Told You That Will Not Work: Optimal Corrections to Planning Domains Using Counter-Example Plans

Songtuan Lin[1], **Alban Grastien**[2], **Rahul Shome**[1], **Pascal Bercher**[1]

[1]School of Computing, The Australian National University, Canberra, Australia
[2]Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
{songtuan.lin, rahul.shome, pascal.bercher}@anu.edu.au, alban.grastien@cea.fr

Abstract

Hardness of modeling a planning domain is a major obstacle for making automated planning techniques accessible. We developed a tool that helps modelers correct domains based on available information such as the known feasibility or infeasibility of certain plans. Designing model repair strategies that are capable of repairing flawed planning domains automatically has been explored in previous work to use *positive* plans (invalid in the given (flawed) domain but feasible in the "true" domain). In this work, we highlight the importance of and study *counter-example negative* plans (*valid* in the given (flawed) domain but *infeasible* in the "true" domain). Our approach automatically corrects a domain by finding an optimal repair set to the domain which turns all negative plans into non-solutions, in addition to making all positive plans solutions. Experiments indicate strong performance in the fast-downward benchmark suite with random errors. A handcrafted benchmark with domain flaws inspired by some practical applications also motivates the method's efficacy.

Introduction

Planning is concerned with finding a sequence of actions that achieves a certain goal. Many theoretical investigations (e.g., see the work by Bylander (1994), by Erol, Nau, and Subrahmanian (1995), and by Erol, Hendler, and Nau (1996)) have been done and many practical approaches (e.g., see the work by Bonet and Geffner (2001), by Hoffmann and Nebel (2001), and by Helmert (2006)) have been developed for planning. The applications of planning are however still rare in practice, especially for areas outside academia. We argue that this is because modeling a practical problem as a planning problem is difficult and error-prone, which requires a modeler to be an expert not only in planning but in the area related to the problem to be modeled. As a result, tools for providing modeling support are needed for the purpose of making planning techniques more accessible.

The functionalities of existing tools for modeling assistance range from providing support for *editing* files which describe planning problems (Magnaguagno et al. 2020; Muise 2016; Strobel and Kirsch 2020; Vaquero et al. 2013) to more advanced features like automatically fixing errors in a planning domain (Gragera et al. 2023; Lin, Grastien, and Bercher 2023; Sreedharan et al. 2020; Lin, Höller, and Bercher 2024).

In particular, motivated by the scenario where there can be *positive* plans – *feasible* plans that might be *invalid* in a *flawed* domain – Lin, Grastien, and Bercher (2023) proposed an approach for correcting flaws by finding *repairs* to the domain turning all positive plans into solutions. Repairs to a domain refer to modifications to actions' preconditions and effects, which fully characterize a domain. While this scenario has its applications (in particular since modeling errors can lead to the exclusion of intended solutions, which in an extreme case may even render problems unsolvable (Gragera et al. 2023)), the opposite situation may also arise. That is, due to modeling mistakes, models may allow the generation of plans that are solutions in these (flawed) models while they are inapplicable in the real world (i.e., the "true" model). We call such plans *(counter-example) negative* plans. We argue that in many scenarios, negative plans are as vital as positive ones because they reflect real world constraints that are not adequately modeled, causing plans to work in the model but not the real world. In motivating applications like robotics, domain models are often used as abstractions for tasks (Dantam et al. 2018; Garrett et al. 2021) but are flawed when compared to a true domain containing motion planning feasibility (Kavraki and LaValle 2008), i.e., feasible plans in the domain might be infeasible for the robot to execute. This paper serves as a promising step towards an effective model repair strategy to *correct flawed domains using counter-examples of both negative and positive plans*.

The key contribution of the paper is an approach which takes as input a planning domain, a set of positive as well as negative plans, and outputs a *cardinality-minimum* set of repairs to the domain which turn all positive plans into solutions and all negative ones into non-solutions. We want to emphasize that our technique is highly general in the sense that it supports repairing *lifted* domains (in contrast to ground planning, where the similar problem is conceptually much simpler (Erol, Nau, and Subrahmanian 1995; Erol, Hendler, and Nau 1996)) and supports repairing both negative and positive preconditions, as well as negative and positive effects – all of which can be specified in the prominent PDDL language (Fox and Long 2003) (although we are not able to change the parameter list of actions). Finding a minimum repair set favors preserving as much information

as possible in the given domain. Further, we assume that, for each negative plan, the action that is inapplicable is given explicitly. We claim that this is a reasonable setting that arises in some motivating applications like robotics where a plan is simulated or executed on a robot. For instance, in task and motion planning, each action motion feasibility is checked using a motion planner (i.e., when the robot is unable to pick up an object, the corresponding `pick` action fails).

Experimental results demonstrate strong empirical performance of the proposed approach across International Planning Competition (IPC) domains with randomized errors. Specifically, repairs found by the approach in the experiments have high *precision*, namely, most of them can successfully fix flaws in a domain. In particular, the repairs were successful in a reasonable time. We also created benchmarks by introducing bespoke flaws to BlocksWorld and Gripper domains. These scenarios are inspired by challenges often introduced in motivating applications in robotics (Dantam et al. 2018; Garrett et al. 2021). Flaws in those domains may occur in specific ways, which is what we tried to emulate in the motivating bespoke benchmarks. The proposed approach could recover *all* introduced flaws, showing that our approach might be a feasible first step towards future robotics applications.

The rest of the paper is organized as follows: We first introduce the planning formalism we used and the formalization of the domain repair problem. After that, we discuss the technical details of our approach. Lastly, we present our experimental results.

Planning Formalism

We consider a *lifted* formalism. A lifted planning *domain* $\mathcal{D}$ is defined over a set $\mathcal{V}$ of variables and is a tuple $(\mathcal{P}, \mathcal{A})$. $\mathcal{P}$ is a set of *predicates*. Each predicate $\mathbf{p} \in \mathcal{P}$ is of the form $P(v_1|_{t_1}, \cdots, v_n|_{t_n})$ for some $n \in \mathbb{N}_0$ ($\mathbb{N}_0 = \mathbb{N} \cup \{0\}$) in which P is the *unique* name of the predicate, and for each $1 \leq i \leq n$, $v_i \in \mathcal{V}$ is a variable, and t_i is the *type* associated with the variable v_i. $v_i|_{t_i}$ represents the restriction that the variable v_i can only be mapped to an *object* of the type t_i. We will discuss this in more detail later on. $\mathcal{A}$ is a set of *action schemas*. Similar to a predicate, an action schema $\mathbf{a} \in \mathcal{A}$ is again of the form $A(v_1|_{t_1}, \cdots, v_n|_{t_n})$ ($n \in \mathbb{N}_0$) where A is the unique action name. An action schema $\mathbf{a}$ is characterized by its positive preconditions, negative preconditions, positive effects, and negative effects, written $\texttt{prec}^+(\mathbf{a})$, $\texttt{prec}^-(\mathbf{a})$, $\texttt{eff}^+(\mathbf{a})$, and $\texttt{eff}^-(\mathbf{a})$, respectively, each of which is a set of predicates. We can view each $\texttt{prec}^+$, $\texttt{prec}^-$, $\texttt{eff}^+$, and $\texttt{eff}^-$ as a *function* mapping each action schema to the respective set of predicates.

Definition 1. *Let* $\mathbf{a} = A(v_1|_{t_1}, \cdots, v_n|_{t_n})$ *be an action schema and* $\mathbf{p} = P(v'_1|_{t'_1}, \cdots, v'_m|_{t'_m})$ *a predicate. We say that* $\mathbf{p}$ *is compatible with* $\mathbf{a}$ *if for any* $1 \leq i \leq m$*, there exists a* j *with* $1 \leq j \leq n$ *such that* $v'_i = v_j$ *and* $t'_i = t_j$*. In other words, every parameter of* $\mathbf{p}$ *is also a parameter of* $\mathbf{a}$*.*

In particular, if a predicate $\mathbf{p}$ is in an action schema $\mathbf{a}$'s preconditions or effects, then $\mathbf{p}$ must be compatible with $\mathbf{a}$.

A planning *task* $\mathcal{T}$ with respect to a planning domain is a tuple $(\mathcal{O}, s_I, g)$. $\mathcal{O}$ is a finite set of *objects*. Each object $o \in \mathcal{O}$ has a *type*. For convenience, we use $\mathbf{t}[o]$ to indicate the type of the object o. A predicate $\mathbf{p} = P(v_1|_{t_1}, \cdots, v_n|_{t_n})$ in $\mathcal{P}$ can be *grounded* to a *proposition* p by a *variable substitution* function $\varrho : \mathcal{V} \to \mathcal{O}$ which maps each v_i ($1 \leq i \leq n$) to an object, written $p = \varrho[\![\mathbf{p}]\!]$. In particular, we say that p is a *valid grounding* of $\mathbf{p}$ under ϱ *iff* for each i with $1 \leq i \leq n$, $t_i = \mathbf{t}[\varrho[\![v_i]\!]]$. Similarly, an action schema $\mathbf{a} \in \mathcal{A}$ can also be grounded to an *action* a by a variable substitution function ϱ, written $a = \varrho[\![\mathbf{a}]\!]$. Note that when an action schema is grounded with a variable substitution function ϱ, all predicates in its precondition and effects are also grounded under ϱ simultaneously. We again use $\texttt{prec}^+(a)$, $\texttt{prec}^-(a)$, $\texttt{eff}^+(a)$, and $\texttt{eff}^-(a)$ to denote the positive preconditions, negative preconditions, positive effects, and negative effects of a grounded action a, respectively. s_I and g are called the *initial state* and the *goal description* each of which is a set of *propositions*. A planning *problem* $\mathbf{\Pi}$ is a pair $(\mathcal{D}, \mathcal{T})$ of a domain and a task. Note that a domain can be paired with different tasks, resulting in different planning problems.

In planning, a *state* is a set of propositions. A grounded action a is said to be *applicable* in a state s *iff* $\texttt{prec}^+(a) \subseteq s$ and $\texttt{prec}^-(a) \cap s = \emptyset$. If action a is applicable in state s, then applying a in s results in a new state $s' = (s \setminus \texttt{eff}^-(a)) \cup \texttt{eff}^+(a)$. We use $s \to_a s'$ to indicate that a is applicable in s, and s' is obtained by applying a in s. A plan (i.e., an action sequence) $\pi = \langle a_1 \cdots a_n \rangle$ is a *solution* to a planning problem *iff* for each $1 \leq i \leq n$, there exists an action schema $\mathbf{a} \in \mathcal{A}$ and a valid variable substitution function ϱ with $a_i = \varrho[\![\mathbf{a}]\!]$, and there exists a state sequence $\langle s_0 \cdots s_n \rangle$ such that $s_0 = s_I$, $g \subseteq s_n$, and $s_{i-1} \to_{a_i} s_i$ for each $1 \leq i \leq n$.

Problem Formulation

Next we formalize the domain repair problem. The formalization is based on the one by Lin, Grastien, and Bercher (2023) to a large extent, who formalized the problem for positive plans only. We add the capability to express and repair negative plans. Since we want to repair a flawed domain, we first define the notion of *atomic repairs*.

Definition 2. *Let* $\mathcal{D}$ *be a planning domain. We define an atomic repair as* $\mathbf{r}[\![\mathbf{a}, \mathbf{p}, \mathtt{c}, \mathtt{op}]\!]$ *where* $\mathbf{a}$ *is an action schema,* $\mathbf{p}$ *is a predicate which is compatible with* $\mathbf{a}$*,* $\mathtt{c} \in \{\texttt{prec}^+, \texttt{prec}^-, \texttt{eff}^+, \texttt{eff}^-\}$*, and* $\mathtt{op} \in \{+, -\}$*. The restriction on the parameters of* $\mathbf{p}$ *requires that every parameter of* $\mathbf{p}$ *is also a parameter of* $\mathbf{a}$*.*

The interpretation of an atomic repair $\mathbf{r}[\![\mathbf{a}, \mathbf{p}, \mathtt{c}, \mathtt{op}]\!]$ is as follows: $\mathbf{a}$ is the action schema that is to be modified. $\mathbf{p}$ is the predicate that is to be inserted into (if $\mathtt{op} = +$) or removed from (if $\mathtt{op} = -$) the respective precondition or effects of $\mathbf{a}$ controlled by $\mathtt{c}$. As a special case, if r is such that 1) $\mathtt{op} = -$ and $\mathbf{p}$ is *not* in the respective component of $\mathbf{a}$ specified by $\mathtt{c}$, or 2) $\mathtt{op} = +$ and $\mathbf{p}$ is already in the component $\mathtt{c}$ of $\mathbf{a}$, then $\mathtt{c}(\mathbf{a})$ is not changed.

When applying an atomic repair r to an action schema $\mathbf{a}$, what is actually being modified is the output of the function $\texttt{prec}^+$, $\texttt{prec}^-$, $\texttt{eff}^+$, or $\texttt{eff}^-$ on the input $\mathbf{a}$. That is, if $\mathbf{a}'$ is obtained by applying a repair $r = \mathbf{r}[\![\mathbf{a}, \mathbf{p}, \mathtt{c}, \mathtt{op}]\!]$ to an action schema $\mathbf{a}$, we in fact still have $\mathbf{a}' = \mathbf{a}$, because the

action schema itself is not modified, while the value of $\mathtt{c}(\mathbf{a})$ is now changed by r. This observation is crucial because it witnesses the validity of having multiple repairs working on the same action schema.

Given two domains $\mathcal{D}$ and $\mathcal{D}'$ together with an atomic repair $r = \mathbf{r}[\![\mathbf{a}, \mathbf{p}, \mathtt{c}, \mathtt{op}]\!]$, we use the notation $\mathcal{D} \Rightarrow_r \mathcal{D}'$ to denote that $\mathcal{D}'$ is obtained by applying r to $\mathcal{D}$.

Next we define how a domain can be modified by a *set* δ of atomic repairs. To this end, we first define a *valid* set of repairs with respect to a domain as follows.

Definition 3. *A repair set δ with respect to a domain is* valid *if and only if there does* not *exist two repairs $r, r' \in \delta$ such that one* undoes *the effect of the other. Two repairs r and r' with $r = \mathbf{r}[\![\mathbf{a}, \mathbf{p}, \mathtt{c}, \mathtt{op}]\!]$ and $r' = \mathbf{r}[\![\mathbf{a}', \mathbf{p}', \mathtt{c}', \mathtt{op}']\!]$* undo *the effect of each other if $\mathbf{a} = \mathbf{a}'$, $\mathbf{p} = \mathbf{p}'$, $\mathtt{c} = \mathtt{c}'$, and $\mathtt{op} \neq \mathtt{op}'$.*

It is easy to verify that the following proposition holds:

Proposition 1. *Let $\mathcal{D}$ be a domain and δ a valid repair set to $\mathcal{D}$. Applying the repairs in δ in an arbitrary order results in the same modified domain $\mathcal{D}'$.*

This result thus allows us to use the notation $\mathcal{D} \Rightarrow_\delta^* \mathcal{D}'$ to indicate that the domain $\mathcal{D}'$ is obtained from $\mathcal{D}$ by applying a *valid* repair set δ. Lastly, we could formulate our extended domain repair problem.

Definition 4. *We define the domain repair problem as a tuple $\mathbf{R} = (\mathcal{D}, \mathbb{T})$ where $\mathcal{D}$ is a planning domain, and $\mathbb{T}$ is a set $\{\mathbf{T}_1, \cdots, \mathbf{T}_n\}$ for some $n \in \mathbb{N}$. Each $\mathbf{T}_i$ is a tuple $(\mathcal{T}_i, \mathbb{P}_i, \mathbb{E}_i)$ where $\mathcal{T}_i$ is a planning task defined with respect to the domain $\mathcal{D}$, $\mathbb{P}_i$ is a set of positive plans with respect to the planning problem $\mathbf{\Pi}_i = (\mathcal{D}, \mathcal{T}_i)$, and $\mathbb{E}_i$ is a set of tuples $\{(\pi_1^-, i_1), \cdots, (\pi_j^-, i_j)\}$ for some $j \in \mathbb{N}$ where for each k with $1 \leq k \leq j$, π_k^- is a negative plan (i.e., a solution) to $\mathbf{\Pi}_i$, and i_k is a number smaller or equal to the length of π_k^-.*

A solution to $\mathbf{R}$ is a valid repair set δ with $\mathcal{D} \Rightarrow_\delta^* \mathcal{D}'$ such that for all $1 \leq i \leq n$, every plan $\pi^+ \in \mathbb{P}_i$ is a solution to the problem $\mathbf{\Pi}_i' = (\mathcal{D}', \mathcal{T}_i)$, and every plan $\pi_k^- \in \mathbb{E}_i$ satisfies the following criterion: It is not *a solution to $\mathbf{\Pi}_i'$, and its i_kth action is the first inapplicable action.*

As mentioned earlier, in this paper, we are aiming at finding an *optimal* solution (i.e., a *cardinality-minimum* set of repairs) to a domain repair problem, which is also what Lin, Grastien, and Bercher (2023) did in their work. Note that there can exist *more than one* optimal solution to a domain repair problem. Furthermore, it has been shown by Lin and Bercher (2021) that the task of finding an optimal solution to a domain repair problem with *only* positive plans is already **NP**-complete in the *grounded* setting (a grounded planning problem is a special case of a planning problem in which every predicate and every action schema are 0-arity). Here, we would like to strengthen this result by showing that even with *only* negative plans, it is also **NP**-complete to find an optimal solution to a *grounded* domain repair problem.

To this end, we first rephrase the task of finding an optimal solution to a domain repair problem as a decision problem called k-bounded domain repair problem as follows:

Definition 5. *Let $\mathbf{R}$ be a domain repair problem and $k \in \mathbb{N}$, the k-bounded domain repair problem is to decide whether there exists a solution δ to $\mathbf{R}$ with $|\delta| \leq k$.*

Theorem 2. *The k-bounded domain repair problem is **NP**-complete in the grounded setting. This holds even in the case where no positive plans are given.*

Proof. Membership: Let $\mathbf{R} = (\mathcal{D}, \{\mathbf{T}_1, \cdots, \mathbf{T}_n\})$ be a domain repair problem with $\mathbf{T}_i = (\mathcal{T}_i, \emptyset, \mathbb{E}_i)$ for each $1 \leq i \leq n$. We can *guess* and *verify* a repair set δ to decide whether it is a solution to $\mathbf{R}$. Note that $|\delta|$ is bounded by the *minimal* number between k and κ in which κ is the total number of repairs needed to empty *all* actions' effects and to add *all* propositions to *all* actions' preconditions. Note that κ is a polynomial with respect to the encoding size of $\mathbf{R}$ because $\mathcal{D}$ is *grounded*, and hence, δ can be guessed in polynomial time. Since verifying whether every negative plan is not a solution in $\mathcal{D}'$ with $\mathcal{D} \Rightarrow_\delta^* \mathcal{D}'$ can also be done in polynomial time, membership thus follows.

Hardness: We reduce from the *minimal hitting set* problem, which is to decide, given a set of *sets* $\Gamma = \{\gamma_1, \cdots, \gamma_n\}$ and a $q \in \mathbb{N}$, whether there exists a set θ with $|\theta| \leq q$ such that $\theta \cap \gamma \neq \emptyset$ for all $\gamma \in \Gamma$. Let $\Omega = \bigcup_{\gamma \in \Gamma} \gamma$. To encode this problem, we construct the domain $\mathcal{D} = (\mathcal{P}, \mathcal{A})$ as follows: For each element $e \in \Gamma$, we construct a *proposition* p_e so that $\mathcal{P} = \{p_e \mid e \in \Omega\} \cup \{p_g\}$ where p_g is just a dummy proposition. $\mathcal{A}$ is consisted of sole one action a which has *no* preconditions and negative effects and has only one positive effect p_g. Then for each $\gamma_i \in \Gamma$, we construct a $\mathbf{T}_i = (\mathcal{T}_i, \emptyset, \mathbb{E}_i)$ with $\mathcal{T}_i = (\emptyset, \{p_e \mid e \in \Omega \setminus \gamma_i\}, \{p_g\})$ and $\mathbb{E}_i = \{(\langle a \rangle, 1)\}$, namely, each negative plan solely consists of the action a. Note that for each problem $(\mathcal{D}, \mathcal{T}_i)$, to make a be inapplicable in the initial state, *at least one* proposition from the set $\{p_e \mid e \in \gamma_i\}$ must be inserted to the preconditions of a. This thus simulates the constraint that at least one element in γ_i must be hitted. Thus, Γ has a hitting set θ with $|\theta| \leq q$ *iff* the constructed domain repair problem has a solution δ with $|\delta| \leq q$. Hardness thus follows. □

Since the grounded setting is a special case of the lifted one, we can obtain **NP**-hardness of the problem in the lifted setting as a simple corollary. Note that **NP**-membership however might not hold in the lifted setting because when inserting a predicate to an action schema, there is an exponential number of mappings between the predicate's parameters and the action schema's parameters. Concretely, for an n-arity action schema $\mathbf{a}$ and an m-arity predicate $\mathbf{p}$, there are $\mathcal{O}(m^n)$ possibilities to add $\mathbf{p}$ to $\mathbf{a}$ if all the parameters of $\mathbf{a}$ and of $\mathbf{p}$ are of the same type.

Corollary 3. *The k-bounded domain repair problem in the lifted setting is **NP**-hard and is in* $\mathbf{NEXPTIME}$.

Repairing Flawed Domains

We move on to discuss how to find an optimal solution to the domain repair problem. Since we will extend the approach by Lin, Grastien, and Bercher (2023) (which can only deal with positive plans), we shall first briefly present the basic idea of their approach.

Basic Algorithm The approach is based upon the *diagnosis* algorithm (Reiter 1987; Slaney 2014). Specifically, given

Algorithm 1: A general template for the algorithm finding a cardinality-minimum set of repairs.

Input: A domain repair problem $\mathbf{R} = (\mathcal{D}, \mathbb{T})$
Output: A cardinality-minimal set of repairs δ to $\mathcal{D}$

1: $\Phi \leftarrow \emptyset$ ▷ The set of computed conflicts
2: **while** True **do**
3: $\quad \delta \leftarrow$ a minimum hitting set of Φ
4: $\quad$ **if** δ is a solution to $\mathbf{R}$ **then**
5: $\quad\quad$ **return** δ
6: $\quad$ ▷ Procedure for computing new conflicts
7: $\quad \Theta \leftarrow$ ExtractConflicts(δ, $\mathbf{R}$)
8: $\quad \Phi \leftarrow \Phi \cup \Theta$
9: **procedure** ExtractConflicts(δ, $\mathbf{R}$)
10: $\quad \Theta \leftarrow \emptyset$
11: $\quad$ **for all** $\mathbf{R}_s = (\mathcal{D}, (\mathcal{T}, \{\pi^+\}, \emptyset))$ in $\mathbf{R}$ **do**
12: $\quad\quad$ ▷ Computing a conflict for a positive plan
13: $\quad\quad (\phi, \varphi) \leftarrow$ ConfPos(δ, $\mathbf{R}_s$)
14: $\quad\quad \Theta \leftarrow \Theta \cup \{(\phi, \varphi)\}$
15: $\quad$ **for all** $\mathbf{R}_s = (\mathcal{D}, (\mathcal{T}, \emptyset, \{(\pi^-, j)\}))$ in $\mathbf{R}$ **do**
16: $\quad\quad$ ▷ Computing a conflict for a negative plan
17: $\quad\quad (\phi, \varphi) \leftarrow$ ConfNeg(δ, $\mathbf{R}_s$)
18: $\quad\quad \Theta \leftarrow \Theta \cup \{(\phi, \varphi)\}$
19: $\quad$ **return** Θ

a domain repair problem (with *only* positive plans), the algorithm finds a *cardinality-minimum* repair set $\delta^* \subseteq \Delta$ (where Δ is the set of all atomic repairs) that can turn all positive plans into solutions. It does so by iteratively computing a set Φ of *simple* and *complex conflicts*. A simple conflict (Reiter 1987) is a set φ of repairs such that if none of them are applied, then at least one positive plans will *not* be a solution. That is, at least one repair in φ must be applied in order to solve the problem. A complex conflict (Struss and Dressler 1989) is a tuple (ϕ, φ) each of which is a set of repairs. The interpretation of a complex conflict is that if *all* repairs in ϕ are executed, at least one repair in φ should be applied in order to solve the problem. We could view a simple conflict φ as a special case of a complex conflict (ϕ, φ) with $\phi = \emptyset$. Thus, for convenience, we will use the term *conflicts* to refer to complex conflicts in the paper. It follows from the definition of conflicts that the cardinality-minimum set δ^* must be a *minimum hitting set* of Φ computed by the algorithm. A minimum hitting set δ^* of a set of conflicts Φ is such that for any $(\phi, \varphi) \in \Phi$, if $\phi \subseteq \delta^*$, then $\delta^* \cap \varphi \neq \emptyset$.

The presented idea leads to a general template of the algorithm for solving the domain repair problem, which is shown by Alg. 1. The algorithm maintains a set Φ of computed conflicts, which is empty at the beginning. On each iteration, the algorithm computes a minimum hitting set δ of Φ. If δ turns out to be a solution to the domain repair problem, the procedure terminates. Otherwise, it computes new conflicts (based on the current δ) and adds them to Φ.

The set δ computed (line 3) is called a *candidate*. The procedure for computing new conflicts (line 7) takes as input the current candidate δ. The reason for this is that we want to exploit δ to let the procedure be more informed about which repairs are required. We will elaborate on this when we discuss how to compute new conflicts.

We can use the same algorithm template (Alg. 1) to deal with the domain repair problem with negative plans. The key for this is the improved procedure for computing conflicts *wrt*. not only positive plans but negative ones.

Computing Conflicts We first extend the notion of conflicts to adapt negative plans. Intuitively speaking, we want an extended conflict (ϕ, φ) to be such that if all repairs in ϕ are applied, then *at least one* in φ must also be applied in order to turn all positive plans into solutions and all negative ones into non-solutions.

Definition 6. *Let $\mathbf{R} = (\mathcal{D}, \mathbb{T})$ be a domain repair problem. A conflict for $\mathbf{R}$ is a tuple (ϕ, φ) each of which is a repair set such that for an arbitrary repair set δ, if $\phi \subseteq \delta$ and $\delta \cap \varphi = \emptyset$, then one of the following holds:*

1) *There is a $\pi^+ \in \mathbb{P}_i$ for some $(\mathcal{T}_i, \mathbb{P}_i, \mathbb{E}_i) \in \mathbb{T}$ such that π^+ is* not *a solution to the planning problem $(\mathcal{D}', \mathcal{T}_i)$ where $\mathcal{D} \Rightarrow_\delta^* \mathcal{D}'$.*
2) *There exists a negative plan π^- with $(\pi^-, j) \in \mathbb{E}_i$ for some $(\mathcal{T}_i, \mathbb{P}_i, \mathbb{E}_i) \in \mathbb{T}$ such that the jth action in π^- is* not *the first* inapplicable *action with respect to the planning problem $(\mathcal{D}', \mathcal{T}_i)$ where $\mathcal{D} \Rightarrow_\delta^* \mathcal{D}'$.*

Bearing this extended definition, the next result follows.

Corollary 4. *Let $\mathbf{R}$ be a domain repair problem, Θ an arbitrary set of conflicts for $\mathbf{R}$, and $\delta = \{r_1, \cdots, r_n\}$ an optimal solution to $\mathbf{R}$. It holds that δ is a minimum hitting set of Θ.*

This result thus ensures the soundness of Alg. 1 when it is used to solve the domain repair problem with both positive and negative plans. Since Alg. 1 is *not* a decision procedure, it will *not* terminate if the problem is not solvable.

For the purpose of computing conflicts, one crucial observation is that a conflict for a *subproblem* $(\mathcal{D}, (\mathcal{T}_i, \{\pi^+\}, \emptyset))$ with $\pi^+ \in \mathbb{P}_i$ or $(\mathcal{D}, (\mathcal{T}_i, \emptyset, \{(\pi^-, j)\}))$ with $(\pi^-, j) \in \mathbb{E}_i$, is also a conflict for the original problem. As a consequence, we could compute conflicts for a domain repair problem by computing a conflict for each of its subproblems, or in other words, computing a conflict for each of the given plans.

Conflicts for Positive Plans The procedure for computing a conflict for a positive plan has been presented by Lin, Grastien, and Bercher (2023) (i.e., computing a conflict for a sub-problem $\mathbf{R}_s = (\mathcal{D}, (\mathcal{T}_i, \{\pi^+\}, \emptyset))$). Thus, we omit the details and use it as a black-box procedure here (i.e., the procedure ConfPos in line. 13 of Alg. 1). The procedure takes as input a candidate repair set δ and a sub-problem $\mathbf{R}_s$ containing a positive plan and outputs a conflict.

Conflicts for Negative Plans We focus on how to compute a conflict with respect to a negative plan, i.e., dealing with a sub-problem $(\mathcal{D}, (\mathcal{T}_i, \emptyset, \{(\pi^-, i)\}))$, provided a candidate repair set δ. For convenience, we let $\pi^- = \langle a_1 \cdots a_n \rangle$ for some $n \geq i$. The basic idea for this is as follows: We first compute the domain $\mathcal{D}'$ with $\mathcal{D} \Rightarrow_\delta^* \mathcal{D}'$. After that, we compute a repair set φ for the *updated* domain $\mathcal{D}'$ from which at least one must be applied to make the ith action in π^- be the *first* inapplicable action. Lastly, we compute a set of repairs ϕ which cause the application of a repair in φ. The

intuition for this procedure is that we could view the candidate set δ as an attempt to fixing some errors in the original domain $\mathcal{D}$. Thus, a repair set (i.e., φ) to the updated domain $\mathcal{D}'$ which contributes to making the ith action in π^- be the first one that is inapplicable strongly indicates which repairs are further required.

More concretely, one could observe that applying the candidate δ to the domain $\mathcal{D}$ could result in one of the following two consequences: 1) There exists another action *before* the ith one in π^- that is inapplicable, or 2) the entire prefix $\langle a_1 \cdots a_i \rangle$ of π^- is executable. For the former, we could simply view the subsequence $\langle a_1 \cdots a_{i-1} \rangle$ as a positive plan and then exploit the approach by Lin, Grastien, and Bercher (2023) to compute a conflict. We focus on how to compute a conflict (ϕ, φ) for the latter case.

The core of the procedure is to compute the set φ. To this end, we observe that in order to turn a_i into an inapplicable action (where a_i is the ith action in π^-), we need to find repairs which can make the following condition hold: There exists a proposition in the precondition of a_i which is *not* satisfied in the state where a_i is executed. In order to achieve this, we could consider a set of repairs φ_p which can make a specific proposition p become unsatisfied. The set φ is thus the union of the sets φ_p for *all* possible propositions p. This idea is implemented in Alg. 2 in which the function EnumRepairs serves the purpose of enumerating all repairs that can make p unsatisfied. In particular, p could be unsatisfied in either the *positive* preconditions or the *negative* preconditions. In Alg. 2, we differentiate these two cases by having an additional parameter in the function EnumRepairs, controlling the repair set computed which can make p unsatisfied in the respective component. However, the treatments for the two cases are totally symmetric. Therefore, in the rest part of this section, we only focus on discussing the implementation of EnumRepairs for positive preconditions.

Let p be an arbitrary proposition and s_i the state with respect to the *updated* domain $\mathcal{D}'$ where a_i is executed. Depending on whether p is in s_i and whether p is in the (positive) precondition of a_i, the treatment for the computation for φ_p is different. We elaborate on these cases as follows.

Case 1: When $p \in \mathtt{prec}^+(a_i)$ and $p \in s_i$, φ_p is the set of repairs each of which can remove p from s_i. One can observe that φ_p is the union of two sets, φ_p^+ and φ_p^-. The repairs in φ_p^+ prevent p from being added to some state before s_i while those in φ_p^- delete p from a previous state of s_i. Concretely, let a_j be the action with the *largest* index j such that $j < i$, $p \in \mathtt{eff}^+(a_j)$, and for every k satisfying $j < k < i$, $p \notin \mathtt{eff}^-(a_k)$. φ_p^+ is the set of repairs $\mathbf{r}[\![\mathbf{a}, \mathbf{p}, \mathtt{eff}^+, -]\!]$ where $\varrho[\![\mathbf{a}]\!] = a_j$ and $\varrho[\![\mathbf{p}]\!] = p$ for some ϱ. To compute the set φ_p^-, for each a_k, $j < k < i$, we compute the set of repairs $\mathbf{r}[\![\mathbf{a}, \mathbf{p}, \mathtt{eff}^-, +]\!]$ with $\varrho[\![\mathbf{a}]\!] = a_k$ and $\varrho[\![\mathbf{p}]\!] = p$ for some ϱ. φ_p^- is thus the union of all these sets.

Case 2: The second case is that $p \notin \mathtt{prec}^+(a_i)$ while $p \notin s_i$. The repair set that addresses this case is the set of repairs $\mathbf{r}[\![\mathbf{a}, \mathbf{p}, \mathtt{prec}^+, +]\!]$ with $\varrho[\![\mathbf{a}]\!] = a_i$ and $\varrho[\![\mathbf{p}]\!] = p$ for some ϱ. That is, the consequence of applying any of those repairs is having p in a_i's preconditions.

Algorithm 2: Computing a conflict for a negative plan.

```
1:  procedure ConfNeg(δ, R_s)
2:      ▷ R_s = (D, (T, ∅, {(π⁻, i)}))
3:      D' ← D' with D ⇒*_δ D'
4:      if a_i is not the first inapplicable action then
5:          π' ← ⟨a_1, ··· , a_{i−1}⟩
6:          return ConfPos(δ, (D, (T, {π'}, ∅)))
7:      φ ← ∅
8:      for all propositions p do
9:          ▷ repairs for making p unsatisfied in prec⁺
10:         φ⁺ ← EnumRepairs(p, prec⁺)
11:         ▷ repairs for making p unsatisfied in prec⁻
12:         φ⁻ ← EnumRepairs(p, prec⁻)
13:         φ ← φ ∪ (φ⁺ ∪ φ⁻)
14:     φ ← ∅
15:     for all r ∈ φ do
16:         if r undoes some r' ∈ δ then
17:             φ, φ ← φ ∪ {r'}, φ \ {r}
18:     return (φ, φ)
```

Case 3: The last case, which is the most complicated one, is when $p \notin \mathtt{prec}^+(a_i)$ but $p \in s_i$. The complexity of computing the repair set φ_p addressing this case stems from the constraint that we need to insert p to $\mathtt{prec}^+(a_i)$ and remove p from s_i *simultaneously*. The former can be achieved by a repair from the set computed for the *second* case while the latter one can be done by a repair from the set computed for the *first* case. It thus implies that we need to apply those two types of repairs simultaneously. Formally, we have that

$$\varphi_p = \{r \circ r' \mid r \in \varphi_1, r' \in \varphi_2\}$$

where φ_1, φ_2 refer to the sets computed for the first and second cases, respectively, and $\circ$ indicates that r and r' are applied simultaneously. It however does *not* align with the notion of conflicts because by definition, a conflict should be a set of repairs at least one of which must be applied. Here, φ_p is a set of conjunctions of two repairs. Fortunately, we could simply let $\varphi_p = \varphi_2$ (in other words, the set computed for this third case is identical to the one computed for the second case). The reason for this is that the fact that at least one *pair* of r and r' must be executed implies that at least one r' must be applied.

Note that we also have the last case where $p \in \mathtt{prec}^+(a_i)$ and $p \notin s_i$. This is the case where a_i is already not applicable, meaning that we do not need to do any computation.

The implementation of the procedure is shown in Alg. 3. Note that lines 3 to 4 illustrate how finding repairs for positive preconditions and for negative ones are symmetric.

The set φ is the union of all the sets φ_p The computation for the set ϕ (the conditional part of the conflict) is done in the same way as that for dealing with a positive plan. That is, we iterate through every $r \in \delta$. If there exists a $r' \in \varphi$ that undoes r, we remove r' from φ and add r to ϕ.

Finally, we prove that Alg. 2 indeed computes a conflict. For this, we first observe the following lemma, which holds because EnumRepairs is an exhaustive search procedure.

Lemma 5. *If the prefix $\langle a_1 \cdots a_i \rangle$ of π^- is executable, then*

Algorithm 3: Computing the set φ_p for a proposition p.

procedure ENUMREPAIRS(p, $\mathtt{prec}$)
  if $\mathtt{prec} = \mathtt{prec}^+$ **then**
    $\mathtt{c}, \mathtt{c}', \mathtt{c}'' \leftarrow \mathtt{prec}^+, \mathtt{eff}^+, \mathtt{eff}^-$
  else $\mathtt{c}, \mathtt{c}', \mathtt{c}'' \leftarrow \mathtt{prec}^-, \mathtt{eff}^-, \mathtt{eff}^+$
  if $p \in \mathtt{c}(a_i)$ and $p \in s_i$ **then**
    $\varphi_p^- \leftarrow \emptyset$
    for $k = i-1, \cdots, 1$ **do**
      $\mathbf{a}, \varrho \leftarrow \mathbf{a}, \varrho$ with $\varrho[\![\mathbf{a}]\!] = a_k$
      if $p \in \mathtt{c}'(a_k)$ **then**
        $\varphi_p^+ \leftarrow \{\mathbf{r}[\![\mathbf{a}, \mathbf{p}, \mathtt{c}', -]\!] \mid \varrho[\![\mathbf{p}]\!] = p\}$
        break
      $\varphi_p^- \leftarrow \{\mathbf{r}[\![\mathbf{a}, \mathbf{p}, \mathtt{c}'', +]\!] \mid \varrho[\![\mathbf{p}]\!] = p\} \cup \varphi_p^-$
    $\varphi_p \leftarrow \varphi_p^+ \cup \varphi_p^-$
  else if $p \notin \mathtt{c}(a_i)$ **then**
    $\mathbf{a}, \varrho \leftarrow \mathbf{a}, \varrho$ with $\varrho[\![\mathbf{a}]\!] = a_i$
    $\varphi_p \leftarrow \{\mathbf{r}[\![\mathbf{a}, \mathbf{p}, \mathtt{c}, +]\!] \mid \varrho[\![\mathbf{p}]\!] = p\}$
  return φ_p

line 8 to 13 in Alg. 2 compute a set of repairs from which at least one must be applied in order to turn a_i into the first inapplicable action.

Based on this result, we could show that Alg. 2 computes a conflict, which thus entails the correctness of our approach.

Theorem 6. *Given a candidate set of repairs δ, Alg. 2 computes a conflict (ϕ, φ).*

Proof. We only need to consider the case where after applying δ, the prefix $\langle a_1 \cdots a_i \rangle$ is executable. Suppose $\delta \subseteq \phi$ and $\delta \cap \varphi = \emptyset$. Then $\delta \cap (\varphi \cup \phi^{-1}) = \emptyset$ where ϕ^{-1} is the set of repairs which undo the consequence of those repairs in ϕ. This holds because $\delta \cap \phi^{-1} = \emptyset$, which follows from the fact that $\delta \subseteq \phi$. Furthermore, $\varphi \cup \phi^{-1}$ is the set computed by line 8 to 13 in Alg. 2. Now we consider a repair $r \in \delta$ which makes a proposition in a_i's precondition become unsatisfied. The consequence of r must *not* happen in $\mathcal{D}'$ because otherwise $\varphi \cup \phi^{-1}$ will not be computed. It however leads to a contradiction that $r \in \varphi \cup \phi^{-1}$ because of Lem. 5. This concludes our proof. □

Example We use an example to illustrate how conflicts are computed for negative plans where all action schemas and predicates have 0 arity (i.e., the domain is *propositional*). We consider two negative plans shown in Fig. 1. For the first plan, a_3 is supposed to be the first inapplicable action while for the second plan, a_4 should be inapplicable. The initial

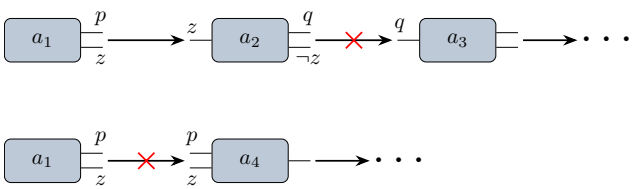

Figure 1: An example containing two negative plans.

state for both these two plans is empty. The computation for conflicts on the first iteration is as follows: For the first plan, the proposition q falls into Case 1. Hence, the repair set φ_q associated with it is $\{\mathbf{r}[\![a_2, q, \mathtt{eff}^+, -]\!]\}$. z falls into Case 2, which means that the repair set φ_z is $\{\mathbf{r}[\![a_3, z, \mathtt{prec}^+, +\}$. p falls into Case 3, and hence, it corresponds to the repair set $\{\mathbf{r}[\![a_3, p, \mathtt{prec}^+, +\}$. Taken together, the conflict on the first iteration for the first plan is the union of all these three sets. For the second plan, both p and z correspond to Case 1, and q corresponds to Case 2. Hence, the conflict for this second plan consists of $\mathbf{r}[\![a_4, q, \mathtt{prec}^+, +]\!]$, $\mathbf{r}[\![a_1, p, \mathtt{eff}^+, -]\!]$, and $\mathbf{r}[\![a_1, z, \mathtt{eff}^+, -]\!]$. After the first iteration, the collection of conflicts will contain those two conflicts.

On the start of the second iteration, a minimal hitting set δ is computed. To make the example more illustrative, let δ consists of $\mathbf{r}[\![a_3, p, \mathtt{prec}^+, +]\!]$ and $\mathbf{r}[\![a_1, z, \mathtt{eff}^+, -]\!]$. The consequence of applying δ to the domain is that the second plan now becomes a non-solution as desired, whereas a_2 becomes the first inapplicable action in the first plan. Therefore, the approach by Lin, Grastien, and Bercher (2023) is used to compute a *complex* conflict (ϕ, φ) for the first plan where ϕ consists of $\mathbf{r}[\![a_1, z, \mathtt{eff}^+, -]\!]$, and φ consists of the repair $\mathbf{r}[\![a_2, z, \mathtt{prec}^+, -]\!]$.

Lastly, on the third iteration, any minimal hitting set could make both input plans become non-solutions. For instance, $\{\mathbf{r}[\![a_3, p, \mathtt{prec}^+, +]\!], \mathbf{r}[\![a_1, p, \mathtt{eff}^+, -]\!]\}$ is one such set.

Evaluation

Lastly, we would like to present the performance of our extended domain repair approach in practice. In particular, we compared our approach with the one by Lin, Grastien, and Bercher (2023), which solves the same domain repair problem as our approach but can only deal with positive plans. The purpose of the comparison is to demonstrate the impact of having this additional feature of dealing with negative plans. We evaluated our approach in two aspects: 1) the *quality* of the repairs found by our approach (to evaluate the usefulness of our repairs and hence approach), and 2) the *runtime* of our approach for repairing a domain (to evaluate the feasibility of our approach). The quality of a set of repairs (to a domain) is evaluated in terms of *precision*, which is the percentage of the repairs which successfully fix some errors in the domain, and *recall*, which is the percentage of the errors that are fixed.

There are two approaches which, at first glance, are solving a similar problem as our approach but are in fact *different* from ours. The first one is developed by Gragera et al. (2023), which fixes a flawed domain with missing *positive* effects. Their approach relies on the assumption that the errors in the domain cause a solvable planning problem becomes unsolvable. Hence, the approach takes as input an unsolvable planning problem and fixes the domain by adding missing positive effects to turn the problem into a solvable one. The other work is by Aineto, Celorrio, and Onaindia (2019), which *learns* a domain from an input plan. However, they assume that the state trace induced by the input plan, which is *partially observed*, is flawless. The problems solved by these two approaches align more with the scenario of having *only* positive plans and *contradict* to the

	Precision		Recall		Number of plans		Average Length	
	Both	No negative	Both	No negative	Positive plans	Negative plans	Positive plans	Negative plans
TETRIS	**0.80**	0.40	**0.40**	0.10	5	10	24.60	20.80
FREECELL	**0.73**	0.70	**0.55**	0.35	62	120	43.15	38.00
TIDYBOT	**1.00**	**1.00**	**0.17**	0.08	4	10	29.25	19.20
WOODWORKING11	**0.53**	0.30	**0.27**	0.10	7	2	62.29	9.00
WOODWORKING08	**0.17**	0.07	**0.20**	0.07	30	10	20.77	6.00
LOGISTICS00	0.50	**0.80**	**0.25**	0.20	28	2	47.54	1.00
LOGISTICS98	0.60	**0.70**	**0.45**	0.35	27	12	67.96	1.33
GED	**0.90**	0.40	**0.36**	0.12	20	18	14.30	8.67
SCANALYZER	0.70	**1.00**	**0.70**	0.50	14	2	42.57	4.00
SOKOBAN	**1.00**	**1.00**	**1.00**	0.50	2	6	40.57	17.67
SLITHERLINK	0.50	**1.00**	**0.50**	**0.50**	1	4	19.00	14.00
HIKING	0.40	**0.60**	**0.20**	0.15	7	4	20.43	13.50
FLOORTILE	0.70	**0.80**	**0.35**	0.20	1	10	27.00	1.00
THOUGHTFUL	0.53	**0.70**	**0.16**	0.14	15	2	135.27	2.00
MPRIME	**1.00**	**1.00**	**1.00**	0.50	30	58	11.10	4.66

Table 1: The precision and recall of returned repairs for each domain with randomized errors.

scenario of having negative plans, which is our main contribution. More concretely, the scenario of a planning problem being unsolvable (due to errors in the domain) coincides with the scenario of some positive plans not being solutions. A correct, partially observed state trace required by the approach by Aineto, Celorrio, and Onaindia (2019) can only come from an executable plan under the correct domain and henceforth is only related to a positive plan. For this reason, we did not compare our approach against them.

Experiment Configuration We have published our code[1], all benchmarks[2], and all experimental results (Lin et al. 2025). We ran the experiments on an Intel Xeon processor with 8GB memory. Our experiments consist of two parts. In the first part, we generated flawed domains by *randomly* introducing errors to domains from the Fast Downward (FD) benchmark suite[3]. More specifically, for each domain $\mathcal{D}$ in the suite, we first uniformly selected 20% of action schemas and randomly introduced one of the following three errors to each selected action schema: 1) deleting a precondition, 2) deleting a negative effect, and 3) adding a positive effect. Intuitively speaking, these errors *overly relax* the ground truth domain. After that, we again randomly selected 20% of action schemas and introduced one of the following errros to each of them which *overly restricts* the domain: 1) adding a precondition, 2) adding a negative effect, and 3) deleting a positive effect. Note that some action schemas might be selected twice and hence have more than one error. We denote the flawed domain obtained from D as D'. For each planning task $\mathcal{T}$ with respect to $\mathcal{D}$ in the FD problem suite, we computed a set of positive plans and of negative plans for $(\mathcal{D}', \mathcal{T})$ by invoking the FD planner (Helmert 2006) with a 15 minute timeout. More concretely, we compiled the task of computing positive plans and negative plans as another planning problem whose solutions represent plans that are solutions in one domain but not the other (recall that a positive plan is a solution in the ground truth but not in the flawed domain while a negative plan is the other way round). We created a benchmark set containing 15 flawed domains in total. We excluded domains which do *not* have both positive and negative plans and which have features that are unsupported by our approach. The number of positive plans and of negative plans for each domain is shown in Tab. 1. It also shows the average length of positive plans and of negative ones for each domain. In particular, since the suffix of a negative plan after its first inapplicable action does not matter, the length of a negative plan in the table actually refers to the length of the prefix before the inapplicable action.

In the second part of the evaluation, we handcrafted three flawed domains. Two out of three were from BlocksWorld, each with different errors. The last one is generated from Gripper. For the first BlocksWorld domain, we removed the predicate `clear` from the preconditions of `stack`, `unstack`, and `pickup`. For the second BlocksWorld domain, the predicate `handempty` was removed from the preconditions of `unstack` and the positive effects of `stack`. For the last benchmark, we modified the Gripper domain by deleting the predicate `free` from the preconditions of `pick` and the positive effects of `drop`.

The bespoke benchmarks are inspired by the specific challenges in robotics. Planning domains can be used for modeling problems in 3D environments (Roberts et al. 2021). Geometric robot motion infeasibility (Kavraki and LaValle 2008) needs to be considered within domain models in a category task and motion planning problems (Dantam et al. 2018; Garrett et al. 2021). Flawed planning domains like BlocksWorld are used to model robotic arms manipulating objects (Dantam et al. 2018; Pan et al. 2021), where *negative* plans in BlocksWorld can be robot motion infeasible. In the first modified BlocksWorld benchmark, `clear` represents the geometric information about whether the robot hand (end-effector) can approach the object unimpeded. `handempty` (in BlocksWorld) and `free` (in Gripper) rep-

[1]https://github.com/Songtuan-Lin/classical-domain-repairer

[2]https://github.com/Songtuan-Lin/repairer-benchmarks

[3]https://github.com/aibasel/downward-benchmarks

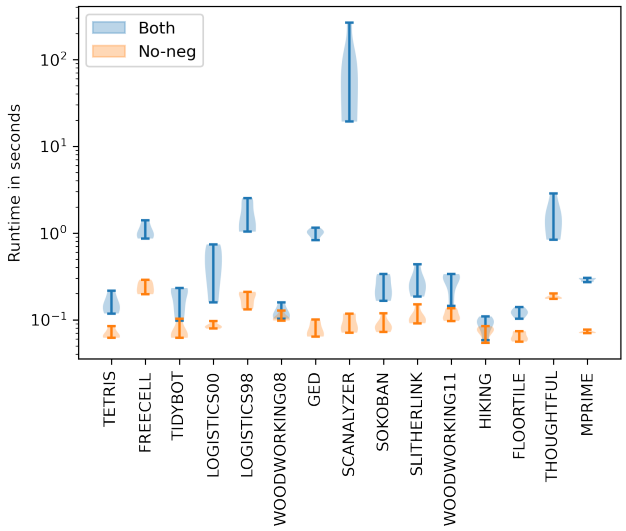

Figure 2: Runtime for solving each domain repair problem instance with randomized errors.

resent whether there is some geometry (grasped) between the robot end-effector that restricts its operation.

Results for Domains with Random Errors Since there might be multiple cardinality-minimum repair sets which can solve a domain repair problem (among which only one can modify the flawed domain to the ground truth), both our approach and the one by Lin, Grastien, and Bercher (2023) might return any of them in a single run due to the randomness involved in the hitting set solver used. Thus, to better estimate the quality of the repairs found, for each flawed domain in the benchmark set, we ran the two approaches 5 times. After that, we computed the precision and recall of the repairs found for each domain by taking the average of the results for these 5 runs. The total precision and recall are the average over all domains. We provided both positive and negative plans to our approach and only positive ones to the approach by Lin, Grastien, and Bercher (2023). The results for each benchmark domain are shown in Tab. 1 where the columns labeled with "Both" show the result for our approach while those labeled with "No negative" are the results for the approach by Lin, Grastien, and Bercher (2023). As we can see from the table, our approach achieved higher precision in more than half benchmarks, and in almost every domain, our approach had better recall. We believe that the reason that our approach has lower precision in some domains is due to randomness in the hitting set solver we used.

Fig. 2 depicts the runtime of our approach and of the one by Lin, Grastien, and Bercher (2023) for solving each domain repair problem instance. Each box corresponds to one benchmark, encapsulating the runtime for all 5 runs. The upper and lower edges of a box indicate the maximal and minimal runtime. As can be seen from the plot, our approach requires more time than the other. This is *not* surprised because our approach is dealing with a more complicated reasoning task where the search space is much larger because

	Precision		**Recall**	
	Both	No-neg	Both	No-neg
BLOCKS (`clear`)	**1.00**	0.00	**1.00**	0.00
BLOCKS (`handempty`)	**1.00**	0.60	**1.00**	0.30
GRIPPER (`free`)	**1.00**	0.00	**1.00**	0.00

Table 2: The precision and recall of the founded repairs for each domain with bespoke errors.

of the possible preconditions that can be added. However, almost all domains were repaired within 100 seconds, among which many need less than 10 seconds. The results thus show that the approach can correct a domain in reasonable time showing the feasibility of our approach. In particular in the area of modeling support such very small runtimes that we could achieve are important as – based on common knowledge and personal experience by the authors – users likely want to obtain feedback and support quickly and are too impatient or busy to wait for several minutes until they can continue with their domain modeling task.

Results for Domains with Bespoke Errors We again ran our approach 5 times and compared it with the one by Lin, Grastien, and Bercher (2023). Tab. 2 shows the results. Our approach achieved 100% precision and 100% recall. In particular, the other approach failed to find any repair for two benchmarks. This is because the errors introduced to those two domains do *not* overly restrict the ground truth, meaning that every solution in the ground truth is still a solution in the flawed domain. Hence, the approach by Lin, Grastien, and Bercher (2023) cannot find any repairs. This further emphasizes the importance of having negative plans, because these benchmarks are closely related to practical applications.

Conclusion

In this paper, we proposed an approach for fixing a flawed domain which takes as input a domain, a set of positive and of negative plans, and outputs a *cardinality-minimum* set of repairs to the domain so as to turn all positive plans into solutions and all negative ones into non-solutions. Our experimental results show strong empirical performance of our approach in fixing domains both drawn from the IPC with randomized errors and from handcrafted (flawed) domains motivated by robotics applications. This thus demonstrates the potential of deploying our approach in modeling assistance as well as forging connections to applications like robotics.

Acknowledgements

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.

References

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning Action Models with Minimal Observability. *Artificial Intelligence*, 275: 104–137.

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

Bylander, T. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, 69: 165–204.

Dantam, N. T.; Kingston, Z. K.; Chaudhuri, S.; and Kavraki, L. E. 2018. An Incremental Constraint-Based Framework for Task and Motion Planning. *The International Journal of Robotics Research*, 37: 1134–1151.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity Results for HTN Planning. *Annals of Mathematics and Artificial Intelligence*, 18: 69–93.

Erol, K.; Nau, D. S.; and Subrahmanian, V. 1995. Complexity, Decidability and Undecidability Results for Domain-Independent Planning. *Artificial Intelligence*, 76: 75–88.

Fox, M.; and Long, D. 2003. PDDL2.1 : An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20: 61–124.

Garrett, C. R.; Chitnis, R.; Holladay, R.; Kim, B.; Silver, T.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Integrated Task and Motion Planning. *Annual Review of Control, Robotics, and Autonomous Systems*, 4: 265–293.

Gragera, A.; Fuentetaja, R.; Olaya, Á. G.; 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.

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

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

Kavraki, L. E.; and LaValle, S. M. 2008. *Motion Planning*, 109–131. Berlin, Heidelberg: Springer Berlin Heidelberg.

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.; 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.; Grastien, A.; Shome, R.; and Bercher, P. 2025. Experimental Results for the AAAI 2025 Paper "Told You That Will Not Work: Optimal Corrections to Planning Domains Using Counter-Example Plans". doi: 10.5281/ZENODO.14533200.

Lin, S.; Höller, D.; and Bercher, P. 2024. Modeling Assistance for Hierarchical Planning: An Approach for Correcting Hierarchical Domains with Missing Actions. In *Proceedings of the 17th International Symposium on Combinatorial Search, SoCS 2024*, 55–63. AAAI.

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.

Muise, C. 2016. Planning.Domains. In *System Demonstrations at the 26th International Conference on Automated Planning and Scheduling, ICAPS 2016*.

Pan, T.; Wells, A. M.; Shome, R.; and Kavraki, L. E. 2021. A general task and motion planning framework for multiple manipulators. In *Proceedings of the 34th IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021*, 3168–3174. IEEE.

Reiter, R. 1987. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32: 57–95.

Roberts, J. O.; Mastorakis, G.; Lazaruk, B.; Franco, S.; Stokes, A. A.; and Bernardini, S. 2021. vPlanSim: An Open Source Graphical Interface for the Visualisation and Simulation of AI Systems. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling, ICAPS 2021*, 486–490. AAAI.

Slaney, J. 2014. Set-theoretic Duality: A Fundamental Feature of Combinatorial Optimisation. In *Proceedings of the 21st European Conference on Artificial Intelligence, ECAI 2014*, 843–848. IOS.

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.

Struss, P.; and Dressler, O. 1989. "Physical Negation": Integrating Fault Models into the General Diagnostic Engine. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence. IJCAI 1989*, 1318–1323. Morgan Kaufmann.

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: 215–230.