Modeling Assistance for Hierarchical Planning: An Approach for Correcting Hierarchical Domains with Missing Actions

Songtuan Lin[1], **Daniel Höller**[2], **Pascal Bercher**[1]
[1]School of Computing, The Australian National University
[2]Computer Science Department, Saarland University
{songtuan.lin, pascal.bercher}@anu.edu.au, hoeller@cs.uni-saarland.de

Abstract

The complexity of modeling planning domains is a major obstacle for making automated planning techniques more accessible, raising the demand of tools for providing modeling assistance. In particular, tools that can automatically correct errors in a planning domain are of great importance. Previous works have devoted efforts to developing such approaches for correcting classical (non-hierarchical) domains. However, no approaches exist for hierarchical planning, which is what we offer here. More specifically, our approach takes as input a flawed hierarchical domain together with a plan known to be a solution but actually contradicting the domain (due to errors in the domain) and outputs corrections to the domain that add *missing* actions to the domain which turn the plan into a solution. The approach achieves this by compiling the problem of finding corrections to another hierarchical planning problem.

Introduction

In the last few decades, a significant development of Automated Planning has been witnessed. Many techniques are developed for both hierarchical (Bercher, Alford, and Höller 2019) and non-hierarchical (Ghallab, Nau, and Traverso 2004) planning, e.g., see the work by Bonet and Geffner (2001), Hoffmann and Nebel (2001), Helmert (2006), and Höller et al. (2020). Along the way, some application scenarios of automated planning like greenhouse logistic management (Helmert and Lasinger 2010) and Robotics (Karpas and Magazzeni 2020) have also been explored.

In spite of those developments, we did not see a wide deployment of planning techniques in areas outside academia. One major reason for this is arguably the complexity of the task of *modeling planning domains*. This task is error-prone and hence requires domain modelers to spend immense efforts on correcting domains crafted by them.

For the purpose of making planning techniques more accessible, it is thus vital to have tools which provide *modeling support* to domain modelers. Many techniques in this direction have been developed that provide not only basic syntax-level support (Vaquero et al. 2013; Muise 2016; Strobel and Kirsch 2020) but advanced assistance (Hoffmann 2011; Sreedharan et al. 2020; Lin, Grastien, and Bercher 2023; Gragera et al. 2023) that requires reasoning on the structure of the domain. Those approaches are for *non-hierarchical* planning. Only few (McCluskey, Richardson, and Simpson 2002) exist for *hierarchical* planning where a sequence of *primitive actions* (i.e., a plan) can only be obtained by decomposing so-called *compound tasks* using *methods*. Hierarchical planning has received increasing attention in the last decade. It is strictly more expressive than non-hierarchical planning (Höller et al. 2014, 2016; Lin and Bercher 2022) and hence can model a broader range of problems.

As a contribution toward providing modeling assistance for hierarchical planning, we revisit the scenario where we correct a flawed domain by turning an action sequence that is known to be valid but actually contradicts the domain into a solution. A practical approach (Lin, Grastien, and Bercher 2023) solving this problem has been developed for *non-hierarchical* planning whereas only theoretical investigations (Lin and Bercher 2021, 2023) have been done for the *hierarchical* setting. It was shown that the problem is $\mathbf{NP}$-complete in hierarchical planning. In this paper, we propose an approach which solves this problem in *hierarchical* planning. Our approach takes as input a (flawed) hierarchical domain and a plan and outputs an *optimal* set of corrections to the domain which turns the plan into a solution.

Our approach is developed based on the *Hierarchical Task Network* (HTN) formalism, which is the most broadly used hierarchical planning formalism and on which most theoretical investigations and practical implementations were done. We further make the following two assumptions: 1) The input domain is *totally ordered*, that is, every compound task is decomposed into a *sequence* of subtasks, and 2) flaws in the domain are due to missing primitive actions in methods. Although these two assumptions restrict the application of the proposed approach, we argue that our approach is still of value. One reason is that total order (TO) HTN planning is used more broadly. E.g., in the IPC 2020 and 2023 on HTN Planning, TOHTN benchmarks far outnumbered partial order (PO) ones (in POHTN planning, compound tasks are decomposed into *partially ordered* subtasks). Additionally, adding missing actions to a domain is the most computationally expensive operation (Lin and Bercher 2023) and a basic operation for correcting hierarchical domains on top of which advanced corrections can be built. We will discuss the strength and weakness of our approach in more detail in the discussion section. One remark is that these two assumptions

---

do *not* reduce the complexity of the problem because **NP**-completeness holds in TOHTN planning with only adding actions being allowed (Lin and Bercher 2021).

Our approach solves the problem of finding a minimal set of corrections to a domain by transforming it into another (total order) HTN planning problem such that a *cost optimal* solution to the transformed problem indicates the corrections. We will present how the encoding is done in the following sections. Along the way, we will also introduce some encoding techniques which can be used in broader scenarios, e.g., how to encode a counter in HTN planning.

Related Works

Significant effort has been devoted to developing tools for modeling support for *non-hierarchical* (classical) planning. Muise (2016) developed the successful online planning education platform, Planning.Domains, which supports a wide range of editing features such as syntax highlighting, autocompletion, and tuning an online planner. It provides interfaces for developing customized plugins for advanced features. Strobel and Kirsch (2020) developed a similar tool which provides more advanced features for supporting editing domain files such as stronger capability of syntax highlighting and automatic indentation. itSIMPLE by Vaquero et al. (2013) is another widely used tool which eases modeling domains whose main purpose is to translate a domain described as a UML diagram to a PDDL file. PDDL is the most commonly used language which describes a classical planning domain. See the work done by Haslum et al. (2019) for more details about PDDL. Further, Magnaguagno et al. (2020) developed Web Planner supporting testing like verifying whether a plan is executable.

Apart from those tools providing assistance restricted to supporting creating and editing domain files (i.e., on the syntax level), techniques have also been developed which provide advanced support (i.e., on the semantic level). Sreedharan et al. (2020) adopted the approach for *explainable AI planning* to correct a domain. This approach however works specifically for dialogue domains. Gragera et al. (2023) proposed an approach which fixes a domain by finding missing positive effects in actions. More concretely, they consider errors in a domain which cause a solvable problem becoming unsolvable. Hence, their approach takes as input an unsolvable planning problem and outputs corrections to the domain which add missing positive effects to actions so as to turn the problem into a solvable one. An earlier approach developed by Göbelbecker et al. (2010) also turns an unsolvable planning problem into a solvable one. The difference is that this approach only modifies the initial state of the problem. Lin, Grastien, and Bercher (2023) were concerned with another type of error in a domain which results in the situation that a set of plans which are supposed to be solutions now become infeasible. They thus developed an approach which corrects the domain by turning all those plans into solutions.

Approaches for *learning* planning domains also coincide with modeling assistance. Cresswell, McCluskey, and West (2009), Cresswell and Gregory (2011), Celorrio, Fernández, and Borrajo (2008), Bachor and Behnke (2024), and Li et al. (2024) have developed various approaches which can generate domains automatically from different sources, e.g., state traces and narrative texts.

All the works discussed above work only for *classical* domains. For *hierarchical* ones, most works related to modeling assistance are restricted to investigations into computational complexity (Lin and Bercher 2021, 2023). One practical realization which was not initially developed for modeling assistance for hierarchical planning but can be adapted for this purpose is by Xiao et al. (2020). The approach aims at adding missing actions to methods, which is also what our approach does. Their goal of doing so is to turn an unsolvable planning problem into a solvable one whereas our approach is to turn an infeasible plan into a solution. Their approach relies on preferences given by a human. Thus, the inputs to the approach by Xiao et al. (2020) are an unsolvable hierarchical planning problem and a certain preference provided by a human, similar to the approach by Göbelbecker et al. (2010) for classical planning. Contrastively, our approach takes as input a hierarchical problem and a plan. The approach by McCluskey, Richardson, and Simpson (2002) works for both hierarchical and non-hierarchical planning. It acquires a domain from an action trace. It is however based on a formalism which is rarely used.

Since Höller et al. (2014) have shown that a TOHTN planning problem is basically equivalent to a context-free grammar, any work which is to support constructing context-free grammars can also be adapted to support modeling TOHTN domains, and vice versa. However, to our best knowledge, there exist only few works in this direction, and none of them can do so in a fully automated way. One such example is the work done by Leung, Sarracino, and Lerner (2015), which is to synthesize the syntax of a programming language (which is a context-free grammar) based on human instructions. Our approach can thus also be adapted to fix an incorrect context-free grammar, expanding the applications of our approach.

Preliminary

We first reproduce the TOHTN planning formalism based on the one used by Behnke, Höller, and Biundo (2018). A TOHTN planning problem is a tuple $\mathbf{\Pi} = (\mathcal{D}, s_I, c_I, g)$ where $\mathcal{D} = (\mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{M}, \alpha)$ is the *domain* of $\mathbf{\Pi}$. More specifically, $\mathcal{P}$ is a finite set of *propositions*, $\mathcal{A}$ a set of *primitive* tasks (i.e., actions), $\mathcal{C}$ a set of *compound* tasks, $\mathcal{M}$ a set of *methods*, and $\alpha : \mathcal{A} \to 2^{\mathcal{P}} \times 2^{\mathcal{P}} \times 2^{\mathcal{P}}$ a function mapping each action $a \in \mathcal{A}$ to its *preconditions*, *positive* effects, and *negative* effects, written $\alpha(a) = (\texttt{prec}(a), \texttt{eff}^+(a), \texttt{eff}^-(a))$. $s_I \in 2^{\mathcal{P}}$ and $g \subseteq \mathcal{P}$ are the *initial state* and the *goal* of $\mathbf{\Pi}$, respectively, and $c_I \in \mathcal{C}$ is called the initial task. Since we restrict ourselves to the total order setting, we simply use HTN to replace TOHTN unless otherwise specified.

An action $a \in \mathcal{A}$ is said to be *applicable* in a *state* $s \in 2^{\mathcal{P}}$ *iff* $\texttt{prec}(a) \subseteq s$. Applying an action a in a state s results in a new state s' with $s' = (s \setminus \texttt{eff}^-(a)) \cup \texttt{eff}^+(a)$, written $s \to_a s'$. Further, given two states s and s' and a sequence of actions $\pi = \langle a_1 \cdots a_n \rangle$, we use $s \to_\pi^* s'$ to denote that there exists a sequence of states $\langle s_0 \cdots s_n \rangle$ such that $s_0 = s$, $s' = s_n$, and for each $1 \leq i \leq n$, a_i is applicable in s_{i-1} and $s_{i-1} \to_{a_i} s_i$. That is, s' is obtained by applying π in s.

A method $m = (c, tn)$ decomposes a compound task $c \in \mathcal{C}$ into a *task network* tn which is a *sequence* of primitive and compound tasks (i.e., $tn \in (\mathcal{A} \cup \mathcal{C})^*$ where $*$ is the Kleene star), written as $c \rightarrow_m tn$. The notion of decomposition can be extended from a single compound task to a task network. Concretely, let $tn = \langle t_1 \cdots t_n \rangle$ be a task network in which $n \in \mathbb{N}$ and $t_k \in \mathcal{A} \cup \mathcal{C}$ for each $1 \leq k \leq n$ and $m = (t_i, tn^*)$ a method with $tn^* = \langle t^*_{i_1} \cdots t^*_{i_j} \rangle$ that decomposes a task t_i ($1 \leq i \leq n$) in tn into the task network tn^*, we say that tn is decomposed into another task network tn' by m *iff*

$$tn' = \langle t_1 \cdots t_{i-1} \; t^*_{i_1} \cdots \; t^*_{i_j} \; t_{i+1} \cdots t_n \rangle$$

That is, t_i in tn is replaced by tn^*. For any compound task c or task network tn, we write $c \rightarrow^*_{\overline{m}} tn'$ (*resp*. $tn \rightarrow^*_{\overline{m}} tn'$) to denote that c (*resp*. tn) is decomposed into the task network tn' by a *sequence* of methods $\overline{m}$.

A *solution* to an HTN planning problem is a *primitive* task network (i.e., an action sequence) $\pi = \langle a_1 \cdots a_n \rangle$ such that there is a method sequence $\overline{m}$ with $c_I \rightarrow^*_{\overline{m}} \pi$, and $s_I \rightarrow^*_{\pi} s$ for some s with $g \subseteq s$ (i.e., the goal is satisfied in s). In many scenarios, an action a in a planning problem has certain *cost*. The total cost of an action sequence π is the summation of the cost of every action in it. A *cost-optimal* solution to a planning problem is a solution of the minimal cost, i.e., there exist no other solutions of cost smaller than it.

Fig. 1 shows an example about an HTN planning problem. Each white box is a compound task while each blue box represents a primitive one. The problem has four propositions, $\{p, q, f, r\}$ and four actions, $\{a_1, \cdots, a_4\}$. The preconditions and effects of each action are depicted in the figure. The initial task c_I can be decomposed by solely one method m_I into a sequence of two compound tasks, $\langle c_1 \;\; c_2 \rangle$. c_1 can be decomposed into an action sequence $\langle a_1 \;\, a_2 \rangle$ by the method m_1 while c_2 can be decomposed into either $\langle a_3 \rangle$ or $\langle a_4 \rangle$ by the method m_2 or m_3, respectively. The initial state s_I is $\{p\}$. The goal g is $\{r\}$. The action sequence $\langle a_1 \; a_2 \; a_3 \rangle$ is a solution as it can be obtained by decomposing c_I using the method sequence $\langle m_I \; m_1 \; m_2 \rangle$, and the action sequence is executable and achieves the goal. Note that $\langle a_1 \; a_2 \; a_4 \rangle$ is *not* a solution despite that it can also be obtained by decomposing c_I. It is *not* executable because the precondition of a_4 is not satisfied (p is removed by a_1).

One remark regarding the example is that the method sequence $\langle m_I \; m_2 \; m_1 \rangle$ can also decompose the initial task c_I into $\langle a_1 \; a_2 \; a_3 \rangle$. In fact, both two method sequences which results in the solution are captured by the same decomposition hierarchy shown in Fig. 1. Such a hierarchy is called a *decomposition tree* (Geier and Bercher 2011), which shows how a solution is obtained by decomposing the initial task. One may further notice that for TOHTN planning, a decomposition tree shares a lot of similarities with a *parsing tree* in the context of *context-free grammars* (CFGs). Höller et al. (2014) have shown that a TOHTN planning problem in fact is identical to a CFG in the sense that we could view a primitive task as a terminal symbol, a compound task as a nonterminal symbol, and a method as a production rule.

Having presented the planning formalism, we now formulate the problem of correcting an HTN planning domain. For this purpose, we first define atomic corrections with respect to a domain. Since we make the assumption that flaws in a domain stem from missing actions in methods, we thus only consider corrections that add actions to methods. Formally, let $\mathcal{D}$ be a domain and $m \in \mathcal{M}$ an arbitrary method with $m = (c, tn)$ and $tn = \langle t_1 \cdots t_n \rangle$ for some $n \in \mathbb{N}$, we define $\mathbf{I}[a, m, i]$ with $a \in \mathcal{A}$ and $0 \leq i \leq n$ as a correction which inserts the action a into the position between t_i and t_{i+1} in m. As two special cases, when $i = 0$ or n, the action a will simply be inserted into the position before a_1 or after a_n. As an example, applying the correction $\mathbf{I}[a, m, i]$ will modify the method m to a new one which decomposes the compound task c into the task network $\langle t_1 \cdots t_i \;\; a \;\; t_{i+1} \cdots t_n \rangle$. Let o be an atomic correction with respect to a domain $\mathcal{D}$, we use the notation $\mathcal{D} \Rightarrow_o \mathcal{D}'$ to indicate that $\mathcal{D}'$ is a new domain obtained by applying o to $\mathcal{D}$. The problem of correcting an HTN domain is then defined as follows.

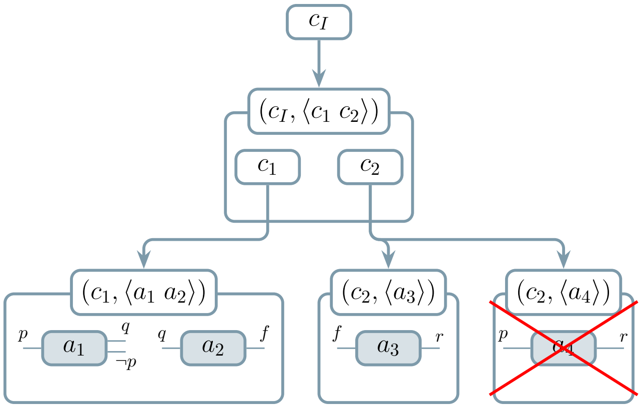

Figure 1: An example of HTN planning. The initial state is $s_I = \{p\}$. The goal is $g = \{r\}$.

Definition 1. Let $\mathbf{\Pi} = (\mathcal{D}, s_I, c_I, g)$ be an HTN planning problem and $\pi = \langle a_1 \cdots a_n \rangle$ an action sequence, the domain correction problem is a tuple $(\mathbf{\Pi}, \pi)$ which is to find an optimal sequence of corrections $\langle o_1 \cdots o_j \rangle$ such that $\mathcal{D} \Rightarrow_{o_1} \mathcal{D}_1 \Rightarrow_{o_2} \cdots \Rightarrow_{o_{j-1}} \mathcal{D}_{j-1} \Rightarrow_{o_j} \mathcal{D}_j$ and π is a solution to the planning problem $\mathbf{\Pi}'$ with $\mathbf{\Pi}' = (\mathcal{D}_j, s_I, c_I, g)$. In particular, by a correction sequence being optimal, we mean that there exists no other correction sequences of smaller length which can turn the action sequence into a solution.

Note that in this paper, we do not consider corrections to actions' preconditions and effects (flaws in actions' preconditions and effects might cause an action sequence not being executable and hence not being a solution). This is however equivalent to correcting a *non-hierarchical* domain (one could identify that correcting actions' preconditions and effects is orthogonal to correcting methods) and has been addressed properly (Lin, Grastien, and Bercher 2023).

Encoding

Now we shift our attention to solving the problem of correcting an HTN domain. We do so by transforming this problem into an HTN planning problem. Intuitively speaking, the transformed problem completes two tasks: 1) It decides what corrections should be made to the domain, and 2) it verifies whether the given action sequence is a solution to the planning problem with the *new* domain. Note that the latter one is the *plan verification* problem for HTN planning from which a reduction to an HTN planning problem (Höller et al. 2022) exists. Hence, our encoding is built on top of theirs while incorporating corrections to the domain. For clarity, we start by reproducing the transformation from the plan verification problem to an HTN problem, and then we introduce how to incorporate corrections into it.

Encoding the Plan Verification Problem

Consider an HTN planning problem $\mathbf{\Pi} = (\mathcal{D}, s_I, c_I, g)$ and an action sequence $\pi = \langle a_1 \cdots a_n \rangle$. Note that π could have duplicate actions, i.e., there exist some $1 \leq i, j \leq n$ with $i \neq j$ and $a_i = a_j$. For simplicity, since we do not consider correcting actions' preconditions and effects, we assume that for any action $a \in \mathcal{A}$, $\texttt{prec}(a) = \texttt{eff}^+(a) = \texttt{eff}^-(a) = \emptyset$, and $g = s_I = \mathcal{P} = \emptyset$. The goal of the encoding is to construct a new HTN planning problem $\mathbf{\Pi}' = (\mathcal{D}', s'_I, c'_I, g')$ such that $\mathbf{\Pi}'$ has a solution *iff* π is a solution to $\mathbf{\Pi}$.

By consulting the solution criteria for HTN planning, one can observe that the core of deciding whether π is a solution to $\mathbf{\Pi}$ is searching for a decomposition hierarchy (a method sequence) decomposing c_I to π. To simulate this search procedure, the constructed problem $\mathbf{\Pi}'$ should preserve the following two properties: 1) $\mathbf{\Pi}'$ has *solely* one solution π', encoding π, and 2) the decomposition hierarchy decomposing c'_I into π' simulates the one that decomposes c_I into π.

To ensure the former, consider an action a in $\mathbf{\Pi}$ with $a = a_i$ for some a_i in π. A primitive task a'_i is constructed for $\mathbf{\Pi}'$, encoding that a appears at the ith position of π. For convenience, we use $\mathbb{A}\llbracket a, i \rrbracket$ to denote a'_i. One could think of $\mathbb{A}\llbracket a, i \rrbracket$ as a function which takes as input a primitive task a in $\mathbf{\Pi}$ and a position i in π and outputs the respective action constructed for $\mathbf{\Pi}'$. For convenience, throughout the paper, we will use $\mathbb{A}\llbracket\cdot\rrbracket$, $\mathbb{C}\llbracket\cdot\rrbracket$, $\mathbb{M}\llbracket\cdot\rrbracket$, and $\mathbb{P}\llbracket\cdot\rrbracket$ to represent the action, the compound task, the method, and the proposition constructed according to certain parameters, respectively. In other words, for each action a in $\mathbf{\Pi}$, a *set* $\mathbf{A}\llbracket a \rrbracket$ of primitive tasks is constructed for $\mathbf{\Pi}'$ such that

$$\mathbf{A}\llbracket a \rrbracket = \{\mathbb{A}\llbracket a, i \rrbracket \mid a = a_i \text{ for some } a_i \text{ in } \pi\}$$

Here we abuse the notation to let $\mathbf{A}\llbracket a \rrbracket$ denote the action set. Furthermore, in order to make the semantics of $\mathbb{A}\llbracket a, i \rrbracket$ hold, additional propositions shall be constructed as $\mathbb{A}\llbracket a, i \rrbracket$'s precondition and effects. More specifically, the action $\mathbb{A}\llbracket a, i \rrbracket$ has solely one precondition $\mathbb{P}\llbracket i-1 \rrbracket$, one positive effect $\mathbb{P}\llbracket i \rrbracket$, and one negative effect $\mathbb{P}\llbracket i-1 \rrbracket$ where the parameters refer to the respective positions in π. The positive effect $\mathbb{P}\llbracket i \rrbracket$ asserts that $\mathbb{A}\llbracket a, i \rrbracket$ occupies the ith position of π while the precondition $\mathbb{P}\llbracket i-1 \rrbracket$ ensures that the $(i-1)$th position of π must already be occupied. The negative effect $\mathbb{P}\llbracket i-1 \rrbracket$ ensures that the $(i+1)$th position must be the next one to be occupied. By letting $s'_I = \{\mathbb{P}\llbracket 0 \rrbracket\}$ and $g' = \{\mathbb{P}\llbracket n \rrbracket\}$, we ensure that the sole solution π' to $\mathbf{\Pi}'$ is that $\pi' = \langle \mathbb{A}\llbracket a_1, 1 \rrbracket \cdots \mathbb{A}\llbracket a_n, n \rrbracket \rangle$.

To ensure the second property, $\mathbf{\Pi}'$ preserves all compound tasks and methods in $\mathbf{\Pi}$ except that for each existing method m, every *action* a in m is replaced with a newly constructed compound task. That is, for each $a \in \mathcal{A}$ in $\mathbf{\Pi}$, a compound task denoted as $\mathbb{C}\llbracket a \rrbracket$ is constructed for $\mathbf{\Pi}'$. One could again view $\mathbb{C}\llbracket a \rrbracket$ as a function which maps the action a in $\mathbf{\Pi}$ to the respective compound task in $\mathbf{\Pi}'$. For convenience, we also define $\mathbb{C}\llbracket c \rrbracket = c$ for each $c \in \mathcal{C}$ in $\mathbf{\Pi}$, meaning that c is preserved in $\mathbf{\Pi}'$. Furthermore, each $\mathbb{C}\llbracket a \rrbracket$ can be decomposed by $|\mathbf{A}\llbracket a \rrbracket|$ methods each of which decomposes $\mathbb{C}\llbracket a \rrbracket$ into an action $\mathbb{A}\llbracket a, i \rrbracket \in \mathbf{A}\llbracket a \rrbracket$. Similarly, we use $\mathbb{M}\llbracket m \rrbracket$ to denote the method in $\mathbf{\Pi}'$ which adheres to the method m in $\mathbf{\Pi}$ and $\mathbb{M}\llbracket a, i \rrbracket$ to denote the one decomposing $\mathbb{C}\llbracket a \rrbracket$ into $\mathbb{A}\llbracket a, i \rrbracket$.

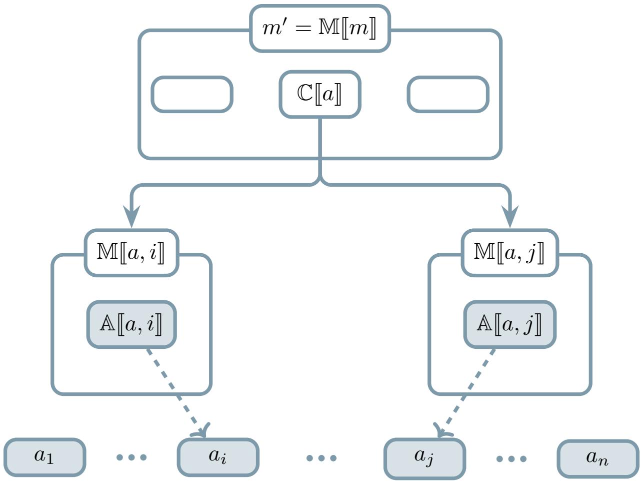

Figure 2: An example about how to transform a plan verification problem to an HTN planning problem.

By construction, if a sequence of methods $\langle m_1 \cdots m_j \rangle$ for $\mathbf{\Pi}$ decomposes c_I into π, the sequence $\langle \mathbb{M}\llbracket m_1 \rrbracket \cdots \mathbb{M}\llbracket m_j \rrbracket \rangle$ then decomposes $\mathbb{C}\llbracket c_I \rrbracket$ into $\langle \mathbb{C}\llbracket a_1 \rrbracket \cdots \mathbb{C}\llbracket a_n \rrbracket \rangle$, which can further be decomposed into $\langle \mathbb{A}\llbracket a_1, 1 \rrbracket \cdots \mathbb{A}\llbracket a_n, n \rrbracket \rangle$, meaning that $\mathbf{\Pi}'$ has a solution *iff* π is a solution to $\mathbf{\Pi}$. For the detailed proof for this fact, see the work by Höller et al. (2022).

In summary, the problem $\mathbf{\Pi}' = (\mathcal{D}', s'_I, c'_I, g')$ with $\mathcal{D}' = (\mathcal{P}', \mathcal{A}', \mathcal{C}', \mathcal{M}', \alpha')$ is constructed as follows:

- $\mathcal{P}' = \{\mathbb{P}\llbracket 0 \rrbracket, \cdots, \mathbb{P}\llbracket n \rrbracket\}$, and $\mathcal{A}' = \bigcup_{a \in \mathcal{A}} \mathbf{A}\llbracket a \rrbracket$.
- $\mathcal{C}' = \{\mathbb{C}\llbracket t \rrbracket \mid t \in \mathcal{A} \cup \mathcal{C}\}$.
- $\mathcal{M}' = \{\mathbb{M}\llbracket m \rrbracket \mid m \in \mathcal{M}\} \cup \mathcal{M}^\dagger$ with $\mathcal{M}^\dagger$ being the set $\{\mathbb{M}\llbracket a, i \rrbracket \mid a = a_i \text{ for some } a_i \text{ in } \pi\}$.
- $s'_I = \{\mathbb{P}\llbracket 0 \rrbracket\}$, $g' = \{\mathbb{P}\llbracket n \rrbracket\}$, and $c'_I = \mathbb{C}\llbracket c_I \rrbracket$.

Fig. 2 shows an example about how to revive a method m in the original given HTN problem, assuming that m initially has three tasks where the second one is an action a. It is thus replaced with the compound task $\mathbb{C}\llbracket a \rrbracket$. Further, we assume that the ith and the jth action in π are a, that is, $a_i = a_j = a$. As a result, $\mathbb{C}\llbracket a \rrbracket$ can be decomposed by two methods. The first one decomposes it into the action $\mathbb{A}\llbracket a, i \rrbracket$ while the other decomposes it into $\mathbb{A}\llbracket a, j \rrbracket$. $\mathbb{A}\llbracket a, i \rrbracket$ and $\mathbb{A}\llbracket a, j \rrbracket$ can match a_i and a_j, respectively, meaning that the *original* action a in the method m can be either a_i or a_j.

Incorporating Corrections

Having presented how to encode the plan verification as an HTN problem, we now move on to incorporate domain corrections into the encoding, which is our main contribution.

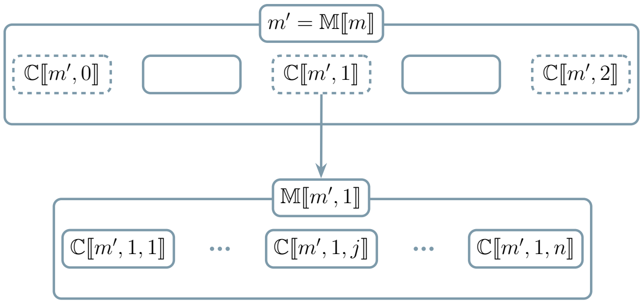

Figure 3: An example about how to revive a method to control action insertions.

Let $\mathbf{\Pi}' = (\mathcal{D}', s'_I, c'_I, g')$ with $\mathcal{D}' = (\mathcal{P}', \mathcal{A}', \mathcal{C}', \mathcal{M}', \alpha')$ be the HTN problem which encodes the problem of deciding whether a plan $\pi = \langle a_1 \cdots a_n \rangle$ is a solution to an HTN problem $\mathbf{\Pi} = (\mathcal{D}, s_I, c_I, g)$ with $\mathcal{D} = (\mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{M}, \alpha)$. Our goal is to make some further constructions to $\mathbf{\Pi}'$ to simulate making corrections to $\mathbf{\Pi}$. Since we only consider adding missing actions, our new constructions are thus restricted to methods $\mathbb{M}[\![m]\!] \in \mathcal{M}'$ with $m \in \mathcal{M}$. This is because $\mathbb{M}[\![m]\!]$ is basically a copy of m, and hence, adding actions to $\mathbb{M}[\![m]\!]$ already simulates adding actions to m.

Consider a method $\mathbb{M}[\![m]\!]$ decomposing a compound task into a task network $\langle t_1 \cdots t_k \rangle$. We first observe that there are $k+1$ blocks to which we can add actions, namely, any block between the tasks t_{i-1} and t_i for some $1 < i \leq k$ together with the two that are before t_1 and after t_k. The fundamental idea of simulating action insertions is constructing one compound task for each such blocks, controlling which actions should be added to the respective block. We use $b[m', i]$ with $m' = \mathbb{M}[\![m]\!]$ to denote a block (i.e., the ith block in the method m') and $\mathbb{C}[\![m', i]\!]$ to denote the compound task constructed for the respective block.

Concretely, each $\mathbb{C}[\![m', i]\!]$ is placed at the respective block in the method m'. The control over action insertions is done by decomposing $\mathbb{C}[\![m', i]\!]$ into a task network that consists of n *compound* tasks, $\langle \mathbb{C}[\![m', i, 1]\!] \cdots \mathbb{C}[\![m', i, n]\!] \rangle$, where each $\mathbb{C}[\![m', i, j]\!]$ ($1 \leq j \leq n$) is to decide whether the action a_j in π is inserted into the block $b[m', i]$. We use $\mathbb{M}[\![m', i]\!]$ to denote the method that decomposes $\mathbb{C}[\![m', i]\!]$. Fig. 3 depicts an example about how to revive a method $m' = \mathbb{M}[\![m]\!]$ with $m \in \mathcal{M}$ for some m. m' initially contains two tasks (represented as solid boxes), meaning that there are three blocks into which actions can be inserted. We thus construct three compound tasks $\mathbb{C}[\![m', i]\!]$, $0 \leq i \leq 2$, and put them into the respective places.

In order to encode the situation that a_j is added to $b[m', i]$, we construct a method $\mathbb{M}[\![m', i, j, +]\!]$ which decomposes $\mathbb{C}[\![m', i, j]\!]$ into the *compound* task $\mathbb{C}[\![a_j]\!]$. Intuitively speaking, the reason for using $\mathbb{C}[\![a_j]\!]$ here is that the action a_j in $\mathbf{\Pi}$ is represented as $\mathbb{C}[\![a_j]\!]$ in $\mathbf{\Pi}'$. We will explain this in more detail later on. Similarly, in order to encode that a_j is not inserted, we construct another method $\mathbb{M}[\![m', i, j, -]\!]$ which decomposes $\mathbb{C}[\![m', i, j]\!]$ into an empty task network.

Methods $\mathbb{M}[\![m', i, j, \{+, -\}]\!]$ with $m' = \mathbb{M}[\![m]\!]$ for some method m in $\mathbf{\Pi}$ are however not constructed adequately. The reason is that in the scenario of correcting the domain of $\mathbf{\Pi}$, if an action is inserted to the method m which occurs more than once in decomposition, the insertion must take effects for all occurrences of m. The analogy of this constraint for $\mathbf{\Pi}'$ is that the consequence of adding (or not adding) an action to m' must be consistent among all occurrences of m'.

To achieve such consistency, we construct one more *primitive* task $\mathbb{A}[\![m', i, j, +]\!]$ for $\mathbb{M}[\![m', i, j, +]\!]$ and put it in front of the task $\mathbb{C}[\![a_j]\!]$. Similarly, we construct another primitive task $\mathbb{A}[\![m', i, j, -]\!]$ and place it into $\mathbb{M}[\![m', i, j, -]\!]$. The action $\mathbb{A}[\![m', i, j, +]\!]$ has one precondition, $\mathbb{P}[\![m', i, j, +]\!]$, one *negative* effect, $\mathbb{P}[\![m', i, j, -]\!]$, and no positive effects. In contrast, $\mathbb{A}[\![m', i, j, -]\!]$ has $\mathbb{P}[\![m', i, j, -]\!]$ as its precondition while it deletes $\mathbb{P}[\![m', i, j, +]\!]$. We put these two propositions into the initial state. By construction, if for the first occurrence of m' in some method sequence, $\mathbb{M}[\![m', i, j, +]\!]$ is applied to decompose $\mathbb{C}[\![m', i, j]\!]$ (which encodes adding a_j to the block $b[m', i]$), then for other occurrences of m', $\mathbb{M}[\![m', i, j, -]\!]$ is not available because the precondition of $\mathbb{A}[\![m', i, j, -]\!]$ has been removed, ensuring the consistency. The construction of these actions as well as the structure of the methods $\mathbb{M}[\![m', i, j, \{+, -\}]\!]$ is shown in Fig. 4. At this moment, one could ignore the proposition $\mathbb{P}'$, the action $\mathbb{A}'$, and the method $\mathbb{M}'$ in Fig. 4. We will describe the construction and the purpose of those components shortly. The compound task $\mathbb{C}[\![\mathcal{U}]\!]$ in the method $\mathbb{M}[\![m', i, j, +]\!]$ represents a *counter* which can count up to a certain bound $\mathcal{U}$. The purpose of the counter is to make our constructed HTN problem more easy to be solved. The detailed implementation of the counter will be introduced in the next section.

The fact that the method m' can be used more than once in decomposition is another reason for why using $\mathbb{C}[\![a_j]\!]$ in $\mathbb{M}[\![m', i, j, +]\!]$. The action a_j inserted might also be used to match another action a_k in π with $a_k = a_j$ and $k > j$. For such a case, $\mathbb{C}[\![a_j]\!]$ can be decomposed into $\mathbb{A}[\![a_j, j]\!]$ for the first occurrence of m' and $\mathbb{A}[\![a_k, k]\!]$ for the other.

Taken together, the HTN problem $\mathbf{\Pi}^* = (\mathcal{D}^*, s^*_I, c^*_I, g^*)$ with $\mathcal{D}^* = (\mathcal{P}^*, \mathcal{A}^*, \mathcal{C}^*, \mathcal{M}^*, \alpha^*)$ which encodes the problem of correcting an HTN domain is as follows:

- $\mathcal{P}^* = \mathcal{P}' \cup \mathcal{P}^\dagger$ where $\mathcal{P}^\dagger$ is the set consisting of propositions $\mathbb{P}[\![m', i, j, o]\!]$ with $m' = \mathbb{M}[\![m]\!]$ for some $m \in \mathcal{M}$, $1 \leq j \leq n$, $o \in \{+, -\}$, and $0 \leq i \leq |m'| + 1$. Here, $|m'|$ is the length of the task sequence resulting from m'.
- $\mathcal{A}^*$ is the union of $\mathcal{A}'$ and $\mathcal{A}^\dagger$ where $\mathcal{A}^\dagger$ is the set of actions $\mathbb{A}[\![m', i, j, o]\!]$ whose parameters are under the same constraints as those of $\mathbb{P}[\![m', i, j, o]\!]$ defined above.
- $\mathcal{C}^* = \mathcal{C}' \cup \mathcal{B} \cup \mathcal{S}$ where $\mathcal{B}$ is the set of compound tasks $\mathbb{C}[\![m', i]\!]$, and $\mathcal{S}$ consisting of tasks $\mathbb{C}[\![m', i, j]\!]$. m', i, and j all follow the same constraints as above.
- $\mathcal{M}^* = \mathcal{M}' \cup \mathcal{M}'' \cup \mathcal{M}^\dagger$ where $\mathcal{M}''$ is the set of methods each of which decomposes a compound task $\mathbb{C}[\![m', i, j]\!]$, and $\mathcal{M}^\dagger$ contains the methods $\mathbb{M}[\![m', i, j, o]\!]$ with the parameters following the same constraints as above.
- $s^*_I = s' \cup \mathcal{I}$ with $\mathcal{I} = \mathcal{P}^\dagger$, $c^*_I = c'_I$, and $g^* = g'$.

The construction ensures that the plan π can be turned into a solution to $\mathbf{\Pi}$ by correcting the domain *iff* the problem $\mathbf{\Pi}^*$ is *solvable*. This relies on the fact that for any method sequence $\overline{m}$ with respect to $\mathbf{\Pi}^*$ that decomposes c^*_I into a solution, methods $\mathbb{M}[\![m', i, j, +]\!]$ and $\mathbb{M}[\![m', i, j, -]\!]$ for any

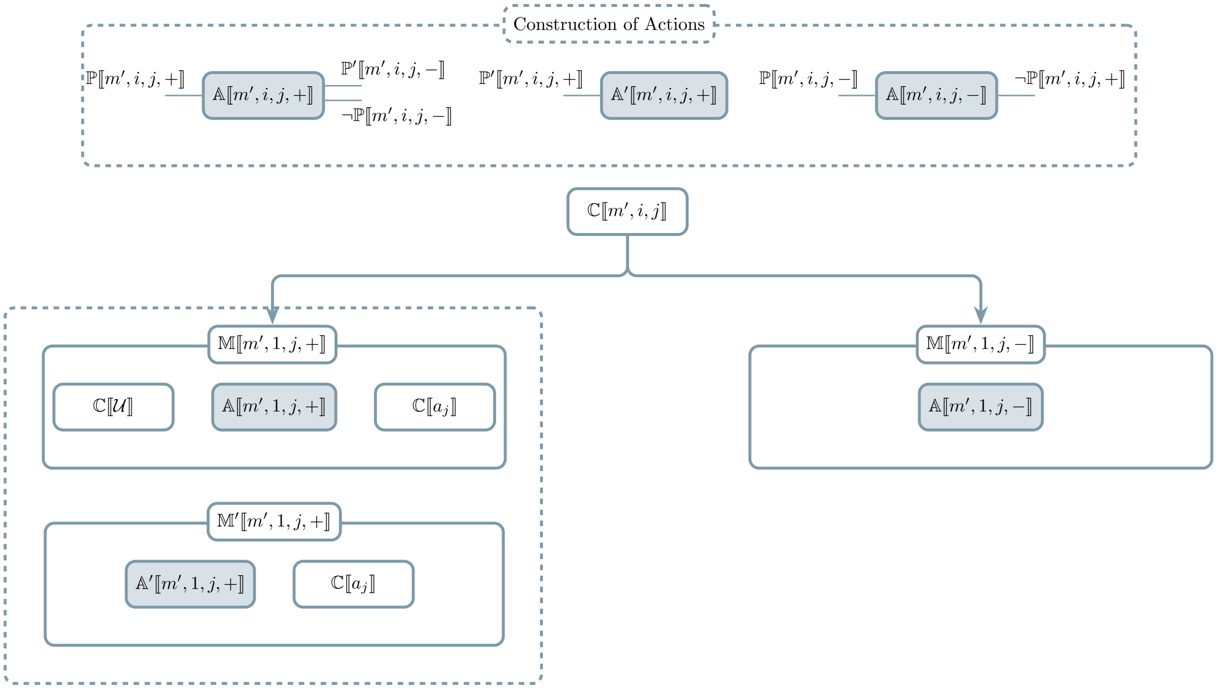

Figure 4: An example of the construction that incorporates action insertions.

m', i, and j cannot exist simultaneously in $\overline{m}$ together with the fact that $\mathbf{\Pi}'$ encodes the plan verification problem.

Theorem 1. $\mathbf{\Pi}^*$ *is solvable iff* π *can be turned into a solution to* $\mathbf{\Pi}$ *by adding actions to methods in* $\mathbf{\Pi}$.

However, we are looking for an *optimal* sequence of corrections. To this end, we want to assign cost to the actions in $\mathbf{\Pi}^*$ such that the cost optimal solution to $\mathbf{\Pi}^*$ indicates the optimal corrections. One naive attempt is letting each $\mathbb{A}[\![m', i, j, +]\!]$ have cost one, representing the cost of adding an action. The remaining actions all have zero cost. This is however not enough. If an action is added to a method m' while m' is applied multiple times, $\mathbb{A}[\![m', i, j, +]\!]$ associated with this insertion will occur multiple times, meaning that the cost of this insertion is also counted more than once.

To address this problem, for each $\mathbb{M}[\![m', i, j, +]\!]$, we construct a new method $\mathbb{M}'[\![m', i, j, +]\!]$ which is identical to the original one except that the action $\mathbb{A}[\![m', i, j, +]\!]$ is replaced by a new one $\mathbb{A}'[\![m', i, j, +]\!]$. This action has zero cost. Both positive and negative effects of it is empty. Its sole precondition is a new proposition $\mathbb{P}'[\![m', i, j, +]\!]$, which is also added to the positive effects of $\mathbb{A}[\![m', i, j, +]\!]$. As a result, the method $\mathbb{M}'[\![m', i, j, +]\!]$ can be applied *iff* $\mathbb{M}[\![m', i, j, +]\!]$ has been used previously. Since $\mathbb{A}'[\![m', i, j, +]\!]$ has zero cost, this construction ensures that the cost of every action insertion is counted only once. Consequently, the cost optimal solution to the problem $\mathbf{\Pi}^*$ with this new construction reveals the optimal correction to the domain of $\mathbf{\Pi}$.

Counter

The construction presented previously already encodes the domain correction problem. We could further improve it by providing the *maximal* number of action insertions allowed. If we could incorporate this bound into our construction, an HTN problem solver (which is used to solve the constructed problem) can exploit this information to significantly reduce the search space by *not* adding any further action when the number of insertions has reached the bound.

One could immediately observe that given a domain correction problem $(\mathbf{\Pi}, \pi)$ with $\mathbf{\Pi} = (\mathcal{D}, s_I, c_I, g)$ and $\pi = \langle a_1 \cdots a_n \rangle$, the maximal number $\mathcal{U}$ of action insertions to $\mathcal{D}$ is $n - \gamma(c_I)$ where $\gamma(c_I)$ represents the *minimal* number of actions that can be obtained from c_I. For if we add more actions than that bound, then any primitive task network obtained from the initial task has actions which outnumbers the given plan π, meaning that π can never be obtained.

Proposition 1. *Let* $(\mathbf{\Pi}, \pi)$ *be a domain correction problem. The number of actions inserted to* $\mathbf{\Pi}$ *turning* π *into a solution cannot exceed* $|\pi| - \gamma(c_I)$ *where* $|\pi|$ *is the length of* π, c_I *is the initial task of* $\mathbf{\Pi}$, *and* $\gamma(c_I)$ *is the minimal number of actions that can be obtained from* c_I.

The computation for the number $\gamma(c_I)$ has already been studied by some existing work (Bercher et al. 2017). Therefore, we will skip the detailed computation here but focus on how to incorporate this bound $\mathcal{U}$ into our construction.

For this, we need a *counter* that can count up to $\mathcal{U}$. Such a counter is represented as a compound task $\mathbb{C}[\![\mathcal{U}]\!]$. There are $\mathcal{U}$ methods that can decompose $\mathbb{C}[\![\mathcal{U}]\!]$, each of which decomposes it into an action $\mathbb{A}[\![i, \mathcal{U}]\!]$ $(1 \leq i \leq \mathcal{U})$. We use $\mathbb{M}[\![i]\!]$ to denote the method decomposing $\mathbb{C}[\![\mathcal{U}]\!]$ into $\mathbb{A}[\![i, \mathcal{U}]\!]$. $\mathbb{A}[\![i, \mathcal{U}]\!]$ can be interpreted as counting to the ith step. $\mathbb{A}[\![i, \mathcal{U}]\!]$ has one single proposition $\mathbb{P}[\![i-1, \mathcal{U}]\!]$ as its precondition, meaning that the $(i-1)$th step has been *counted*. Similarly, $\mathbb{A}[\![i, \mathcal{U}]\!]$

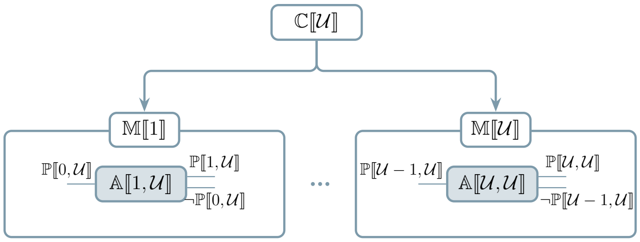

Figure 5: The construction of a counter.

has one single positive effect $\mathbb{P}\llbracket i,\mathcal{U}\rrbracket$, indicating that the step i has been counted. $\mathbb{A}\llbracket i,\mathcal{U}\rrbracket$ also removes $\mathbb{P}\llbracket i-1,\mathcal{U}\rrbracket$, which ensures that the counter can only count incrementally. Fig. 5 illustrates the implementation of a counter.

To incorporate the bound $\mathcal{U}$ into the construction, we simply place the counter $\mathbb{C}\llbracket\mathcal{U}\rrbracket$ at the beginning of *every* method $\mathbb{M}\llbracket m',i,j,+\rrbracket$. This thus ensures that when an action is inserted, the counter will increase by 1, and when the counter reaches $\mathcal{U}$, no further methods $\mathbb{M}\llbracket m',i,j,+\rrbracket$ can be applied, i.e., no further actions can be inserted. Note that the counter $\mathbb{C}\llbracket\mathcal{U}\rrbracket$ is *not* added to methods $\mathbb{M}'\llbracket m',i,j,+\rrbracket$, ensuring that one action insertion will not be counted multiple times.

Empirical Evaluation

In this section, we present the experimental results with respect to our approach. The two metrics we used to evaluate the performance of the approach are the runtime for correcting a domain and the coverage on the benchmark set (i.e., the percentage of the instances that can be solved in the benchmark set). The reason for using these two is that in the real scenario of modeling support, the time required for finding corrections is the most critical factor because when deploying our approach into practice, it can be called iteratively and interactively. In every iteration, the user can decide whether the found corrections address the errors successfully. If that is not the case, the user can rule out those unsatisfactory corrections and instruct our approach to block them in the next iteration. This continues until all errors are addressed. Note that we did not compare our approach with the one by Xiao et al. (2020) because, as mentioned in the related work section, the inputs to these two approaches are different.

Configuration

The experiments ran on an Intel Cascade Lake CPU, with 20-minutes timeout. However, there existed no benchmark sets of flawed hierarchical domains at the time when we conducted the empirical evaluation, and hence, we had to construct a novel benchmark set.

The procedure for constructing the benchmark set is as follows. We drew 200 TOHTN planning problem instances together with the respective solutions from 11 domains from the IPC 2020 on HTN Planning benchmark set. We emphasize that the meaning of the term "domain" here is different from a domain $\mathcal{D}$ defined in the TOHTN planning formalism. The term "domain" here refers to the name of a class of HTN planning problems that model problems in the same scenario. For instance, the domain *Rover* refers to the class of planning problems in the scenario of Mars exploration. To distinguish these two concepts, we will use the term "*domain name*" to refer to the name of a class of planning problems. Note that the instances drawn in this step are *unflawed*. Next, for each drawn problem instance $\mathbf{\Pi} = (\mathcal{D}, s_I, c_I, g)$ with $\mathcal{D} = (\mathcal{P}, \mathcal{A}, \mathcal{C}, \mathcal{M}, \alpha)$, we let every *action* in every method $m \in \mathcal{M}$ have a 30% chance of being removed. This thus randomly introduced errors to the domain $\mathcal{D}$. Lastly, we discarded those instances to which no errors were introduced. This left 167 instances with flawed domains in total.

	Total	**Solved**	**Plan Length**	
			Min	Max
Hiking	20	0	26	45
Transport	20	0	16	50
Entertainment	10	10	24	50
Rover	20	13	16	49
Monroe (FO)	20	9	3	48
Depots	20	4	15	50
Woodworking	9	7	4	24
Satellite	13	10	12	50
Blocksworld	5	1	21	40
Monroe (PO)	20	5	11	48
Childsnack	10	3	50	50
	167	**62**		

Table 1: The experimental results for correcting flawed domains.

Experimental Results

The experimental results (Lin, Höller, and Bercher 2024) are summarized in Tab. 1. The left-most column lists the domain names. The columns labeled with "total" and "solved" show the total number of instances and of solved instances. The two columns under the name "plan length" report the minimum and the maximal length of provided plans. As can be seen from the table, our approach solved around 37.12% instances, i.e., it can provide optimal corrections to 37.12% flawed domains. Our approach performed badly on the domains Hiking and Transport. For the former, the reason is that the domain already contains numerous methods. Our approach thus creates an HTN problem of large size which is hard to be solved by an HTN planner. For the Transport domain, it is not clear what causes the bad performance. We hypothesize that it is due to some specific structure, e.g., cyclic structure, within the domain.

Fig. 6 shows the runtime for solving each instance against the plan length, including those unsolved instances whose runtime exceeded the timeout, i.e., each point in the plot represents a domain correction problem instance.

Discussions and Future Work

In this paper we assume that errors in a domain only attribute to missing actions, causing the limitation of our approach that corrections are restricted to inserting actions. In practice, there are more types of corrections that could be considered, e.g., adding methods/compound tasks and deleting actions/methods/compound tasks. One remark is that inserting

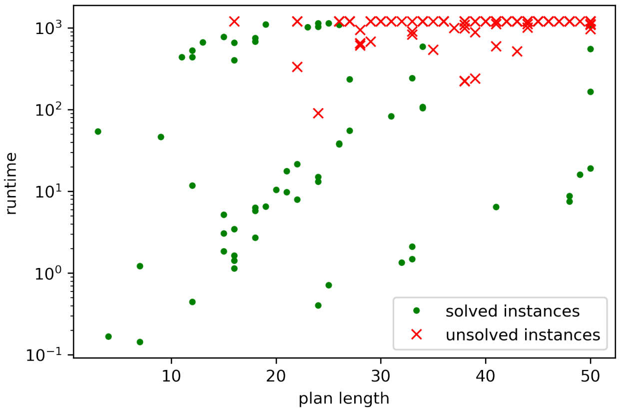

Figure 6: Runtime for correcting flawed domains.

actions can serve as a basis for many of those corrections. E.g., the treatment for inserting compound tasks is similar to inserting actions while adding methods can be viewed as a two-step process: We first create an empty method and then insert actions and compound tasks into it. Consequently, we plan in our future work to implement all those corrections so that our approach can be deployed more broadly.

Additionally, our approach only works for *grounded* HTN planning. The *lifted* formalism is however more widely used in the context of domain modeling. The grounded formalism is defined in terms of *propositional* logic whereas the lifted one is defined in *first-order* logic. Hence, the lifted formalism is a generalization of the grounded one, meaning that all corrections which are meaningful in the grounded setting are also relevant to the lifted one whereas the lifted formalism can have additional corrections, e.g., correcting the arguments of a primitive task, a compound task, or a method. An important future work is thus to generalize our approach so that it can work for the lifted setting.

We have mentioned before that correcting primitive tasks' preconditions and effects is also an important branch of correcting a planning domain. This task has been addressed by Lin, Grastien, and Bercher (2023). However, there exist hierarchical planning formalisms which support methods/compound tasks' preconditions and effects. Fixing these components is also an important aspect of fixing domains.

Another limitation is that some corrections returned might not be *desired* by the modeler. To address this problem, we want to employ an iterative process involving human-in-the-loop. Concretely, our approach is invoked iteratively. On each iteration, the user could decide which corrections returned by our approach are desired and which are not. The next iteration will then start by adapting those desired corrections and forbidding those undesired ones. This process keeps running until the domain is completely fixed.

Our evaluation shows that our approach only solved less than 50% problems in the evaluation. To improve the performance, we consider transforming the problem to SAT or ILP because they have the same complexity as the domain correction problem, i.e., $\mathbf{NP}$-complete (whereas TOHTN planning is $\mathbf{EXPTIME}$-complete (Alford, Bercher, and Aha 2015)), which might have more efficient solving techniques given the specific problem.

Conclusion

We presented an approach for correcting a TOHTN domain with missing actions by compiling such a problem to a planning problem. We are the first to study such an approach for hierarchical planning, making it an important contribution toward modeling supports for hierarchical planning. Our approach can also be applied to a wide range of applications because of the connection between a TOHTN problem and a context-free grammar.

References

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight Bounds for HTN Planning. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling, ICAPS 2015*, 7–15. AAAI.

Bachor, P.; and Behnke, G. 2024. Learning Planning Domains from Non-Redundant Fully-Observed Traces: Theoretical Foundations and Complexity Analysis. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence, AAAI 2024*. AAAI.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – Totally-Ordered Hierarchical Planning Through SAT. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, 6110–6118. AAAI.

Bercher, P.; Alford, R.; and 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*, 6267–6275. IJCAI.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An Admissible HTN Planning Heuristic. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI 2017*, 480–488. IJCAI.

Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence*, 129(1-2): 5–33.

Celorrio, S. J.; Fernández, F.; and Borrajo, D. 2008. The PELA Architecture: Integrating Planning and Learning to Improve Execution. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence, AAAI 2008*, 1294–1299. AAAI.

Cresswell, S.; and Gregory, P. 2011. Generalised Domain Model Acquisition from Action Traces. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011*, 42–49. AAAI.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2009. Acquisition of Object-Centred Domain Models from Planning Examples. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009*, 338–341. AAAI.

Geier, T.; and 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*, 1955–1961. IJCAI.

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

Göbelbecker, M.; Keller, T.; Eyerich, P.; Brenner, M.; and Nebel, B. 2010. Coming Up With Good Excuses: What to do When no Plan Can be Found. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010*, 81–88. AAAI.

Gragera, A.; Fuentetaja, R.; Ángel García-Olaya; and Fernández, F. 2023. A Planning Approach to Repair Domains with Incomplete Action Effects. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling, ICAPS 2023*, 153–161. AAAI.

Haslum, P.; Muise, C.; Magazzeni, D.; and Lipovetzky, N. 2019. *An Introduction to the Planning Domain Definition Language*. Springer.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M.; and Lasinger, H. 2010. The Scanalyzer Domain: Greenhouse Logistics as a Planning Problem. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010*, 234–237. AAAI.

Hoffmann, J. 2011. Analyzing Search Topology Without Running Any Search: On the Connection Between Causal Graphs and h+. *Journal of Artificial Intelligence Research*, 41: 155–229.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In *The 21st European Conference on Artificial Intelligence, ECAI 2014*, 447–452. IOS.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the Expressivity of Planning Formalisms through the Comparison to Formal Languages. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling, ICAPS 2016*, 158–165. AAAI.

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

Höller, D.; Wichlacz, J.; Bercher, P.; and Behnke, G. 2022. Compiling HTN Plan Verification Problems into HTN Planning Problems. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling, ICAPS 2022*, 145–150. AAAI.

Karpas, E.; and Magazzeni, D. 2020. Automated Planning for Robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 3: 417–439.

Leung, A.; Sarracino, J.; and Lerner, S. 2015. Interactive Parser Synthesis by Example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015*, 565–574. ACM.

Li, R.; Cui, L.; Lin, S.; and Haslum, P. 2024. NaRuto: Automatically Acquiring Planning Models from Narrative Texts. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence, AAAI 2024*. AAAI.

Lin, S.; and Bercher, P. 2021. Change the World – How Hard Can that Be? On the Computational Complexity of Fixing Planning Models. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI 2021*, 4152–4159. IJCAI.

Lin, S.; and Bercher, P. 2022. On the Expressive Power of Planning Formalisms in Conjunction with LTL. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling, ICAPS 2022*, 231–240. AAAI.

Lin, S.; and Bercher, P. 2023. Was Fixing this *really* That Hard? On the Complexity of Correcting HTN Domains. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI 2023*, 12032–12040. AAAI.

Lin, S.; Grastien, A.; and Bercher, P. 2023. Towards Automated Modeling Assistance: An Efficient Approach for Repairing Flawed Planning Domains. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI 2023*, 12022–12031. AAAI.

Lin, S.; Höller, D.; and Bercher, P. 2024. Experimental Results for the SoCS 2024 Paper: "Modeling Assistance for Hierarchical Planning: An Approach for Correcting Hierarchical Domains with Missing Actions". doi: 10.5281/zenodo.10946945.

Magnaguagno, M. C.; Pereira, R. F.; Móre, M. D.; and Meneguzzi, F. 2020. Web Planner: A Tool to Develop, Visualize, and Test Classical Planning Domains. In *Knowledge Engineering Tools and Techniques for AI Planning*, 209–227. Springer.

McCluskey, T. L.; Richardson, N. E.; and Simpson, R. M. 2002. An Interactive Method for Inducing Operator Descriptions. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems, AIPS 2002*, 121–130. AAAI.

Muise, C. 2016. Planning.Domains. In *The 26th International Conference on Automated Planning and Scheduling – Demonstrations*.

Sreedharan, S.; Chakraborti, T.; Muise, C.; Khazaeni, Y.; and Kambhampati, S. 2020. – D3WA+ – A Case Study of XAIP in a Model Acquisition Task for Dialogue Planning. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling, ICAPS 2020*, 488–498. AAAI.

Strobel, V.; and Kirsch, A. 2020. MyPDDL: Tools for Efficiently Creating PDDL Domains and Problems. In *Knowledge Engineering Tools and Techniques for AI Planning*, 67–90. Springer.

Vaquero, T. S.; Silva, J. R.; Tonidandel, F.; and Beck, J. C. 2013. itSIMPLE: Towards an Integrated Design System for Real Planning Applications. *Knowledge Engineering Review*, 28(2): 215–230.

Xiao, Z.; Wan, H.; Zhuo, H. H.; Herzig, A.; Perrussel, L.; and Chen, P. 2020. Refining HTN Methods via Task Insertion with Preferences. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence, AAAI 2020*, 10009–10016. AAAI.