedures are not new, but we turned them into pseudocode now to make them directly accessible. As a new contribution, we show how to speed up the exact inference procedures by mainly exploiting the relaxed preconditions and effects to save unnecessary calculation.

We begin with possible effects followed by guaranteed ones, then we consider preconditions. Afterwards, the precondition-relaxed versions are considered and illustrated with an example. We conclude with ideas on how to improve the exact inference procedures.

3.1 Effects and Postconditions of Compound Tasks

We begin with calculating *possible state-independent effects* of a compound task $c \in C$. Intuitively, a fact $f \in F$ is such an effect if there exists a state in which f does not hold and a refinement of c such that f holds after executing the refinement in that state. This is the case if the planning problem $\Pi = (\mathcal{D}, s_I = F \setminus \{f\}, tn_I = \langle c \rangle, \{f\})$ is solvable (Olz et al., 2021, Thm. 1).

Analogously, f is a *postcondition* if we alter the initial state to $s_I = F$, meaning that it is possible that f holds after decomposing c, even if it was not generated but held already. As we do not consider negative preconditions we cover every possible initial state by using the state that contains all facts in F, i.e., setting all facts to true.

If we want to calculate *state-dependent* postconditions w.r.t. some state s, then we set $s_I = s$ (Olz et al., 2021, Cor. 1).

In order to compute *negative* effects we can make use of auxiliary facts, which represent the negation of the original ones (Olz et al., 2021, Cor. 2). Therefore, we add for a fact f a new fact *not-f*, which encodes the negation of f. This idea is well known from compiling away negative preconditions (Gazen & Knoblock, 1997). Actions are modified so that all actions that add f then also delete *not-f*, and actions deleting f also add *not-f*. We will refer to this process of adapting the domain by the procedure called `AddNegatedFact`$(\mathcal{D}, f)$.

Algorithm 1 summarizes the previous explanations to compute possible state-independent negative effects.

Algorithm 1 Computes all possible state-independent negative effects

Input: $\mathcal{D} = (F, A, C, M)$, a planning domain, and $c \in C$, a compound task.
Output: $\textit{poss-eff}^-_*(c)$

1: **if** $\Pi = (\mathcal{D}, F, \langle c \rangle, \emptyset)$ is unsolvable **then**
2: ⌊ **return** $\textit{poss-eff}^-_*(c) = \textit{undef}$
3: $\textit{poss-eff}^-_*(c) = \emptyset$
4: **for all** $f \in F$ **do**
5: ⌊ $(\mathcal{D}', \textit{not-f}) =$ `AddNegatedFact(`$\mathcal{D}$`,`f`)`
6: ⌊ **if** $\Pi = (\mathcal{D}', F, \langle c \rangle, \{\textit{not-f}\})$ is solvable **then** ▷ $\textit{not-f} \notin F$
7: ⌊ ⌊ $\textit{poss-eff}^-_*(c) = \textit{poss-eff}^-_*(c) \cup \{f\}$
8: **return** $\textit{poss-eff}^-_*(c)$

Theorem 2. *Algorithm 1 is sound and complete, i.e., it returns exactly the set* $\textit{poss-eff}^-_*(c)$ *of possible negative effects of a compound task c.*

Proof. Let c be a compound task and $\textit{poss-eff}^-_*(c)$ its possible negative effects. Let $\textit{out} \subseteq F$ be the set of effects calculated and returned by Algorithm 1. We show that $\textit{poss-eff}^-_*(c) = \textit{out}$. The case $\textit{poss-eff}^-_*(c) = \textit{undef}$ is clearly covered by the algorithm, so we assume that $\textit{poss-eff}^-_*(c) \neq \textit{undef}$.

"$\subseteq$": Suppose there is a fact $f \in F$ with $f \in \textit{poss-eff}^-_*(c)$ but $f \notin \textit{out}$. Then, by definition, there is a state $s \in E(c)$ and a state $s_r \in R_s(c)$ such that $f \in s$ but $f \notin s_r$. Let $\bar{a}$ be the corresponding refinement. After modifying the domain to $\mathcal{D}'$ by adding the auxiliary variable *not-f* (using `AddNegatedFact(`$\mathcal{D}$`,`f`)`) and applying $\bar{a}$ with updated actions in s we end up in the state $s'_r = s_r \cup \{\textit{not-f}\}$. Since $F \supseteq s$ and there are no negative preconditions the planning problem $\Pi = (\mathcal{D}', F, \langle c \rangle, \{\textit{not-f}\})$ must be solvable as otherwise s'_r can not exist. This contradicts $f \notin \textit{out}$, so it must hold $f \in \textit{out}$.

"$\supseteq$": Suppose it holds $f \in \textit{out}$. Then $\Pi = (\mathcal{D}', F, \langle c \rangle, \{\textit{not-f}\})$ is solvable. Let s_g be one of the goal states. By construction, it holds $\textit{not-f} \in s_g$ and $f \notin s_g$. Moreover it holds $f \in F$, $F \in E(c)$, $s_g \setminus \{\textit{not-f}\} \in R_F(c)$ (w.r.t. $\mathcal{D}$, in which $\{\textit{not-f}\}$ does not exist). Thus, we can conclude $f \in \textit{poss-eff}^-_*(c)$. □

In order to compute *guaranteed* state-independent positive effects we have to ensure that for a possible effect f there is no refinement after whose execution the fact does not hold. To do so we make use of the negated fact and construct two planning problems as shown in Algorithm 2 (lines 6 and 7). If f holds in the initial state, then it must not be deleted in the end. Otherwise, if *not-f* holds initially, then f must be added eventually. So there may not exist a solution in which *not-f* holds at the end, i.e., we check whether the two problems are unsolvable (line 8). Note that we can also capture this with a single planning problem instead of two, by additionally introducing a new compound task c', which has two methods m_1 and m_2, where $m_1 = (c', \langle(\emptyset, \{f\}, \emptyset)c\rangle)$ and $m_2 = (c', \langle(\emptyset, \{\textit{not-f}\}, \emptyset)c\rangle)$. The two methods of c' simulate the two initial states of Π'_1 and Π'_2, respectively, i.e., if m_1 gets chosen we end up with Π'_1 and Π'_2 otherwise. So again one needs to check whether $\Pi' = (\mathcal{D}', F \setminus \{f\}, \langle c'\rangle, \{\textit{not-f}\})$ is unsolvable (Olz et al., 2021, Thm. 2).

The adaptions to compute negative effects or postconditions are straightforward, so we restrict ourselves to Algorithm 2, the computation of guaranteed state-independent positive effects, here.

Algorithm 2 Computes all guaranteed state-independent positive effects

Input: $\mathcal{D} = (F, A, C, M)$, a planning domain, and $c \in C$, a compound task.
Output: $\mathit{eff}^+_*(c)$

1: **if** $\mathit{poss\text{-}eff}^+_*(c) = \mathit{undef}$ **then**
2: ⌊ **return** *undef*
3: $\mathit{eff}^+_*(c) = \emptyset$
4: **for all** $f \in \mathit{poss\text{-}eff}^+_*(c)$ **do**
5: ⌊ $(\mathcal{D}', \textit{not-f}) =$ `AddNegatedFact(`$\mathcal{D}$`,`f`)`
6: ⌊ $\Pi'_1 = (\mathcal{D}', F, \langle c\rangle, \{\textit{not-f}\})$
7: ⌊ $\Pi'_2 = (\mathcal{D}', F \setminus \{f\} \cup \{\textit{not-f}\}, \langle c\rangle, \{\textit{not-f}\})$
8: ⌊ **if** Π'_1 and Π'_2 are not solvable **then**
9: ⌊ ⌊ $\mathit{eff}^+_*(c) = \mathit{eff}^+_*(c) \cup \{f\}$
10: **return** $\mathit{eff}^+_*(c)$

Theorem 3. *Algorithm 2 is sound and complete, i.e., it returns exactly the set* $\mathit{eff}^+_*(c)$ *of guaranteed positive effects of a compound task* c.

Proof. Let c be a compound task and $\mathit{eff}^+_*(c)$ of guaranteed positive effects. Let $\mathit{out} \subseteq F$ be the set of effects calculated and returned by Algorithm 2. We show that $\mathit{eff}^+_*(c) = \mathit{out}$. Again, the case $\mathit{eff}^+_*(c) = \mathit{undef}$ is covered in the beginning of the algorithm, so we assume that $\mathit{eff}^+_*(c) \neq \mathit{undef}$.

"$\subseteq$": Suppose there is a fact $f \in F$ with $f \in \mathit{eff}^+_*(c)$. Then it is also a possible effect with $f \in \mathit{poss\text{-}eff}^+_*(c)$, $f \notin \bigcap_{s \in E(c)} s$, and $f \in \bigcap_{s \in E(c)} \bigcap_{s' \in R_s(c)} s'$. This means that there are no two states $s_2 \in E(c)$, $s'_2 = R_{s_2}(c)$ with $f \notin s'_2$. This implies that Π'_1 and Π'_2, as defined in the algorithm, are not solvable because by construction for all goal states of the two problems with $s_g \supseteq \{\textit{not-f}\}$ it holds $f \notin s_g$. So $f \in \mathit{out}$.

"$\supseteq$": Suppose it holds $f \in \mathit{out}$. Since Π'_1 is unsolvable we know that there is no executable refinement of c that deletes f. Moreover, since Π'_2 is unsolvable we also know that f is always added if it is not true beforehand. So there is no resulting state in which f

does not hold. Lastly, since $f \in \mathit{poss\text{-}eff}^+_*(c)$ it (f) is contained in at least one resulting state. Additionally, it holds $f \notin \bigcap_{s \in E(c)} s$, which means it is not just a positive postcondition but explicitly added. Thus, we conclude $f \in \mathit{eff}^+_*(c)$. □

3.2 Preconditions of Compound Tasks

Algorithm 3 Computes all mandatory preconditions of a compound task

Input: $\mathcal{D} = (F, A, C, M)$, a planning domain, and $c \in C$, a compound task.
Output: $\mathit{prec}(c)$

```
if Π = (D, F, ⟨c⟩, ∅) is unsolvable then
    return prec(c) = undef
prec(c) = ∅
for all f ∈ F do
    Π' = (D, F \ {f}, ⟨c⟩, ∅)
    if Π' is not solvable then
        prec(c) = prec(c) ∪ {f}
return prec(c)
```

The computation of mandatory preconditions of a compound task c is described in Algorithm 3. We just need to check for every fact f whether there does not exist an executable refinement in the state $s = F \setminus \{f\}$ (Olz et al., 2021, Thm. 3).

The correctness of Algorithm 3 is straightforward.

3.3 Precondition-Relaxed Effects

Computing the exact preconditions and effects of a compound task may take too much time in specific applications as can be seen from the algorithms and the underlying computation complexity of the problem, which we reported in the previous section. Therefore, we introduced precondition-relaxation to get an under- and over-approximation, respectively, of these that can be computed in polynomial time. In this subsection, we present the respective algorithms, which were already described purely textually in 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 et al. (2021).

Before we describe the algorithms we would like to clarify that we always assume that there are no compound tasks or decomposition methods with only non-terminating refinements. By this we mean that every task or method can be refined into a (possibly empty) primitive action sequence. Without this assumption the following algorithms can lead to wrong results. Note, however, that this assumption is very natural (because such tasks would only distract the planner) and easy (in P) to check beforehand, which is also done and ruled out by our used planner in the evaluation. More specifically, the deployed grounder (Behnke, Höller, Schmid, Bercher, & Biundo, 2020) removes compound tasks and methods without primitive refinements.

Moreover, w.l.o.g., we assume for the remainder of this article that for all actions $a \in A$ it holds $\mathit{prec}(a) \cap \mathit{add}(a) = \emptyset$ and $\mathit{add}(a) \cap \mathit{del}(a) = \emptyset$.

Algorithm 5 is the main procedure to compute precondition-relaxed effects. We consider one fact $f \in F$ after another and curtail the domain according to several subroutines, listed in Algorithm 4.

- We keep only primitive actions that add or delete f as all others are irrelevant. Therefore, the function RestrictToEffects$(\mathcal{D}, f)$ 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 $A' = \{(\emptyset, add(a) \cap \{f\}, del(a) \cap \{f\}) \mid a \in A\} \setminus \{(\emptyset, \emptyset, \emptyset)\}$. The set M' is obtained from M by restricting the task networks to tasks from $A' \cup C$ instead of $A \cup C$. We could actually also keep the other facts in the domain but the algorithms are clearer and easier to grasp if we remove unnecessary information.

- A main observation is that it is always the last primitive task in a refinement that is responsible for determining whether a fact holds after execution or not (recall that we removed all facts except for the one under consideration and actions that do not affect it). In order to identify such actions we first need to know whether compound tasks can be refined into empty task networks. Therefore, the function EmptyRefinements$(\mathcal{D}')$ is called on the restricted domain and computes the set of compound tasks admitting an empty refinement, $C_\varepsilon \subseteq C$ by propagating the information in an upwards manner starting with methods with empty task networks. The while-loop terminates after at most $|M|$ iterations. 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 guaranteed (positive or negative) precondition-relaxed effect. (Lines 7-9 and 15 are not necessary since the stopping condition is that the set C_ε has not changed. However, they might have a positive impact on the runtime.)

- As mentioned before, it is always the last primitive task responsible for adding or deleting f. Therefore, we can remove tasks preceding this task in all method's task networks as they are irrelevant. However, it is important to consider the case where a compound task in a task network can be refined into an empty refinement; in such cases, the preceding task also becomes relevant. Thus, the function ShortenMethodsFromRight$(\mathcal{D}', C_\varepsilon)$ identifies this task for all decomposition methods and removes the tasks that are ordered before it, i.e., we examine every method's task network from right to left and cut off the remaining tasks after we either found a primitive task or a compound task that does *not* admit an empty refinement.

- 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, i.e., $M_R = \{m \in M \mid m = (c, tn)\} \cup \{m = (c', tn) \in M \mid \exists\, tn' : \langle c \rangle \to tn' \wedge c' \in tn'\}$.

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 et al. (2021) and the proof below. Furthermore, we can conclude that f is even a guaranteed positive precondition-relaxed effect if (1) it is a possible positive effect, (2) not a possible negative one, and c can not be refined into an

empty refinement. This is because, in the case of (2) or (3), there are refinements after the execution of which f is not added. The procedure for negative effects follows analogously.

Algorithm 4 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** $m = (c, tn = \langle t_1 \dots t_n \rangle) \in M$ **do**
5: **if** $tn = \varepsilon$ **and** $c \notin C_\varepsilon$ **then**
6: $C_\varepsilon = C_\varepsilon \cup \{c\}$
7: $M' = M' \setminus \{m\}$
8: **if** $\exists\, i : t_i \in A$ **then**
9: $M' = M' \setminus \{m\}$
10: **while** *setChanged* **do**
11: *setChanged* $=$ *false*
12: **for all** $m = (c, tn = \langle t_1 \dots t_n \rangle) \in M'$ **do**
13: **if** $c \notin C_\varepsilon$ **and** $\forall i : t_i \in C_\varepsilon$ **then**
14: $C_\varepsilon = C_\varepsilon \cup \{c\}$
15: $M' = M' \setminus \{m\}$
16: *setChanged* $=$ *true*
17: **return** C_ε

18: ▷ *Returns a domain, where all methods have task networks with at most one task not in C_ε (which then is the first one).* ◁
19: **function** ShortenMethodsFromRight($\mathcal{D}, C_\varepsilon$)
20: $M' = \emptyset$
21: **for all** $m = (c, tn = \langle t_1 \dots t_n \rangle) \in M$ **do**
22: $tn_2 = tn$
23: **for** $i = n \dots 1$ **do**
24: **if** $t_i \notin C_\varepsilon$ **then**
25: $tn_2 = \langle t_i \dots t_n \rangle$
26: **break**
27: $M' = M' \cup \{(c, tn_2)\}$
28: **return** $\mathcal{D}' = (F, A, C, M')$

Theorem 4. *Algorithm 5 is sound and complete, i.e., it returns exactly the sets of precondition-relaxed effects of all compound tasks.*

Proof. We only prove the case of positive effects. Therefore, let $c \in C$ be a compound task and $\textit{poss-eff}_*^{\emptyset+}(c), \textit{eff}_*^{\emptyset+}(c)$ its positive precondition-relaxed effects. Let $\textit{poss-out}(c) \subseteq F$ and $\textit{gua-out}(c) \subseteq F$ be the respective sets returned by Algorithm 5. We show that these respective sets are identical in both cases. We assume that $\mathcal{D}', C_\varepsilon, \mathcal{D}''$, and M_R are calculated and given according to lines 3-6.

We start with possible effects. Let $f \in \textit{poss-eff}_*^{\emptyset+}(c)$. Then there is a (not necessarily executable) primitive refinement $\bar{a} = \langle a_1 \dots a_n \rangle$ of c such that there exists an action $a_i \in \bar{a}$

Algorithm 5 Calculates the precondition-relaxed effects for all compound tasks

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

1: $\mathit{poss\text{-}eff}_*^{\emptyset+}(c) = \mathit{poss\text{-}eff}_*^{\emptyset-}(c) = \mathit{eff}_*^{\emptyset+}(c) = \mathit{eff}_*^{\emptyset-}(c) = \emptyset$ for all $c \in C$
2: **for all** $f \in F$ **do**
3: $\quad \mathcal{D}' = \textsc{RestrictToEffects}(\mathcal{D}, f)$
4: $\quad C_\varepsilon = \textsc{ComputeEmptyRefinements}(\mathcal{D}')$
5: $\quad \mathcal{D}'' = \textsc{ShortenMethodsFromRight}(\mathcal{D}', C_\varepsilon)$
6: $\quad M_R = \textsc{DecompositionReachability}(\mathcal{D}'')$
7: $\quad$ **for all** $c \in C$ **do**
8: $\quad\quad$ **if** $\exists\, (c', \langle t_1 \dots t_n \rangle) \in M_R(c) \wedge i \in \{1 \dots n\} : f \in \mathit{add}(t_i)$ **then**
9: $\quad\quad\quad \mathit{poss\text{-}eff}_*^{\emptyset+}(c) = \mathit{poss\text{-}eff}_*^{\emptyset+}(c) \cup \{f\}$
10: $\quad\quad$ **if** $\exists\, (c', \langle t_1 \dots t_n \rangle) \in M_R(c) \wedge i \in \{1 \dots n\} : f \in \mathit{del}(t_i)$ **then**
11: $\quad\quad\quad \mathit{poss\text{-}eff}_*^{\emptyset-}(c) = \mathit{poss\text{-}eff}_*^{\emptyset-}(c) \cup \{f\}$
12: $\quad\quad$ **if** $c \notin C_\varepsilon$ **then**
13: $\quad\quad\quad$ **if** $f \in \mathit{poss\text{-}eff}_*^{\emptyset+}(c) \wedge f \notin \mathit{poss\text{-}eff}_*^{\emptyset-}(c)$ **then**
14: $\quad\quad\quad\quad \mathit{eff}_*^{\emptyset+}(c) = \mathit{eff}_*^{\emptyset+}(c) \cup \{f\}$
15: $\quad\quad\quad$ **if** $f \notin \mathit{poss\text{-}eff}_*^{\emptyset+}(c) \wedge f \in \mathit{poss\text{-}eff}_*^{\emptyset-}(c)$ **then**
16: $\quad\quad\quad\quad \mathit{eff}_*^{\emptyset-}(c) = \mathit{eff}_*^{\emptyset-}(c) \cup \{f\}$
17: **return** $\mathit{poss\text{-}eff}_*^{\emptyset+}(c), \mathit{poss\text{-}eff}_*^{\emptyset-}(c), \mathit{eff}_*^{\emptyset+}(c), \mathit{eff}_*^{\emptyset-}(c)$ for all $c \in C$

with $f \in \mathit{add}(a)$ and none of the subsequent actions add or delete f. Let $m \in M$ be the method that inserted a_i into $\bar{a}$. Then the corresponding method m'' of $\mathcal{D}''$ must contain a_i and $m'' \in M_R(c)$. Suppose this is not the case. Since $f \in \mathit{add}(a_i)$, it holds $a_i \in A'$ (of $\mathcal{D}'$). So, a_i must have either been cut out from m by $\textsc{ShortenMethodsFromRight}(\mathcal{D}', C_\varepsilon)$ or m'' it is not reachable anymore because some compound task leading to it has been deleted from a method. If the former happened, then there must be an action a_j with $f \in \mathit{add}(a_j) \cup \mathit{del}(a_j)$ succeeding a_i in m and thus also in $\bar{a}$, which contradicts our assumption. So, a_i is not reachable anymore, since a compound task c_1 leading to m has been cut out. Suppose this happened in method m_1. Then, there must be a succeeding primitive action a_j adding or deleting f or a compound task c_2 without empty refinement in m_1. However, then a_j or a similar primitive action of the refinement of c_2 is also contained in $\bar{a}$ succeeding a_i, which again contradicts the assumption. Thus, $m'' \in M_R(c)$, $a_i \in m''$, and therefore $f \in \mathit{poss\text{-}out}(c)$. The opposite direction follows with similar arguments.

The correct calculation of the guaranteed effects is easy to see. Generally, there are three types of refinements of c: Either a fact f gets added, deleted, or does not appear in any of the involved actions' effects. In the latter case, c admits an empty refinement with respect to f. Therefore, it holds $f \in \mathit{eff}_*^{\emptyset+}(c)$ if and only if $f \in \mathit{poss\text{-}eff}_*^{\emptyset+}(c)$, $f \notin \mathit{poss\text{-}eff}_*^{\emptyset-}(c)$, and c does not admit an empty refinement. This is verified starting from line 12. □

3.3.1 Example

We illustrate Algorithm 5 with an example shown in Figure 3. A compound task c with its decomposition methods m_1 and m_2 is shown at the top. The methods of compound tasks

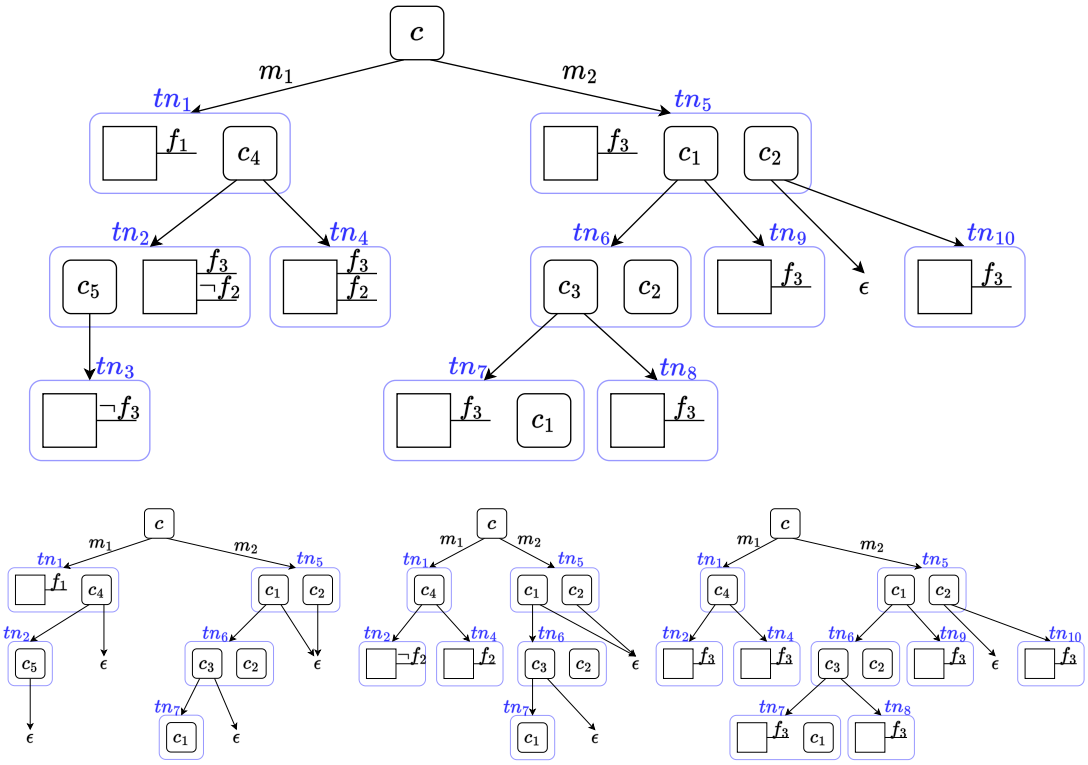


Figure 3: An example demonstrating the mechanisms of Algorithm 5, which computes precondition-relaxed effects. The decomposition hierarchy of a compound task c is depicted at the top. At the bottom one can see the restricted domain after fact f_1 (left), f_2 (middle), and f_3 (right) were considered. Rectangles with sharp corners depict primitive tasks. Tasks within a task network are ordered from left to right. As a result we get: $\mathit{poss\text{-}eff}_*^{\emptyset+}(c) = \{f_1, f_2, f_3\}$, $\mathit{poss\text{-}eff}_*^{\emptyset-}(c) = \{f_2\}$, $\mathit{eff}_*^{\emptyset+}(c) = \{f_3\}$, and $\mathit{eff}_*^{\emptyset-}(c) = \emptyset$.

within the task networks are also given directly. The hierarchy contains a cycle because task network tn_7 can be reached from compound task c_1 and does also contain c_1 again. The methods of c_1 and c_2 are given after tn_5, so we do not repeat them for tn_6 and tn_7. The rectangles with sharp corners are primitive tasks.

At the bottom left we can see the curtailed domain after considering f_1 in the outermost forall-step. Only the primitive task in tn_1 adds (or deletes) f_1, so all other primitive tasks were removed. EmptyRefinements($\mathcal{D}$) does not change anything. We can conclude that f_1 is a possible positive precondition-relaxed effect of c because tn_1 is still reachable via decomposition from c, and it contains a primitive task adding f_1. It is not a guaranteed (precondition-relaxed) effect since c can be refined into an empty task network using m_2.

Next, we examine f_2 (depicted in the middle). We consider the original domain – rather than the one on the left – and once again remove all irrelevant primitive tasks. Since the rightmost task in tn_2 is primitive, we truncate the task network at this point, resulting in the removal of c_5 and making tn_3 no longer reachable. Consequently, we conclude that f_2 is a possible precondition-relaxed positive and negative effect as it can be decomposed into

tn_2 or tn_4. Therefore, and for the same reasons as previously discussed, it cannot be a guaranteed (positive or negative) effect.

Finally, we consider f_3 (on the right side). Again, we keep only actions adding or deleting f_3 and cut task networks: We remove c_5 in tn_2 and in tn_5 we remove the leftmost (primitive) task since c_1 does not admit an empty refinement. Now, f_3 is a (guaranteed) positive precondition-relaxed effect of c because 1) there is a task network reachable from c with an action adding f_3 (e.g., tn_2), 2) there is no such task network with an action deleting it, and 3) c does not admit an empty refinement.

3.4 Executability-Relaxed Preconditions

Executability-relaxed preconditions can be computed similarly to effects, as shown in Algorithm 6. To ensure that every refinement of c contains a primitive action requiring f and no preceding action adds it, the function RestrictToEffects$(\mathcal{D}, f)$ is adapted into two new functions: RestrictToPrecsOnly$(\mathcal{D}, f)$ and RestrictToPrecsAndAdds$(\mathcal{D}, f)$. In the first case we keep only actions with $f \in prec(a)$ and in the latter if $f \in prec(a) \cup add(a)$. Additionally, methods are now curtailed from left to right since we want to check the first task in each refinement instead of the last. So, the function ShortenMethodsFromLeft$(\mathcal{D}'', C'_\varepsilon)$ mirrors ShortenMethodsFromRight$(\mathcal{D}', C_\varepsilon)$, with the loop in line 23 running from 1 to $|tn|$ instead. Then, f is an executability-relaxed precondition of c if no reachable method contains an action adding f, and c does not allow an empty refinement in the restricted domain where only actions requiring f as a precondition are considered.

Algorithm 6 Calculates the executability-relaxed preconditions.

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

1: $prec^\emptyset(c) = \emptyset$ for all $c \in C$
2: **for all** $f \in F$ **do**
3: $\quad \mathcal{D}' =$ RestrictToPrecsOnly$(\mathcal{D}, f)$
4: $\quad C_\varepsilon =$ ComputeEmptyRefinements$(\mathcal{D}')$
5: $\quad \mathcal{D}'' =$ RestrictToPrecsAndAdds$(\mathcal{D}, f)$
6: $\quad C'_\varepsilon =$ ComputeEmptyRefinements$(\mathcal{D}'')$
7: $\quad \mathcal{D}''' =$ ShortenMethodsFromLeft$(\mathcal{D}'', C'_\varepsilon)$
8: $\quad M_R =$ DecompositionReachability$(\mathcal{D}''')$
9: $\quad$ **for all** $c \in C$ **do**
10: $\quad\quad$ **if** $c \notin C_\varepsilon \wedge \nexists\, m = (c', tn = \langle t_1 \dots t_n \rangle) \in M_R(c) : \exists\, i : f \in add(t_i)$ **then**
11: $\quad\quad\quad prec^\emptyset(c) = prec^\emptyset(c) \cup \{f\}$

12: **return** $prec^\emptyset(c)$ for all $c \in C$

Theorem 5. *Algorithm 6 is sound and complete, i.e., it returns exactly the sets of executability-relaxed preconditions of all compound tasks.*

Proof Sketch. Let c be a compound task, $prec^\emptyset(c)$ its executability-relaxed preconditions, and the sets $\mathcal{D}', C_\varepsilon, \mathcal{D}'', C'_\varepsilon, \mathcal{D}'''$, and M_R as calculated by Algorithm 6. According to our

definition, it holds $f \in \mathit{prec}^{\emptyset}(c)$ 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 $i, j \in \{0 \dots n\}$. The first part is ensured by the conjunct $c \notin C_\varepsilon$ in line 10. With similar arguments as for the proof of Theorem 4 it holds that there is a refinement of c with an action a_j adding f and no preceding action a_i with $f \in \mathit{prec}(a_i)$ if and only if there is a method in $M_R(c)$ containing an action that adds f. So we can conclude that if and only if such a method does not exist in $M_R(c)$ (second conjunct in line 10) then no such refinement exists and f is an executability-relaxed precondition. □

4. Optimizations Concerning Exact Inference

Calculating possible positive or negative state-independent effects or postconditions can take quite a while as we have to solve at least one planning problem per pair of compound task and fact (in the way we do it). However, we can exploit the property of exact possible effects being precondition-relaxed possible effects (see Proposition 1) to speed up the calculation.

Because it holds $\mathit{poss\text{-}eff}_*^{\emptyset+}(c) \supseteq \mathit{poss\text{-}eff}_*^{+}(c)$ we can conclude that if a fact $f \notin \mathit{poss\text{-}eff}_*^{\emptyset+}(c)$ then also $f \notin \mathit{poss\text{-}eff}_*^{+}(c)$. So instead of considering all facts in F as being candidates for possible state-independent effects we just take the relaxed ones and therefore need to solve fewer planning problems, i.e. we improve Algorithm 1 in line 4. Since we can calculate the set $\mathit{poss\text{-}eff}_*^{\emptyset+}(c)$ in polynomial time this saves a lot of time as we will see in the evaluation section. Moreover, since the possible effects are taken as basis for the guaranteed ones, this does also affect the efficiency of their calculation.

We had not stated an analogous result for preconditions since we have not defined possible executability-relaxed preconditions so far as we had not seen any benefit at the time. However, now we found an application: We can use them analogously to speed up the inference of mandatory preconditions. So we define them just for this purpose here:

Definition 7 (Possible 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 a* possible executability-relaxed precondition *of c if and only if there exists a primitive refinement (i.e., ignoring executability) $\langle a_0 \dots a_n \rangle$ of c such that 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 8 (Possible Executability-Relaxed Preconditions)**.** *Let $\mathcal{D}$ be a planning domain and $c \in C$ a compound task. Then the set of* possible executability-relaxed preconditions *of c is $\mathit{poss\text{-}prec}^{\emptyset}(c) := \{f \in F \mid f$ is a possible executability-relaxed precondition of $c\}$.*

It is easy to see that $\mathit{prec}(c) \subseteq \mathit{poss\text{-}prec}^{\emptyset}(c)$ holds. Let $f \in \mathit{prec}(c)$. Since $\mathit{prec}(c) := \bigcap_{s \in E(c)} s$ (if $E(c) \neq \emptyset$), we know that there is a state $s \in E(c)$ and a refinement of c executable in s in which there is an action $a \in A$ with $f \in \mathit{prec}(a)$ and no other action adds f beforehand; otherwise, the refinement would also be executable in the state $s' = s \setminus f$, so $s' \in E(c)$, which contradicts $f \in \mathit{prec}(c)$.

We can calculate the possible preconditions using an adaptation of the algorithm to compute executability-relaxed preconditions, i.e., via Algorithm 6. We only need to ensure that there is a refinement in which a fact f is required due to a primitive task's preconditions before it is potentially added by another one. Therefore, we can skip lines 3 and 4 and

alter line 10 as follows: If $\exists\, m = (c', tn = \langle t_1 \dots t_n \rangle) \in M_R(c) : \exists\, i : f \in prec(t_i)$ then $f \in poss\text{-}prec^{\emptyset}(c)$. Note that the logic behind this is very similar to calculating possible effects, which is why we refrain from a formal proof of soundness and completeness. The only difference is that we restrict to actions containing f in preconditions and add effects instead of add and delete effects. Moreover, we traverse the task networks from left to right to check for the first occurrence of actions instead of the last ones.

To summarize, for calculating the exact preconditions and effects we make use of the following subset properties:

1. $poss\text{-}eff^+_*(c) \subseteq poss\text{-}eff^{\emptyset+}_*(c)$ and $poss\text{-}eff^-_*(c) \subseteq poss\text{-}eff^{\emptyset-}_*(c)$
2. $eff^+_*(c) \subseteq poss\text{-}eff^+_*(c)$ and $eff^-_*(c) \subseteq poss\text{-}eff^-_*(c)$
3. $prec(c) \subseteq poss\text{-}prec^{\emptyset}(c)$
4. $eff^+_*(c) \supseteq cor\text{-}eff^{\emptyset+}_*(c)$ and $eff^-_*(c) \supseteq eff^{\emptyset-}_*(c)$
5. $prec(c) \supseteq prec^{\emptyset}(c)$
6. $poss\text{-}eff^-_*(c) \cap eff^+_*(c) = \emptyset$ and $poss\text{-}eff^+_*(c) \cap eff^-_*(c) = \emptyset$
7. $eff^+_*(c) \cap prec(c) = \emptyset$.

We use 1.-3. to restrict the set of possible candidates in Algorithm 1 (line 4) and Algorithm 3 (line 4). Because of 4.-5., we know that we can just add the guaranteed relaxed preconditions and effects to the exact ones and do not need to solve the respective planning problems. Moreover, (6.-7.), we know that every possible negative effect can not be a guaranteed positive effect, every possible positive effect can not be a guaranteed negative effect, and every precondition is not a positive effect. The correctness follows from Proposition 1 and the explanations above.

We evaluate the impact on the runtime of exploiting the relaxed preconditions and effects for computing the exact ones later on. To make the comparison fairer we will not check all facts in the naive approach but only those that actually occur in at least one primitive action in a refinement of the analyzed compound task. They are called *mentioned* preconditions and effects (the latter are later denoted $mnt^+(c)$ and $mnt^-(c)$) (de Silva et al., 2016). It is easy to see that $poss\text{-}eff^{\emptyset+}_*(c) \subseteq mnt^+(c)$ and $poss\text{-}eff^{\emptyset-}_*(c) \subseteq mnt^-(c)$.

4.1 A Discussion on Possible Alternative Algorithms

Before we present the evaluation results, we briefly sketch and discuss an alternative approach for calculating inferred preconditions and effects, as suggested by one of our anonymous reviewers. While we do not provide precise algorithms, evaluations, or theoretical analyses here – these are beyond the scope of this article – we aim to outline the idea and assess its potential advantages and challenges.

The suggestion is based on the concept of cumulative preconditions and effects of STRIPS actions, as introduced by Haslum and Jonsson (2000) as well as Bäckström, Jonsson, and Jonsson (2012). Let $\bar{a} \in A^*$ be a sequence of actions and $a \in A$. The cumulative

preconditions and effects of a sequence of actions are defined recursively as follows:

$$\begin{aligned} prec(\langle\rangle) &= add(\langle\rangle) = del(\langle\rangle) = \emptyset \\ prec(\langle \bar{a} a \rangle) &= prec(\bar{a}) \cup (prec(a) \setminus add(\bar{a})) \\ add(\langle \bar{a} a \rangle) &= (add(\bar{a}) \setminus del(a)) \cup add(a) \\ del(\langle \bar{a} a \rangle) &= (del(\bar{a}) \setminus add(a)) \cup del(a) \end{aligned}$$

Moreover, $\langle \bar{a} a \rangle$ is executable if and only if $del(\bar{a}) \cap prec(a) = \emptyset$ and $\bar{a}$ is executable.

For each decomposition method, one could recursively compute the set of composed tasks corresponding to all executable sequences of primitive tasks generated by the method. However, upon closer examination, several challenges arise:

Consider *exact* preconditions and effects. Even in acyclic hierarchies, the number of executable action sequences can grow exponentially, and in general hierarchies, this number could be infinite. Efficiently combining intermediate concatenated action sequences would require substantial memory, potentially leading to an exponential storage requirement. Furthermore, calculating everything correctly in such cases would involve non-trivial procedures to manage the interactions between preconditions, add effects, and delete effects across multiple sequences. If successful, this approach could compute preconditions and effects for all facts simultaneously, avoiding the need to solve a planning problem for each fact individually. Additionally, it inherently answers the question whether an executable action sequence exists, potentially yielding a new planning algorithm. However, as shown in Theorem 1, deciding whether a fact is an inferred precondition or effect is as computationally hard as the plan existence problem itself. Thus, one could regard it unlikely that this method would significantly speed up the inference process – yet, one could try and compare empirical results.

For calculating relaxed preconditions and effects, this approach seems more promising. Ignoring executability simplifies the set manipulations and again allows all facts to be calculated simultaneously. However, this would still likely require storing an exponential number of intermediate results, trading off faster computation for increased memory usage. While the proposed approach is theoretically interesting, its practical impact may be limited. Our current algorithm is already very fast and not the bottleneck in most practical applications. Nevertheless, this alternative presents an interesting avenue for future research, particularly in comparing its performance to our polynomial-time procedure.

5. Evaluation

In this section, we report the results of an evaluation of the algorithms presented above. The experiments were conducted using the PANDA$_\pi$ planning system[3] (Höller & Behnke, 2021; Höller, Bercher, Behnke, & Biundo, 2020b). Specifically, we used the public release and locally integrated Algorithms 5 and 6 to compute the relaxed preconditions and effects independently of the planning procedure as a preprocessing step. Algorithms 1, 2, to find state-independent effects, and 3 for the mandatory preconditions are mainly implemented as a script that calls the planner accordingly.

3. https://panda-planner-dev.github.io/

Code and scripts are available in the respective repository (Olz, Lodemann, Jutz, Schmautz, Borowiec, Biundo, & Bercher, 2025).

5.1 The Evaluation Setting

We run our experiments on a server with a Xeon(R) Gold 6144 CPU and 174 GiBs of RAM running Ubuntu 20.04 LTS.

The inference problem of preconditions and effects of compound tasks can be reduced to several planning problems as outlined in section 3. We will call these planning problems *decision problems* in the remainder to distinguish them from the general planning problems of which we want to analyze the compound task's preconditions and effects. We attempted to solve every decision problem by using several planning systems in the following order. Every planner was granted 1800 seconds (= 30 minutes) and 8 GiBs of RAM for one decision problem. First, we handed each problem to the progression-based planner PANDA$_\pi$ with GBFS and loop detection in combination with the Relaxed Composition (RC) heuristic (Höller et al., 2020b) with the classical Add heuristic (Bonet & Geffner, 2001) as inner heuristic, since this is its currently best-performing configuration for the t.o. setting (Höller & Behnke, 2021). If a timeout occurred, the problem was given to PANDA$_\pi$-SAT (Behnke et al., 2018; Behnke, 2021). If we still got no result, we tried PANDA$_\pi$-BDD (Behnke & Speck, 2021). Overall, we set a timeout of 15 days to calculate all preconditions and effects of all compound tasks of one planning problem instance. One of the main issues leading to the high computation times was considering the problems that do not have a solution, as most planners are primarily designed for solvable planning problems. Specifically, although the decision problem of whether a total-order HTN planning problem is unsolvable is decidable, most current planners are not decision procedures. This means they might run indefinitely without being able to prove the problem's unsolvability.

We considered the total-order IPC 2020 benchmark set[4] and added the two new domains, Lamps and SharpSAT, from 2023[5] to evaluate the presented techniques and algorithms. In total, we considered 26 domains. Since we were not able to calculate the exact preconditions and effects for most of the contained problem instances within the determined time limit we restricted the analysis to the *first problem instance* of every domain with one exception: We considered the whole AssemblyHierarchical domain since here we got results for all problem instances quickly. Additionally, we excluded the initial compound task of every instance, which has only one method containing the initial task network, in order to get results in reasonable time. As they represent the whole planning problem we see little information gain in calculating their preconditions and effects.

In order to measure the impact of using the relaxed preconditions and effects to calculate the exact ones, presented in Section 4, we calculated them once naively with only using mentioned preconditions and effects and once with the help of the precondition-relaxed version. The other mentioned optimizations we exploited in both runs.

When we compare exact positive guaranteed effects with the relaxed version we consider the adapted positive effects ($cor\text{-}eff_*^{\emptyset+}(c) = eff_*^{\emptyset+}(c) \setminus prec(c)$) instead of the latter so that the relation $cor\text{-}eff_*^{\emptyset+}(c) \subseteq eff_*^{+}(c)$ holds as already discussed in Subsection 2.2.2.

4. https://ipc2020.hierarchical-task.net/
5. https://ipc2023-htn.github.io/

To increase the informative value we also calculate the preconditions and effects of every method in addition to every task. This was done by adding an extra task to the problem which can only be decomposed by the method we want to analyze.

In the following subsections, we present our findings, which are visualized in graphs. For readers who are interested in delving into specific aspects in more detail, we also provide two tables with some of the underlying data in the appendix.

5.2 Computation Effort

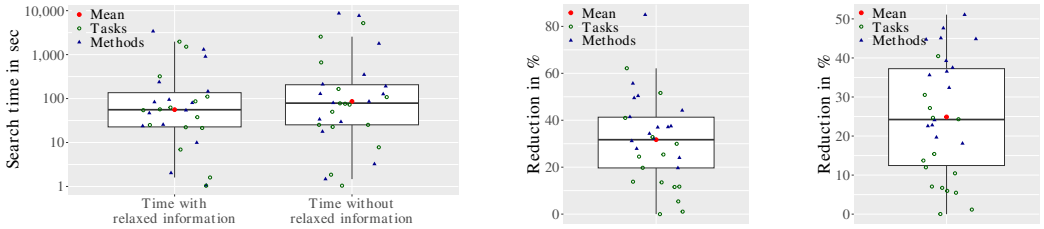
Overall, we observed that calculating the *exact* preconditions and effects of compound tasks and methods is often not possible in reasonable time. This is not surprising given the complexity of the underlying problems (see Theorem 1). In particular, for our evaluation, this means that we do not have information about both Monroe domains and the methods of Minecraft Player since the analyses terminated after the runtime exceeded 15 days. Moreover, we could not completely analyze the domains Blocksworld-GTOHP, Hiking, Satellite, Logistic, Minecraft Player (tasks), SharpSAT, and Freecell because for each of them some of the decision problems were not solvable by any of the used solvers within the given time or memory limit. This was also the case for computing the preconditions and effects of compound *tasks* in the domain Snake, however, the evaluation was successful for the ones of methods.

In this subsection, we want to report how much time it took to infer the exact preconditions and effects and the time that we could save by using the optimizations. For this, we only compare and report the computation effort for domains and problems, respectively, for which the analyses terminated within the time limit in both cases, i.e., when we used the additional information of precondition-relaxed effects and when we did not. Additionally, we excluded the problems for which PANDA$_\pi$-SAT or PANDA$_\pi$-BDD were needed since they skew the results disproportionally (so only problem instances that could be fully analyzed with only PANDA$_\pi$ with GBFS are included). This leaves us with the first problem instances of 15 domains. (The problem instances 3 to 30 of the AssemblyHierarchical domain are only solvable when we used the additional information of precondition-relaxed effects. Transport and problem instance 2 of AssemblyHierarchical needed PANDA$_\pi$-BDD, Snake needed PANDA$_\pi$-SAT.)

We report statistics concerning the inference time in Figure 4. The overall time needed to infer preconditions and effects of one problem instance is given in Figure 4a. On average 286 seconds were needed to calculate the information for compound tasks with optimizations and 604 seconds without. For the methods, it took 427 seconds and 1295 seconds, respectively. Figure 4b illustrates the reduction of calculation effort in percentage terms. Lastly, we report the number of decision problems that we saved by using the precondition-relaxed effects in Figure 4c. For the compound tasks, on average 961 decision problems were needed with the optimization and 1320 without. For preconditions and effects of methods, we have 1712 versus 2940 decision problems.

5.3 Quantitative Analysis

In this section, we present the main results of this note, namely the quantitative evaluation of precondition-relaxed effects and executability-relaxed preconditions compared to the exact



(a) Time to infer exact precs/effs with (left) and without (right) optimizations.

(b) Reduction of calculation time.

(c) Reduction of decision problems.

Figure 4: Boxplots displaying the time and number of decision problems saved per problem instance when *utilizing relaxed* preconditions and effects for *inferring exact* preconditions and effects, compared to the naive approach.

ones. For this, we excluded all of the Assembly instances except for the first one to not overrepresent this domain. However, in the last subsection, we will have a closer look at how the results scale with increasing problem size.

Preconditions Concerning preconditions of compound tasks our results show that the set of executability-relaxed preconditions and the exact ones are *identical*. This is not only surprising but also a quite positive result since for the considered set of domains we can solve an EXPTIME-complete problem (in the worst case) by using a polynomial algorithm.

When looking at the preconditions of methods, the results are slightly weaker. Three of 47 preconditions are missing in the set of executability-relaxed effects in the Elevator domain, which leads to an approximation quality of 93.62% for this domain. All others match perfectly. So, in total we observed an approximation quality of 99.96% overall and 99.62% on average per domain.

Guaranteed Effects Now, we report the comparison of exact and relaxed guaranteed effects, which are displayed in Figure 5.

In case of *negative* effects the sets $\mathit{eff}_*^{\emptyset-}(c)$ and $\mathit{eff}_*^{-}(c)$ are again identical for tasks! When we consider negative effects of methods, the sets are nearly identical with the exception of six missing effects each in the Towers (out of 150) and Rover (out of 51) domain as well as eight out of 28 in the Elevator domain. This results in an overall approximation quality of 99.70%, an average approximation quality per domain of 97.39% for the methods, and 100% for the tasks.

Moving on to guaranteed *positive* effects we can see a diverse spectrum. The approximation of relaxed effects of tasks is very good in domains like Assembly, Childsnack, Lamps, Robot, Towers, and Woodworking. On the other side, we find hardly any relaxed positive guaranteed effects in Elevator-Learned, Entertainment, and Minecraft-Regular. In the other domains, the approximation quality is distributed between around 20% and 80%. Overall we have an approximation quality of 63.67% and 58.37% on average per domain for the tasks. When we compare these results with those of the methods we can see that the relaxed effects of methods are closer to the exact effects in most domains with 90.66% overall approximation quality and 75.84% average approximation quality per domain.

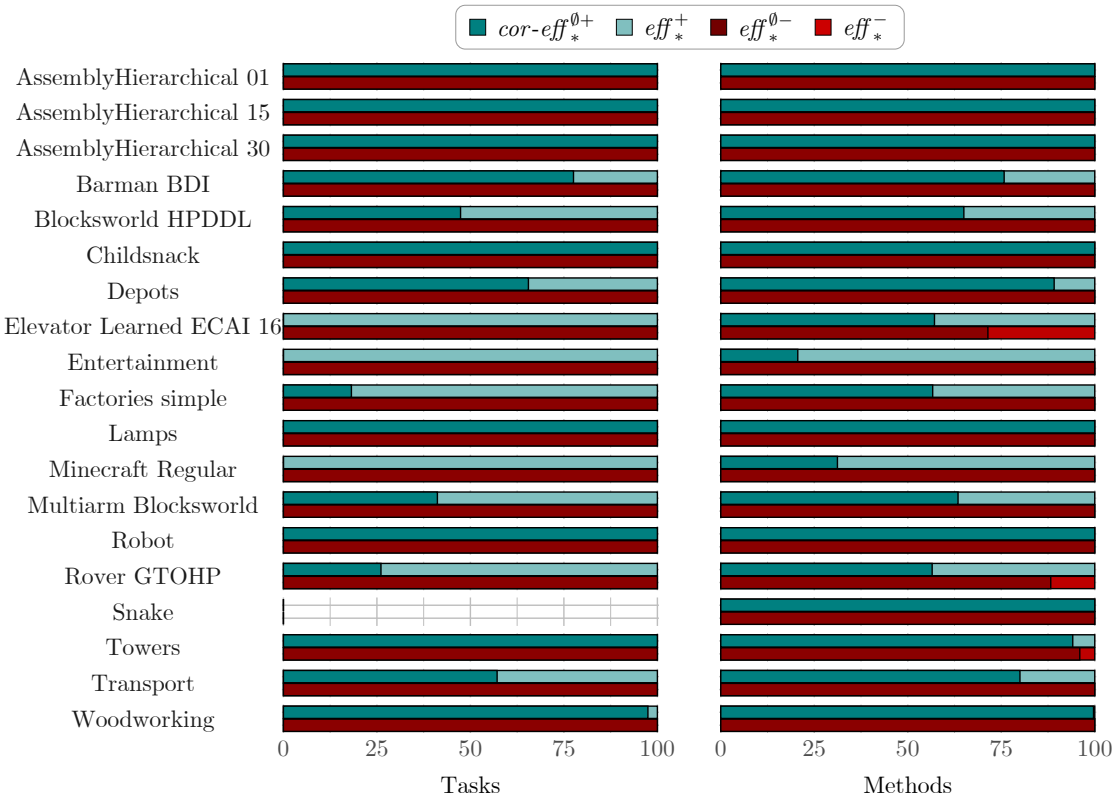


Figure 5: The percentage of precondition-relaxed effects compared to exact ones per domain. The darker colors indicate the percentage of the prec-relaxed effects, i.e. a full bar means that exact and prec-relaxed effects are the same. If X% of a bar is dark then X% of the exact effects are also precondition-relaxed effects. The first line of each domain represents positive effects, the second lines are negative ones.

For the above evaluation, we used the sets *cor-eff*$_*^{\emptyset+}(c)$ (i.e., adapted positive precondition-relaxed effects) instead of the original precondition-relaxed effects as also mentioned earlier. We report on the number of occurrences here. For tasks, 51 out of 447 precondition-relaxed effects are also preconditions. All of them are found in the Woodworking domain. So overall, we adapted 11.41% of the relaxed positive effects. On average per domain, we needed to adjust 2.49%. Considering methods, 798 of 5,117 effects are preconditions as well. This equals 15.60%. These 798 facts are spread over 10 domains resulting in 13.19% effects that need to be adapted on average per domain.

Possible Effects Now, we discuss the possible effects. We first compare exact and precondition-relaxed possible effects, afterwards, we compare possible precondition-relaxed effects with the guaranteed ones. For the tasks, there are 4,083 possible negative precondition-relaxed effects while there are 4,040 exact ones (recall that the *possible* precondition-relaxed effects are an overapproximation of the exact ones). This equals an accuracy of 98.95%. For the methods, we have 16,748 precondition-relaxed and 16,651

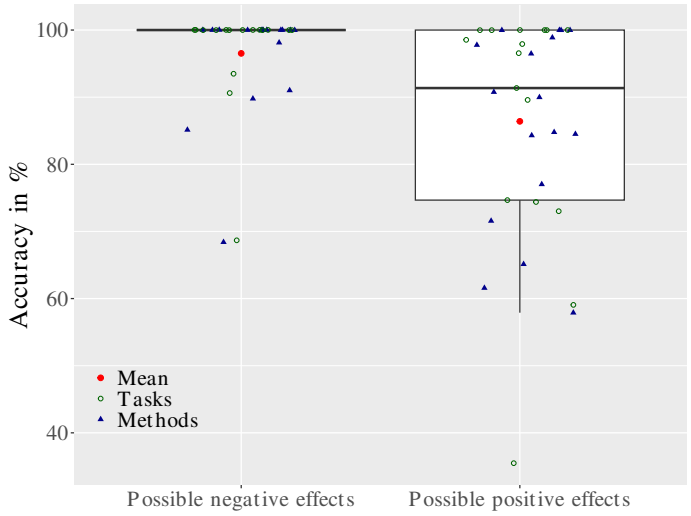


Figure 6: Boxplot displaying percentage difference of possible precondition-relaxed and exact effects. One data point represents one problem instance.

exact effects, which results in 99.42% accuracy. The average accuracy is 97.05% for the tasks and 96.03% for the methods.

Considering positive effects, we found 2,784 possible precondition-relaxed effects of tasks and 12,787 of methods. The exact ones are 2,182 and 11,278 many, respectively. This results in an accuracy of 78.38% (tasks) and 88.20% (methods). On average per problem, we have 86.91% (tasks) and 85.92% (methods) accuracy.

All in all, we can see that the possible precondition-relaxed effects approximate the exact ones quite well in the analyzed domains. We report their percentage difference for positive and negative effects in Figure 6.

In Figure 7, we report how many of the (precondition-relaxed) possible effects are guaranteed ones and compare them additionally with the mentioned effects. For tasks, we can see that the shares of guaranteed effects are quite low for almost all domains except for Assembly and Woodworking. For methods, the shares become higher, which matches our intuition since the guaranteed effects of tasks are the conjunction of the ones of their respective methods. If we compare possible and mentioned effects, we see a quite diverse spectrum. There are domains (e.g., Assembly and Robot) in which the sets are identical. On the other hand, in domains like Childsnack, Entertainment, and Minecraft Regular less than half (for either positive or negative ones) of the mentioned are possible effects. Thus, it seems to depend on the domain whether it pays out to compute possible rather than just mentioned effects (the latter is easier to compute).

Influence of Instance Size Lastly, we look at how the numbers of preconditions and effects scale with increasing instance size. We do this only exemplary for the Assembly-Hierarchical domain since we did not get results for the other ones. Figure 8 on the left shows that the maximum number and mean of preconditions and guaranteed effects, as well as the mean of possible effects per compound task converge quite fast to some number between zero and four. The maximum number of possible effects increases linearly as

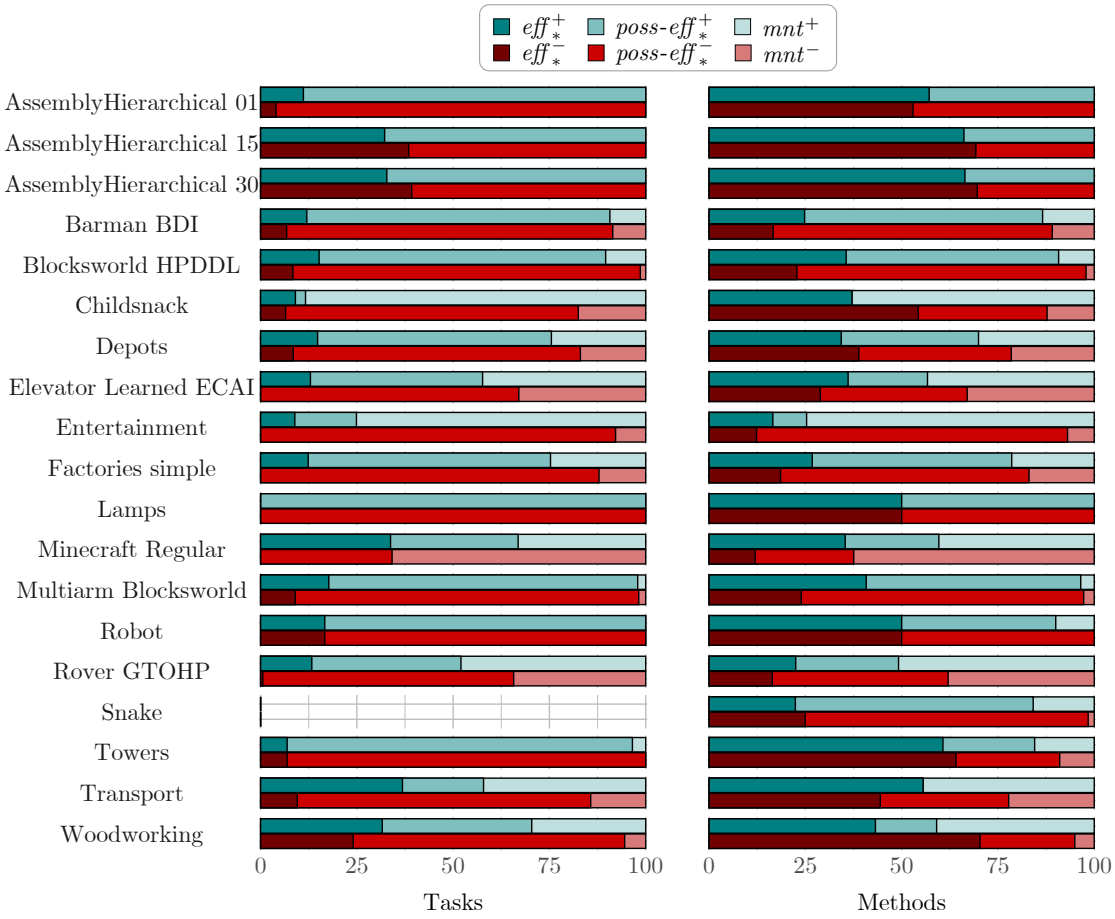


Figure 7: Percentage of how many guaranteed and possible effects are mentioned effects, respectively. The lightest colors represent the percentage of mentioned effects that are not possible (including guaranteed) ones, while the middle colors represent the percentage of effects that are possible but not guaranteed.

shown in Figure 8 on the right. Those findings are reasonable and can be explained as follows: The number of facts increases with increasing instance size, so it is likely that more facts are reachable, which leads to more maximum possible effects. On the other hand, the problem instances were created using one lifted domain description, which means that with increasing instance size one lifted action creates more grounded instances, but they do not change their structure, which makes it reasonable that the number of guaranteed and mean of possible effects does not change with increasing size. However, this is an evaluation of just one domain, it might be different for another one, especially if the problem instances are not just created by only increasing the number of grounded instances of the same lifted actions.

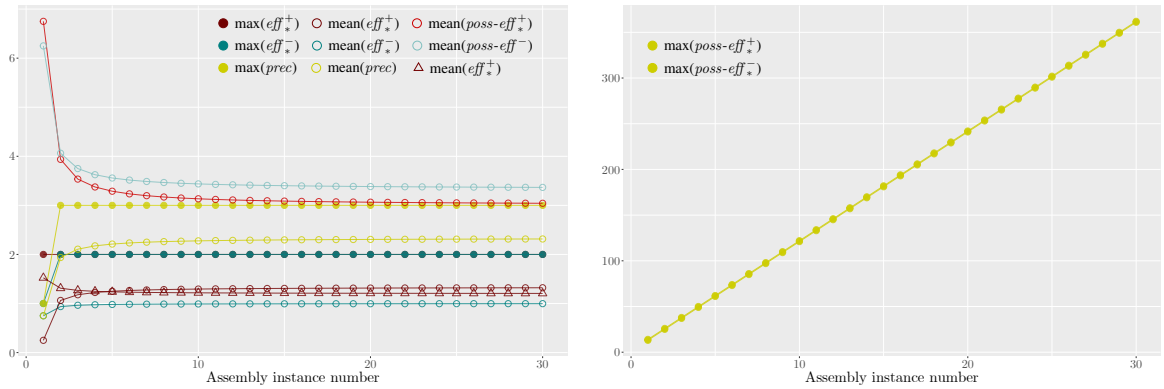


Figure 8: Number of preconditions and effects per compound task (circles) or method (triangles) with increasing problem instance size in the Assembly domain.

5.4 Comparison to HyperTensioN and Lilotane

We mentioned that the planners HyperTensioN (Magnaguagno et al., 2021) and Lilotane (Schreiber, 2021) also infer preconditions in a preprocessing step. In this subsection, we compare these approaches empirically, though this comparison comes with inherent limitations due to differences in the nature of the models they operate on. HyperTensioN and Lilotane are designed to work with lifted models, while our inference algorithms operate on a grounded model. These differences lead to two possible strategies for comparison:

1. We pass the lifted problem description to HyperTensioN and Lilotane and ground their output afterwards. However, this approach complicates the quantitative comparison of results due to transformations introduced by the grounding process. Grounders, such as the one of the Panda$_\pi$ planning system (Behnke et al., 2020), perform feature compilations (e.g., handling of negative preconditions) resulting in additional artificial facts and actions and prune unreachable parts of the domain. Similarly, HyperTensioN and Lilotane prune parts of the domain during preprocessing to improve planning efficiency. While these transformations are advantageous for planning, they can lead to differences in the sets of inferred preconditions making direct comparisons difficult.

2. We pass the *grounded* problem, generated by the grounder of the Panda$_\pi$ planning system and compiled to HDDL, to both HyperTensioN and Lilotane. This new HDDL problem contains only nullary predicates, effectively converting the lifted HTN planning problem into a propositional form. This allows for a more direct comparison of the inferred preconditions and effects. However, even here, challenges arise because HyperTensioN and Lilotane still prune parts of the domain during their preprocessing. Additionally, comparing runtime in this scenario may be unfair, as their algorithms are optimized for lifted domains and may not perform as well on grounded models and could circumvent techniques that only work in the original, lifted domain.

Given these considerations, we adopt the following approach for our evaluation: To compare the sets of inferred preconditions, we use the grounded problem as input for all three systems (our approach, HyperTensioN, and Lilotane). To compare runtime, we also evaluate HyperTensioN and Lilotane on their intended lifted problem.

Since our algorithm to infer relaxed preconditions runs in polynomial time and can be computed in a reasonable timeframe, we consider all problem instances from all domains for this analysis. We report the minimum, maximum, and average inference times of our poly-time algorithms in Table 2 in the Appendix.

domain	$prec^{\emptyset}$(m)	Lilotane	HyperTensioN
Assembly	**48**	**48**	**48**
	3066	2834	2834
Depots	**125**	115	115
	40213	35030	34880
Hiking	**5802**	5790	5790
	258444	258360	258360
Mean coverage compared to $prec^{\emptyset}$(m)	100	99.25	99.19

Table 1: Sum of all inferred preconditions of all methods per problem instance. Grey rows correspond to the first problems, white rows correspond to the largest problems for which all approaches terminated within the given time restrictions. For the other domains, the sums are identical for all three approaches.

In Table 1, we report the number of inferred method preconditions, though these results should be interpreted with caution. As previously mentioned, our relaxed inference procedure is capable of identifying nearly all exact preconditions in the evaluated problems. HyperTensioN and Lilotane yield similar results, so we only highlight the domains where differences exist between the relaxed preconditions and those identified by the other two planners. Except for the larger Assembly instances, Depot, and Hiking, all approaches identify the same number of preconditions per problem, provided they terminate within 1800 seconds. Overall, out of 953 problems, our algorithm – incorporating both relaxed effects and preconditions – was able to terminate within 1800 seconds for 887 problems. In comparison, HyperTensioN terminated for 739 problems and Lilotane for 819. On average, HyperTensioN identified 99.19% and Lilotane 99.25% of our relaxed preconditions.

As HyperTensioN and Lilotane are designed for lifted models and thus more general, a search time comparison based only on the grounded problems seems unfair as described above. So, we also evaluate their inference times on the original lifted problems. In Figure 9, we report the time for all three approaches to infer the preconditions of methods, with HyperTensioN and Lilotane having two variants: one for the original lifted problem and one for the problem that was first grounded. In the lifted domains, HyperTensioN terminated within the given time limits for all 953 problems, while Lilotane solved 948. Although our approach performed worse than these lifted versions in terms of both speed and the number of successfully processed problems, it outperformed the grounded variants of HyperTensioN and Lilotane and solved more problems (905 versus 739 and 819, respectively). Note that the differences are also influenced by the grounding process, which successfully grounded only 906 of the 953 problems. As a result, these findings should again be interpreted cautiously.

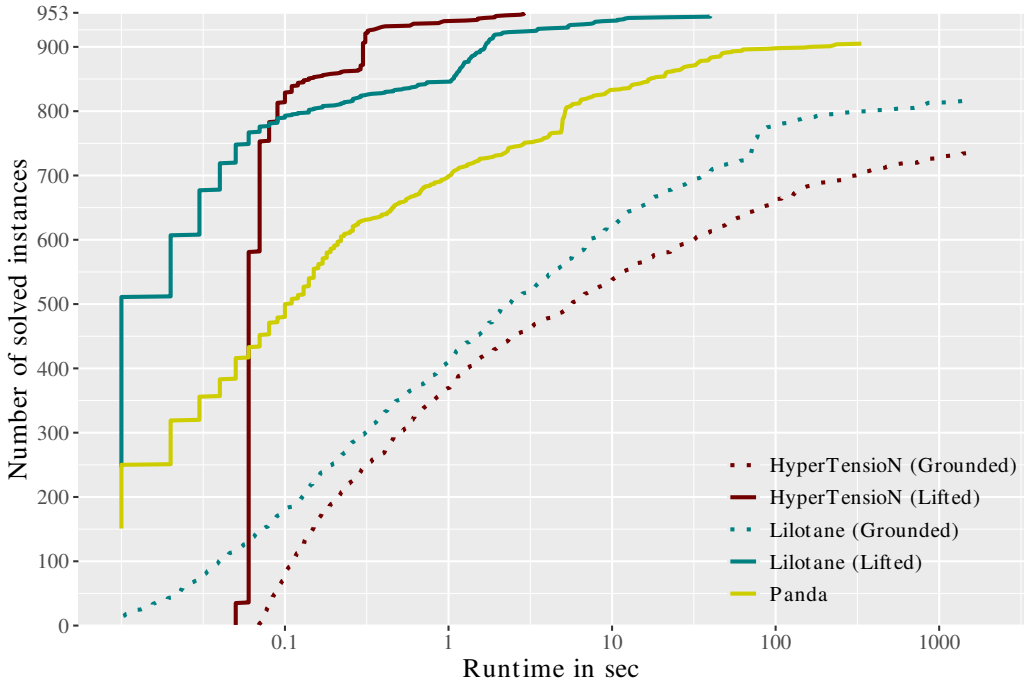


Figure 9: Inference time for relaxed preconditions of methods calculated by HyperTensioN, Lilotane, and our approach. Beware of the log-scale.

6. Conclusion

We introduced pseudocode for computing inferred preconditions and effects of compound tasks and methods. We provided this code both for the *exact* sets (i.e., the complete set of existing preconditions and effects) and for *relaxed* sets (i.e., a subset of the exact ones). Since relaxed preconditions and effects can be computed in polynomial time, whereas the exact sets are between **PSPACE**- and **EXPTIME**-complete, two natural questions arose: (1) Can the exact sets still be computed in practice? (2) How useful are the relaxed ones, i.e., what percentage of the exact preconditions and effects can still be discovered using the relaxed computation?

In our extensive evaluation, our findings show that, as expected, calculating the *exact* preconditions and effects is indeed hardly feasible. Fortunately, our experiments also demonstrate that, in the evaluated domains, the relaxed preconditions reveal large parts of the exact preconditions and effects, despite the relaxation that reduces complexity to only P. By making the inference methods more accessible, we believe that these preconditions and effects are not only useful for the existing applications such as heuristics or pruning in search-based or SAT-based planning, but also provide opportunities for new ideas and further research in total-order HTN planning. In future work, we hope to draw inspiration from HyperTensioN and Lilotane to generalize our inference techniques to lifted models.

Acknowledgments

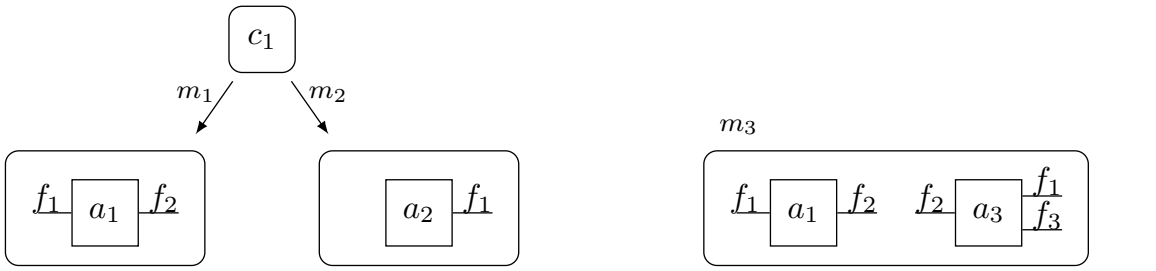
In Subsection 3.3, we added an assumption necessary for the correctness of Algorithm 5, thanks to Patrik Haslum, which had not been stated explicitly in the foundation paper (Olz et al., 2021). Additionally, we sincerely thank our anonymous reviewers for their thoughtful and constructive feedback, which significantly enhanced this article.

Pascal Bercher is the recipient of an Australian Research Council (ARC) Discovery Early Career Researcher Award (DECRA), project number DE240101245, funded by the Australian Government.

Appendix A

A.1 On Effects versus Postconditions

Unfortunately, the definitions of our inferred postconditions and effects do not always align perfectly with our intuitive understanding of their differences. To recap, intuitively, effects are considered to be facts explicitly added by a refinement, whereas postconditions may also hold after execution because they were true before and were not subsequently deleted. The following examples in Figure 10 illustrate some known pitfalls:



(a) A compound task $c1$ with its two decompositions methods $m1$ and m_2.

(b) A decomposition method m_3 (without compound task).

Figure 10: Examples illustrating deficiencies concerning the definitions of inferred effects and postconditions.

In Figure 10a, there is the compound task c_1 with its two decomposition methods. The fact f_1 is included in $\mathit{eff}^+_*(c_1)$ because f_1 holds after the execution of both a_1 and a_2, and it is not a mandatory precondition. However, f_1 is not actually added by a_1, and thus, intuitively, we might consider it a postcondition, yet by definition is declared an effect. This is accurately reflected in the relaxed version, where $f_1 \notin \mathit{eff}^{\emptyset+}_*(c_1)$.

In Figure 10b, we have a method containing two primitive tasks within its task network. The fact f_1 is required as a precondition for a_1 and is additionally added by a_3. It is not entirely clear whether f_1 should be classified as an effect or merely a postcondition. Intuitively, we tend to consider it a postcondition, but since it is explicitly added, one might also view it as an effect. Our definitions classify f_1 as a postcondition, as $f_1 \in \mathit{post}^+_*(m_3)$ and $f_1 \notin \mathit{eff}^+_*(m_3)$. In the relaxed version, however, f_1 is considered an effect, i.e., $f_1 \in \mathit{eff}^{\emptyset+}_*(m_3)$. This example demonstrates that $\mathit{eff}^{\emptyset+}_*(m_3) \subseteq \mathit{eff}^+_*(m_3)$ does not always hold, but rather $\mathit{eff}^{\emptyset+}_*(m_3) \subseteq \mathit{post}^+_*(m_3)$.

When developing the definitions of inferred effects and postconditions, we aimed to avoid these unintuitive situations. However, each version we considered had its own weaknesses, leading us to adopt the version presented here. For practical applications, these issues have minimal impact but should be carefully considered.

A.2 Additional Evaluation Results

In Table 2, we provide an overview of the minimum, maximum, and average computation times for inferring the *relaxed* preconditions and effects per problem, as well as the number of problems per domain for which the inference algorithm successfully terminated. Since the runtime for inferring relaxed preconditions and effects is significantly lower compared to the exact ones, we include all problems from all domains in this table. However, note that in some larger instances, it was not our inference algorithm but rather the parser or grounder exceeding the given time or resource constraints. The maximum and average times are sometimes quite high because they include unsolvable problems. Olz and Bercher (2023b) provide statistics on the percentage of overall search time required for calculating inferred preconditions and effects in their supplemental material.

An overview over the maximum and average number of *exact* preconditions as well as possible and guaranteed effects per compound task and method are displayed in Table 3 and Table 4, respectively. Grey lines refer to the exact preconditions and effects, while white lines correspond to relaxed ones.

domain	problems	coverage	min	max	mean
Assembly	30	30	0.01	0.24	0.09
Barman-BDI	20	20	0.04	276.46	22.21
Blocksw.-GTOHP	30	30	0.02	95.39	6.10
Blocksw.-HPDDL	30	30	0.02	24.55	2.65
Childsnack	30	26	0.09	542.35	37.87
Depots	30	29	0.02	1073.22	49.02
Elevator-Learned	147	147	0.02	0.95	0.29
Entertainment	12	12	0.01	0.23	0.09
Factories	20	20	0.03	7.00	1.53
Freecell-Learned	60	45	12.24	682.26	242.92
Hiking	30	26	0.15	892.97	47.68
Lamps	30	30	0.02	20.15	1.73
Logistics-Learned	80	80	0.02	16.98	3.15
Minecraft Pl.	20	4	5.09	285.48	159.38
Minecraft Reg.	59	42	0.03	62.43	2.37
Monroe FO	20	20	35.01	51.02	39.57
Monroe PO	20	20	35.56	72.90	42.19
Multiarm-Blocksw.	74	74	0.02	6.29	0.74
Robot	30	30	0.01	0.63	0.14
Rover	30	27	0.02	1651.06	207.67
Satellite	20	20	0.01	865.71	98.71
SharpSAT	21	14	0.02	37.43	5.34
Snake	20	20	0.05	25.12	2.22
Towers	20	20	0.02	2.74	0.64
Transport	40	40	0.01	18.12	1.72
Woodworking	30	30	0.02	237.14	34.22

Table 2: For each domain, we report the minimum, maximum, and average inference times for the relaxed preconditions and effects per problem in seconds. The "problem" and "coverage" columns indicate the number of problems for which the inference algorithm successfully terminated within the given time and memory limits (noting that in some cases, these limits were exceeded during the parsing or grounding phase).

domain		max($poss\text{-}eff^-_*(c)$)	mean($poss\text{-}eff^-_*(c)$)	max($poss\text{-}eff^+_*(c)$)	mean($poss\text{-}eff^+_*(c)$)	max($eff^-_*(c)$)	mean($eff^-_*(c)$)	max($eff^+_*(c)$)	mean($eff^+_*(c)$)	max($prec(c)$)	mean($prec(c)$)
Assembly01	Exact	13	6.25	14	6.75	1	0.25	2	0.75	1	0.75
	Relaxed	13	6.25	14	6.75	1	0.25	2	0.75	1	0.75
Assembly15	Exact	181	3.40	182	3.09	2	1.31	2	0.99	3	2.30
	Relaxed	181	3.4	182	3.09	2	1.31	2	0.99	3	2.3
Assembly30	Exact	361	3.37	362	3.04	2	1.32	2	1.00	3	2.32
	Relaxed	361	3.37	362	3.04	2	1.32	2	1	3	2.32
Barman-BDI	Exact	36	13.26	34	14.98	4	0.98	6	1.98	4	1.19
	Relaxed	36	13.26	34	14.98	4	0.98	6	1.54	4	1.19
Blocksw.-HPDDL	Exact	33	10.07	27	8	2	0.86	6	1.36	2	0.86
	Relaxed	33	10.07	29	8.93	2	0.86	1	0.64	2	0.86
Childsnack	Exact	80	11.93	14	1.39	1	0.94	2	1.08	2	1.25
	Relaxed	80	11.93	17	3.92	1	0.94	2	1.08	2	1.25
Depots	Exact	17	7.44	14	5.92	2	0.76	2	1.16	1	0.36
	Relaxed	17	7.44	16	6.48	2	0.76	2	0.76	1	0.36
Elevator-Learned	Exact	13	5.18	10	4.45	0	0	1	1	0	0
	Relaxed	19	7.55	19	7.55	0	0	0	0	0	0
Entertainment	Exact	37	11.79	10	2.95	0	0	2	1.05	0	0
	Relaxed	37	11.79	11	3.95	0	0	0	0	0	0
Factories	Exact	13	6.21	8	4.79	0	0	1	0.79	0	0
	Relaxed	14	6.86	8	4.86	0	0	1	0.14	0	0
Lamps	Exact	2	2	2	2	0	0	0	0	0	0
	Relaxed	2	2	2	2	0	0	0	0	0	0
Minecraft Reg.	Exact	3	1.72	6	3.35	0	0	3	1.69	0	0
	Relaxed	3	1.72	6	3.35	0	0	0	0	0	0
Multiarm-Blocksw.	Exact	24	7.27	23	6.27	2	0.67	6	1.13	2	0.87
	Relaxed	24	7.27	23	6.4	2	0.67	1	0.47	2	0.87
Robot	Exact	3	2	3	2	1	0.33	1	0.33	1	0.67
	Relaxed	3	2	3	2	1	0.33	1	0.33	1	0.67
Rover	Exact	9	4.26	8	3.33	1	0.04	1	0.85	1	0.37
	Relaxed	9	4.56	9	4.48	1	0.04	1	0.22	1	0.37
Towers	Exact	15	5.8	14	5.6	1	0.4	1	0.4	1	0.4
	Relaxed	15	5.8	15	5.8	1	0.4	1	0.4	1	0.4
Transport	Exact	6	3.6	4	2.2	1	0.4	2	1.4	1	0.4
	Relaxed	6	3.6	4	2.2	1	0.4	2	0.8	1	0.4
Woodworking	Exact	11	3.26	13	3.32	3	0.83	5	1.49	4	1.89
	Relaxed	11	3.26	13	4.55	3	0.83	5	2.42	4	1.89

Table 3: Number of preconditions and effects per compound tasks.

domain		$\max(\textit{poss-eff}^-_*(m))$	$\text{mean}(\textit{poss-eff}^-_*(m))$	$\max(\textit{poss-eff}^+_*(m))$	$\text{mean}(\textit{poss-eff}^+_*(m))$	$\max(\textit{eff}^-_*(m))$	$\text{mean}(\textit{eff}^-_*(m))$	$\max(\textit{eff}^+_*(m))$	$\text{mean}(\textit{eff}^+_*(m))$	$\max(\textit{prec}(\text{m}))$	$\text{mean}(\textit{prec}(\text{m}))$
Assembly01	Exact	13	2.43	14	2.67	2	1.29	2	1.52	3	2.29
	Relaxed	13	2.43	14	2.67	2	1.29	2	1.52	3	2.29
Assembly15	Exact	181	2.02	182	1.83	2	1.4	2	1.21	3	2.59
	Relaxed	181	2.02	182	1.83	2	1.4	2	1.21	3	2.59
Assembly30	Exact	361	2.01	362	1.82	2	1.4	2	1.21	3	2.6
	Relaxed	361	2.01	362	1.82	2	1.4	2	1.21	3	2.6
Barman-BDI	Exact	36	8.06	34	9.76	4	1.5	7	2.8	5	1.69
	Relaxed	36	8.06	34	9.76	4	1.5	6	2.12	5	1.69
Blocksw.-HPDDL	Exact	33	8.26	27	6.54	4	1.92	6	2.56	6	2.64
	Relaxed	33	8.26	29	7.21	4	1.92	4	1.72	6	2.64
Childsnack	Exact	20	4.67	2	1.22	3	2.89	2	1.22	3	2.59
	Relaxed	20	4.67	5	1.87	3	2.89	2	1.54	3	2.59
Depots	Exact	16	4.73	13	3.93	7	2.34	5	1.93	5	1.87
	Relaxed	16	4.82	15	4.66	7	2.34	6	2.24	5	1.87
Elevator-Learned	Exact	13	2.32	10	1.96	2	1	5	1.25	4	1.68
	Relaxed	19	3.39	19	3.39	2	0.71	2	0.71	3	1.57
Entertainment	Exact	37	9.45	10	2.36	5	1.25	6	1.55	3	1.2
	Relaxed	37	9.45	11	3.07	5	1.25	3	0.57	3	1.2
Factories	Exact	13	3.32	8	2.84	3	0.74	2	0.97	2	0.84
	Relaxed	14	3.9	8	2.87	3	0.74	1	0.55	2	0.84
Lamps	Exact	2	1	2	1	1	0.5	1	0.5	1	0.5
	Relaxed	2	1	2	1	1	0.5	1	0.5	1	0.5
Minecraft Reg.	Exact	3	1.35	6	2.26	2	0.43	3	1.34	2	0.85
	Relaxed	3	1.35	6	2.66	2	0.43	2	0.82	2	0.85
Multiarm-Blocksw.	Exact	24	5.64	23	5	3	1.39	7	2.11	6	2.48
	Relaxed	24	5.64	23	5.18	3	1.39	4	1.39	6	2.48
Robot	Exact	3	1.43	3	1.29	1	0.71	1	0.71	2	1.43
	Relaxed	3	1.43	3	1.43	1	0.71	1	0.71	2	1.43
Rover	Exact	8	2.68	8	2.1	2	0.71	2	0.96	2	1.21
	Relaxed	9	2.99	9	2.93	2	0.62	1	0.71	2	1.21
Snake	Exact	51	14.41	44	12.32	4	3.65	4	3.28	4	3.61
	Relaxed	51	14.41	44	12.6	4	3.65	4	3.3	4	3.61
Towers	Exact	15	2.63	13	2.63	3	1.85	3	1.89	3	2.44
	Relaxed	15	2.89	15	3.11	3	1.78	3	2	3	2.44
Transport	Exact	4	1.87	3	1.33	2	1.07	3	1.33	2	1
	Relaxed	4	1.87	3	1.33	2	1.07	2	1.07	2	1
Woodworking	Exact	10	2.09	12	2.12	4	1.55	5	1.55	4	2.82
	Relaxed	10	2.09	13	3.44	4	1.55	7	2.81	4	2.82

Table 4: Number of preconditions and effects per decomposition method.

References

Alford, R., Bercher, P., & Aha, D. W. (2015). Tight bounds for HTN planning. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS 2015)*, pp. 7–15. AAAI Press.

Bäckström, C., Jonsson, A., & Jonsson, P. (2012). Macros, reactive plans and compact representations. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pp. 85–90. IOS Press.

Behnke, G. (2021). Block compression and invariant pruning for SAT-based totally-ordered HTN planning. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS 2021)*, pp. 25–35. AAAI Press.

Behnke, G., Höller, D., Bercher, P., Biundo, S., Pellier, D., Fiorino, H., & Alford, R. (2019). Hierarchical planning in the IPC. In *Proceedings of the Workshop on the International Planning Competition*.

Behnke, G., Höller, D., & Biundo, S. (2018). totSAT – Totally-ordered hierarchical planning through SAT. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018)*, pp. 6110–6118. AAAI Press.

Behnke, G., Höller, D., Schmid, A., Bercher, P., & Biundo, S. (2020). On succinct groundings of HTN planning problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*, pp. 9775–9784. AAAI Press.

Behnke, G., & Speck, D. (2021). Symbolic search for optimal total-order HTN planning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021)*, pp. 11744–11754. AAAI Press.

Bercher, P., Alford, R., & Höller, D. (2019). A survey on hierarchical planning – One abstract idea, many concrete realizations. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, pp. 6267–6275. IJCAI.

Bercher, P., Höller, D., Behnke, G., & Biundo, S. (2016). More than a name? On implications of preconditions and effects of compound HTN planning tasks. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*, pp. 225–233. IOS Press.

Bercher, P., Lin, S., & Alford, R. (2022). Tight bounds for hybrid planning. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence and the 25th European Conference on Artificial Intelligence (IJCAI-ECAI 2022)*, pp. 4597–4605. IJCAI.

Bit-Monnot, A., Smith, D. E., & Do, M. (2016). Delete-free reachability analysis for temporal and hierarchical planning. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*, pp. 1698–1699. IOS Press.

Biundo, S., & Schattenberg, B. (2001). From abstract crisis to concrete relief (A preliminary report on combining state abstraction and HTN planning). In *Proceedings of the 6th European Conference on Planning (ECP 2001)*, pp. 157–168. AAAI Press.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*(1-2), 5–33.

Clement, B. J., Durfee, E. H., & Barrett, A. C. (2007). Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research (JAIR)*, *28*, 453–515.

de Silva, L., Meneguzzi, F. R., & Logan, B. (2020). BDI agent architectures: A survey. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI 2020)*, pp. 4914–4921. IJCAI.

de Silva, L., Sardina, S., & Padgham, L. (2009). First principles planning in BDI systems. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pp. 1105–1112. IFAAMAS.

de Silva, L., Sardina, S., & Padgham, L. (2016). Summary information for reasoning about hierarchical plans. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*, pp. 1300–1308. IOS Press.

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

Gazen, B. C., & Knoblock, C. A. (1997). Combining the expressivity of UCPOP with the efficiency of graphplan. In *Proceedings of the 4th European Conference on Planning: Recent Advances in AI Planning (ECP 1997)*, pp. 221–233. Springer.

Geier, T., & Bercher, P. (2011). On the decidability of HTN planning with task insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pp. 1955–1961. AAAI Press.

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

Goldman, R. P., & Kuter, U. (2019). Hierarchical task network planning in common Lisp: the case of SHOP3. In *Proceedings of the 12th European Lisp Symposium (ELS 2019)*, pp. 73–80. ACM.

Haslum, P., & Jonsson, P. (2000). Planning with reduced operator sets. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS 2000)*, pp. 150–158. AAAI Press.

Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D., & Alford, R. (2020a). HDDL: An extension to PDDL for expressing hierarchical planning problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*, pp. 9883–9891. AAAI Press.

Höller, D., Bercher, P., Behnke, G., & Biundo, S. (2020b). HTN planning as heuristic progression search. *Journal of Artificial Intelligence Research (JAIR)*, *67*, 835–880.

Höller, D., & Behnke, G. (2021). Loop detection in the PANDA planning system. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS 2021)*, pp. 168–173. AAAI Press.

Ilghami, O., Muñoz-Avila, H., Nau, D. S., & Aha, D. W. (2005). Learning approximate preconditions for methods in hierarchical plans. In *Proceedings of the 22nd International Conference on Machine Learning (ICML 2005)*, pp. 337–344. ACM.

Ilghami, O., Nau, D. S., Muñoz-Avila, H., & Aha, D. W. (2002). CaMeL: Learning method preconditions for HTN planning. In *Proceedings of the 6th International Conference on AI Planning & Scheduling (AIPS 2002)*, pp. 131–142. AAAI Press.

Kambhampati, S., Mali, A., & Srivastava, B. (1998). Hybrid planning for partially hierarchical domains. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI 1998)*, pp. 882–888. AAAI Press.

Magnaguagno, M. C., Meneguzzi, F. R., & de Silva, L. (2021). Hypertension: A three-stage compiler for planning. In *Proceedings of the 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*, pp. 5–8.

Marthi, B., Russell, S. J., & Wolfe, J. A. (2007). Angelic semantics for high-level actions. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pp. 232–239. AAAI Press.

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

Olz, C., & Bercher, P. (2022). On the efficient inference of preconditions and effects of compound tasks in partially ordered HTN planning domains. In *Proceedings of the 5th ICAPS Workshop on Hierarchical Planning (HPlan 2022)*, pp. 47–51.

Olz, C., & Bercher, P. (2023a). Can they come together? A computational complexity analysis of conjunctive possible effects of compound HTN planning tasks. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS 2023)*, pp. 314–323. AAAI Press.

Olz, C., & Bercher, P. (2023b). A look-ahead technique for search-based HTN planning: Reducing the branching factor by identifying inevitable task refinements. In *Proceedings of the 16th International Symposium on Combinatorial Search (SoCS 2023)*, pp. 65–73. AAAI Press.

Olz, C., Biundo, S., & Bercher, P. (2021). Revealing hidden preconditions and effects of compound HTN planning tasks – A complexity analysis. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021)*, pp. 11903–11912. AAAI Press.

Olz, C., Höller, D., & Bercher, P. (2023). The PandaDealer system for totally ordered HTN planning in the 2023 IPC. In *Proceedings of the 11th International Planning Competition: Planner Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2023)*.

Olz, C., Lodemann, A., & Bercher, P. (2024). A heuristic for optimal total-order HTN planning based on integer linear programming. In *Proceedings of the 27th European Conference on Artificial Intelligence (ECAI 2024)*, pp. 4303–4310. IOS Press.

Olz, C., Lodemann, A., Jutz, B., Schmautz, M., Borowiec, M., Biundo, S., & Bercher, P. (2025). Experimental results for the JAIR article "An extensive empirical evaluation of inferring preconditions and effects of compound tasks in ground HTN planning problems". Zenodo. doi: 10.5281/zenodo.14678643.

Olz, C., Wierzba, E., Bercher, P., & 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)*.

Schreiber, D. (2021). Lilotane: A lifted SAT-based approach to hierarchical planning. *Journal of Artificial Intelligence Research (JAIR)*, *70*, 1117–1181.

Schreiber, D., Pellier, D., & Fiorino, H. (2019). Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS 2019)*, pp. 382–390. AAAI Press.

Taitler, A., Alford, R., Espasa, J., Behnke, G., Fišer, D., Gimelfarb, M., Pommerening, F., Sanner, S., Scala, E., Schreiber, D., Segovia-Aguas, J., & Seipp, J. (2024). The 2023 international planning competition. *AI Magazine*, *45*(2), 280–296.

Tenenberg, J. D. (1988). *Abstraction in Planning*. Ph.D. thesis, Computer Science Department, University of Rochester.

Tsuneto, R., Hendler, J., & Nau, D. (1998). Analyzing external conditions to improve the efficiency of HTN planning. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI 1998)*, pp. 913–920. AAAI Press.

Waisbrot, N., Kuter, U., & Könik, T. (2008). Combining heuristic search with hierarchical task-network planning: A preliminary report. In *Proceedings of the 21st International Florida Artificial Intelligence Research Society Conference (FLAIRS 2008)*, pp. 577–578. AAAI Press.

Wu, Y. X., Olz, C., Lin, S., & Bercher, P. (2023). Grounded (lifted) linearizer: Solving partial order HTN problems by linearizing them. In *Proceedings of the 11th International Planning Competition: Planner Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2023)*.

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

Zhuo, H. H., Hu, D. H., Hogg, C., Yang, Q., & Munoz-Avila, H. (2009). Learning HTN method preconditions and action models from partial observations. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pp. 1804–1809. Morgan Kaufmann.