Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS 2023)

A Planning Approach to Repair Domains with Incomplete Action Effects

Alba Gragera, Raquel Fuentetaja, Ángel García-Olaya, Fernando Fernández

Computer Science and Engineering Department, Universidad Carlos III de Madrid, Spain
agragera@pa.uc3m.es, {rfuentet, agolaya, ffernand}@inf.uc3m.es

Abstract

Automated planning is a problem solving technique for a wide range of scenarios and goals, which typically involves the creation of domain and problem files in formal languages. However, producing complete model descriptions can be challenging and time-consuming, especially for non-experts. Although many tools have been developed to support file editing, mistakes can still be made, such as incomplete or improper specification of the initial state or the set of actions. These errors often result in unsolvable tasks for planners, making it impossible to generate a plan. Explaining the absence of a solution in such cases is essential to support humans in the development of automated planning tasks. In this paper, we introduce a novel approach to repair planning models where the effects of some actions are incomplete, without further information from the user side. We propose a compilation of the unsolvable task to a new extended planning task, in which actions are permitted to insert possible missing effects. The solution is a plan that achieves the goals of the original problem while also alerting users of the modifications made to do so. Experimental results demonstrate that this approach can effectively repair incomplete planning domains.

Introduction

Automated planning tasks are usually defined by a domain description, which specifies all available actions and the predicates used to describe the states; and a problem description that contains the initial state and the goal(s) to achieve. Assuming a solvable and well-defined planning task, and infinite memory and time resources, a planner will return a solution. However, modeling a planning task is not always trivial, and there may be scenarios where neither completeness nor correctness of the planning task specification can be ensured (Kambhampati 2007; McCluskey, Vaquero, and Vallati 2017). Flaws in the task model can arise due to a noisy knowledge acquisition process, or because domain engineers lack expertise in the description language or do not have a deep knowledge of the current task, especially when the domain is difficult to represent. These issues can result in an incomplete specification of the initial state or actions, rendering the planning task unsolvable.

Providing users a comprehensive explanation about the absence of solution and how to solve it is an important challenge for the planning community (Fox, Long, and Magazzeni 2017; Chakraborti, Sreedharan, and Kambhampati 2020). Previous works have considered scenarios where initial states prevent the achievement of goals (Göbelbecker et al. 2010; Sreedharan et al. 2019), providing explanations and alternative initial states that render the task solvable. However, these works assume the proper specification of the domain and do not consider modifications to it. But just making changes to the initial state is sometimes not enough. For instance, forgetting to include the action effect that cleans the glass (Figure 1) in the *barman* domain (Linares López, Jiménez Celorrio, and García-Olaya 2015) can make the task unsolvable, but setting the glass as clean in the initial state will not solve the problem if it gets dirty again.

```
(:action clean-shot
   :pareters (?s - shot ?b - beverage ?h1 ?h2 - hand)
   :precondition (and (holding ?h1 ?s) (handempty ?h2)
                      (empty ?s) (used ?s ?b))
   :effect (and (not (used ?s ?b)) (clean ?s)))
```

Figure 1: Clean shot action. If *(clean ?s)* is omitted, the planner will not find a plan in tasks requiring to reuse the glass.

Due to the number of potential changes to the set of actions, repairing faulty domains is not trivial (Lin and Bercher 2021). Previous works assume guidelines from users, often in the form of a suggested valid plan (McCluskey, Richardson, and Simpson 2002; Simpson, Kitchin, and McCluskey 2007; Nguyen, Sreedharan, and Kambhampati 2017; Lin, Grastien, and Bercher 2023). In contrast, in this work we propose the use of automated planning to repair planning tasks themselves, without additional information from the user. We focus on errors in the domain model that render the planning task unsolvable. Specifically, we consider missing action effects, which can compromise the task's solvability. We present a compilation of the unsolvable task into a new extended planning task that includes operators to add or delete new facts to the current state, linking them as new effects of the original actions. The resulting plan not only achieves the original goals but also provides information on how the model was repaired to make the task solvable.

Background

Automated planning tasks define problems whose solutions are sequences of actions, called plans, that achieve the problem goals when applied to specific initial states. We use the first-order (lifted) planning formalism, where a classical planning task is a pair $\Pi = \langle D, I \rangle$, where D is the *planning domain* and I defines a *problem instance*. A planning domain is a tuple $D = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{A} \rangle$; where $\mathcal{H}$ is a *type hierarchy*; $\mathcal{C}$ is a set of (domain) *constants*; $\mathcal{P}$ is a set of *predicates* defined by the predicate name and the types of its arguments; and $\mathcal{A}$ is a set of *action schemas*. If $p(t) \in \mathcal{P}$ is an n-ary predicate, and $t = t_1, \ldots, t_n$ are either typed constants or typed free variables, then $p(t)$ is an *atom*. An atom is *grounded* if its arguments do not contain free variables. Action schemas $a \in \mathcal{A}$ are tuples $a = \langle name(a), par(a), pre(a), add(a), del(a), cost(a) \rangle$, defining the action *name*; the action *parameters* (a finite set of free variables); the *precondition*, *add* and *delete* lists; and the action *cost*. $pre(a)$ is a set of literals representing what must be true or false in a state to apply the action. $add(a)$ and $del(a)$ represent the changes produced in a state by the application of the action (added and deleted atoms, respectively). A problem instance is a tuple $I = \langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{O}$ is a set of typed constants representing problem-specific *objects*; $\mathcal{I}$ is the set of ground atoms in the *initial state*; and finally, $\mathcal{G}$ is the set of ground atoms defining the *goals*.

Grounded actions $\underline{a}$ are obtained from action schemas a by substituting the free variables in the parameters by constants in $\mathcal{O}$. A grounded action $\underline{a}$ is *applicable* in an state s if $pre(\underline{a}) \subseteq s$. When a grounded action is applied to s we obtain a successor state s', defined as $s' = (s \setminus del(\underline{a})) \cup add(\underline{a})$. A *plan* π is a sequence of grounded actions $\underline{a}_1, \ldots, \underline{a}_n$ such that each $\underline{a}_i$ is applicable to the state s_{i-1} generated by applying $\underline{a}_1, \ldots, \underline{a}_{i-1}$ to $\mathcal{I}$; $\underline{a}_1$ is applicable in $\mathcal{I}$; and the consecutive application of all actions in the plan generates a state s_n containing the goals, $\mathcal{G} \subseteq s_n$. The cost of a plan is defined as $cost(\pi) = \sum_{\underline{a}_i \in \pi} cost(\underline{a}_i)$.

Problem Formulation

First, we define domains with incomplete action effects.

Definition 1 (Effect-incomplete domain). *A planning domain $D^- = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{A}^- \rangle$ is effect-incomplete wrt. an underlying planning domain $D = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{A} \rangle$ iff:*

- *There is a one-to-one correspondence between the action schemas, such that for every pair of corresponding action schemas, $a \in \mathcal{A}$ and $a^- \in \mathcal{A}^-$,*

$$a^- = \langle name(a), par(a), pre(a), add^-(a), del^-(a) \rangle$$

  where $add^-(a) \subseteq add(a)$, $del^-(a) \subseteq del(a)$; and
- *There is at least one action schema in $\mathcal{A}^-$ for which $add^-(a) \subset add(a)$ or $del^-(a) \subset del(a)$.*

For an effect-incomplete domain there always exists at least one $a^- \in \mathcal{A}^-$ whose positive or negative effects are a proper subset of the underlying action $a \in \mathcal{A}$. The solution consists of repairing the actions of the effect-incomplete domain with additional *add* or *del* effects so that the resulting planning task is solvable.

Definition 2 (Repairing set). *Given an effect-incomplete domain $D^- = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{A}^- \rangle$, a repairing set $\hat{R} = \{(\hat{R}_i^+, \hat{R}_i^-)\}_{i=1,\ldots,|\mathcal{A}^-|}$, is a set of pairs of collections of atoms $\hat{R}_i^X = \{p(t) \mid p \subseteq \mathcal{P}, t \subseteq par(a_i^-)\}$, $X \in \{+, -\}$, to extend the positive and negative effects of every $a_i^- \in \mathcal{A}^-$.*

The repaired domain is generated by extending the positive and negative effects of every action with its corresponding repairing set.

Definition 3 (Repaired domain). *Given an effect-incomplete domain $D^- = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{A}^- \rangle$ and a repairing set $\hat{R}$, the repaired domain is $D^{\hat{R}} = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{A}^{\hat{R}} \rangle$ with $add(a_i^{\hat{R}}) = add(a_i^-) \cup \hat{R}_i^+$ and $del(a_i^{\hat{R}}) = del(a_i^-) \cup \hat{R}_i^-$.*[1]

We will denote the repaired domain as $D^{\hat{R}} = D^- \oplus \hat{R}$. We assume that the terms in t for every $p(t)$ in $\hat{R}$ also appear in the parameters of the action being repaired. Based on this, we define an uninformed repairing problem as follows:

Definition 4 (Uninformed repairing problem). *Given an unsolvable planning task $\Pi^- = \langle D^-, I \rangle$ with an effect-incomplete domain wrt. the underlying unknown domain of an assumed solvable planning task $\Pi = \langle D, I \rangle$, the uninformed repairing problem consists of determining a repairing set $\hat{R}$ such that the planning task $\Pi^{\hat{R}} = \langle D^{\hat{R}}, I \rangle$, with $D^{\hat{R}} = D^- \oplus \hat{R}$, is solvable.*

The lack of information about the underlying domain D leads to estimated reparations and many possible repairing sets. Although our approach focuses on repairing missing effects, it is worth noting that an excess of preconditions (positive or negative) can also be reduced to a lack of effects (positive or negative) to satisfy them. In the remainder of the paper we present a compilation of the uninformed repairing problem into classical planning, including its theoretical properties, a way of biasing the search towards more desirable repair plans, and experimental results in some domains. We finish with the presentation of the related work, the conclusions and future work.

Compilation to Classical Planning

To solve the uninformed repairing problem, we compile Π^- into a new planning task Π' that includes additional operators to repair any action in the domain. At the operational level, each grounded action is now divided into three stages: (1) the application of the action, (2) any necessary reparation, and (3) the closure of the action. For example, let us consider the running example. When the planner first instantiates the *clean-shot* action, it is incomplete and it cannot continue without actually cleaning the glass. To fix this, a repair operator is applied to the action, linking it with the *clean* new positive effect. If no further repairs are necessary, the current action is closed and it moves on to the following one. This process is repeated until all goals are achieved. Note that actions can also be closed without any reparation.

To manage this process, we reformulate the planning task elements. We extend the original type hierarchy with new

[1] Note that the repairing set can be empty for some actions.

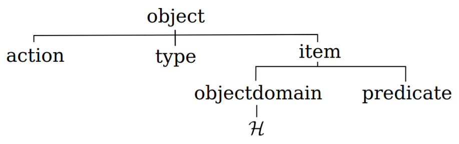

Figure 2: Hierarchy type of the new extended planning task.

types, resulting in a new type hierarchy $\mathcal{H}'$, shown in Figure 2. Types *action* and *type* (denoted in the remainder of the paper as o_act and o_type, respectively) are used to provide information about the different actions and type names present in the original domain D^-. The type *item* (denoted as o_item) represents original domain objects and predicate names, both of which are elements required to repair the planning task. We use *objectdomain* (denoted as o_obj) as an abstract object to summarize the type hierarchy in the original domain. We introduce then the following new domain sets of constants: $\mathcal{C}_a$ of type *o_act* to represent action names, $\mathcal{C}_a = \{name(a) \mid a \in \mathcal{A}\}$; $\mathcal{C}_t$ of type *o_type* to represent the types in the original hierarchy, $\mathcal{C}_t = \{t_{type} \mid type \in \mathcal{H}\}$; and $\mathcal{C}_p$ of type *o_pred* to represent the names of the predicates in the original domain, $\mathcal{C}_p = \{p \mid p(t) \in \mathcal{P}\}$. Figure 3 shows examples of the type redefinition and the domain constants for the *barman* domain.

```
(:types
  action type item - object
  predicate objectdomain - item
  hand level beverage dispenser container - objectdomain
  ingredient cocktail  - beverage
  shot shaker  - container)
(:constants
    grasp leave fill-shot clean-shot ... - action
    handempty empty clean used shaked ... - predicate
    t_shot t_ingredient t_shaker ... - type
...)
```

Figure 3: Reformulated types and constants in Π'.

We also introduce two groups of new predicates that replace the original ones. The first group ($\mathcal{P}_{access}$) contains predicates to access the elements of the original task and the second group ($\mathcal{P}_{control}$) contains control predicates for the repairing task. $\mathcal{P}_{access}$ includes:

- $\mathbf{functor(o\_pred)}$, that allows to define facts to represent predicate names. Possible examples are *functor(handempty)*, *functor(clean)*, etc.
- $\mathbf{type(o\_obj, o\_type)}$, that allows to define facts to access the type and super-types of every object in the original domain. Examples are *(type shaker01 t_shaker)* and *(type shaker01 t_container)*.
- $\mathbf{pred\_\langle n\rangle(o\_pred, o\_type_1, \ldots, o\_type_n)}$, representing that there is a n-ary predicate defined in the original domain, also containing the predicate symbol and the types involved. *(pred_2 contains t_container t_beverage)* means that there is a predicate of arity 2 to represent that a container contains a beverage.
- $\mathbf{in\_state\_\langle n\rangle(o\_pred, o\_obj_1, \ldots, o\_obj_n)}$ represents that the fact $(o\_pred, o\_obj_1, \ldots, o\_obj_n)$ is true in a state. In this way we can group propositions with the same arity n. For instance the grounded atoms *(clean shot01)* and *(empty shot01)*, with arity 1, can be represented by the same predicate in_state_1. Then, if the shot is clean or empty in the current step the new task represents *(in_state_1 clean shot01)* and *(in_state_1 empty shot01)*. This lets us generalize the task, allowing the repair operators to add or remove any fact from the current state grouped by arity.
- $\mathbf{goal\_\langle n\rangle(o\_pred, o\_obj_1, \ldots, o\_obj_n)}$ represents that $(o\_pred, o\_obj_1, \ldots, o\_obj_n)$ is a goal of the task.
- $\mathbf{add\_eff(o\_pred, o\_act)}$, denotes that a predicate name appears in the add effects of an action. For instance, *(add_eff holding grasp)*.
- $\mathbf{del\_eff(o\_pred, o\_act)}$, similarly, denotes that a predicate name appears in the del effects of an action.

The new *control predicates*, $\mathcal{P}_{control}$, are:

- $\mathbf{checked(o\_act)}$, denoting that an action has been already added to the plan, repaired or not. No previously checked actions can be repaired.
- $\mathbf{current\_action(o\_act)}$, used to control which is the current action in course, to repair it if needed.
- $\mathbf{patched(o\_act)}$, indicating that action has been fixed at least once.
- $\mathbf{fix(o\_act, o\_pred)}$, meaning that an action has been repaired and the predicate involved.
- $\mathbf{used(o\_item)}$, for repairing the action with the objects currently in use.
- **open**, indicating that the reparation is allowed.
- **fixed**, denoting that an action has been fixed with at least one of the add effects involved in a fix.
- $\mathbf{add\_added(o\_pred, o\_act)}$, representing that a functor has been already used to repair an action as a new positive effect.
- $\mathbf{del\_added(o\_pred, o\_act)}$, to denote that a predicate has been used to repair an action as a new negative effect.

The new extended planning task is $\Pi' = \langle D', I' \rangle$, where $D' = \langle \mathcal{H}', \mathcal{C}', \mathcal{P}', \mathcal{A}' \rangle$. $\mathcal{H}$' is the described new hierarchy (see Figure 2); the constants include the original constants and the new ones: $\mathcal{C}' = \mathcal{C} \cup \mathcal{C}_a \cup \mathcal{C}_t \cup \mathcal{C}_p$; the predicates definition includes the new access and control predicates: $\mathcal{P}' = \mathcal{P}_{access} \cup \mathcal{P}_{control}$; and $\mathcal{A}'$ is new set of actions schemes, defined as $\mathcal{A}' = \mathcal{A}_\alpha \cup \mathcal{A}_\phi \cup \mathcal{A}_{\psi_+} \cup \mathcal{A}_{\psi_-} \cup \mathcal{A}_\omega$, where $\mathcal{A}_\alpha$ consists of actions generated from the original domain actions, but compiled to match the new object representation. The remaining sets of actions are used for repair purposes, as explained below.[2]

[2]The extended planning task requires of negative preconditions and forall effects. We consider classical planning extended with that additional expressivity incorporated in ADL (Pednault 1989).

ACTIONS FROM ORIGINAL ACTIONS ($\mathcal{A}_\alpha$) There is an action $\alpha_a \in \mathcal{A}_\alpha$ for every action $a \in \mathcal{A}$, defined as:

$$\begin{aligned}
name(\alpha_a) =& name(a)\\
par(\alpha_a) =& par(a)\\
pre(\alpha_a) =& \{in\_state\_\langle n\rangle(p,t) \mid p(t) \in pre(a)\} \cup\\
& \{\neg open\}\\
add(\alpha_a) =& \{in\_state\_\langle n\rangle(p,t) \mid p(t) \in add(a)\} \cup\\
& \{current\_action(a), open\} \cup\\
& \{used(x) \mid x \in par(a)\}\\
del(\alpha_a) =& \{in\_state\_\langle n\rangle(p,t) \mid p(t) \in del(a)\}\\
cost(\alpha_a) =& 0
\end{aligned}$$

where $p(t)$, $t \subseteq par(a) \cup \mathcal{C}$, denotes any atom in the action preconditions or effects. To adapt to the new object representation, predicates are replaced by their $in\_state\_\langle n\rangle$ reformulation, where n is the arity of the original predicate. Additionally, effects now include information about the current action being executed and the objects involved in that action. Finally, we introduce an *open* flag fact to indicate that an action can be repaired after it has been applied. While this fact is present in the state, no other action in $\mathcal{A}_\alpha$ can be applied. Figure 4 provides an example of this compilation.

```
(:action clean-shot
   :parameters (?s - shot ?b - beverage ?h1 ?h2 - hand)
   :precondition (and (in_state2 holding ?h1 ?s)
                      (in_state1 handempty ?h2)
                      (in_state1 empty ?s)
                      (in_state2 used ?s ?b)
                      (not (open))
   :effect (and (not (in_state2 used ?s ?b))
                (current_action clean-shot)
                (used ?s) (used ?b)
                (used ?h1) (used ?h2) (open)))
```

Figure 4: Clean shot action compiled for the reparation task. The positive effect *clean ?s* is omitted.

FIX ACTION ($\mathcal{A}_\phi$) This repair operator selects a predicate symbol and links it as a new effect of any action. The operator takes two parameters: a, which is a variable of type *o_act* representing the action to be repaired, and p, which is a variable of type *o_pred* representing the predicate to be added as a new effect. It follows the next action scheme:

$$\begin{aligned}
par(\phi) =& (a, p)\\
pre(\phi) =& \{current\_action(a), functor(p),\\
& \neg checked(a), open\}\\
add(\phi) =& \{fix(a,p), patched(a)\}\\
del(\phi) =& \emptyset\\
cost(\phi) =& C_\phi
\end{aligned}$$

This operator can only be applied to an action that is currently open and has not been checked (i.e., added to the plan) previously. Once an action has been used in the plan, it cannot be linked to new effects. The repair operator updates the state to indicate that the action has been patched with a predicate symbol $p \in \mathcal{C}_t$. This operator has an associated cost, which will be used as a bias to minimize the number of reparations made in the domain.

After a predicate symbol is linked to an action as a new effect, we include the following two actions to establish whether the effect will be positive or negative.

ADD-FIX ACTIONS ($\mathcal{A}_{\psi_+}$) These operators perform the reparation as a positive effect by matching the predicate symbol with its parameters and adding it to the state with the appropriate objects. The number of objects required depends on the arity n of the predicate being added. For example, a predicate *clean* may require a single object of type *shot*, whereas other predicates may require a larger number of objects. Accordingly, for each arity n we define a repair action ψ_n that takes the following parameters: variables a and p of type *o_act* and *o_pred*, respectively; n variables, $x_1, \ldots, x_n$, of type *o_obj*, representing domain objects; and n variables, $y_1, \ldots, y_n$ of type *o_type*, representing their types.

$$\begin{aligned}
par(\psi_{+n}) =& (a, p, x_1, \ldots, x_n, y_1, \ldots, y_n)\\
pre(\psi_{+n}) =& \{current\_action(a), fix(a,p), functor(p),\\
& pred\_\langle n\rangle(p, y_1, \ldots, y_n)\} \cup\\
& \{type(x_i, y_i) \mid 1 \le i \le n\} \cup\\
& \{used(x_i) \mid 1 \le i \le n\}\\
& \{\neg del\_added(p,a)\}\\
& \{\neg in\_state\_\langle n\rangle(p, x_1, \ldots, x_n)\}\\
add(\psi_{+n}) =& \{in\_state\_\langle n\rangle(p, x_1, \ldots, x_n)\} \cup \{fixed\} \cup\\
& \{add\_added(p,a)\}\\
del(\psi_{+n}) =& \emptyset\\
cost(\psi_{+n}) =& 0
\end{aligned}$$

DEL-FIX ACTIONS ($\mathcal{A}_{\psi_-}$) These operators follow a similar action scheme as the ADD-FIX operators, but they remove atoms from the current state. This simulates a negative effect of the action to which it was linked. We add a precondition to ensure that a predicate previously fixed as a positive effect cannot become a negative effect for the same action. DEL-FIX actions have an associated cost. This penalty is necessary because we generally prefer to add facts to the state rather than delete them. For example, if an action has *(not (blocked ?path))* as precondition, it is more desirable to search for another path instead of deleting the obstacle.

$$\begin{aligned}
par(\psi_{-n}) =& (a, p, x_1, \ldots, x_n, y_1, \ldots, y_n)\\
pre(\psi_{-n}) =& \{current\_action(a), fix(a,p), functor(p),\\
& pred\_\langle n\rangle(p, y_1, \ldots, y_n)\} \cup\\
& \{type(x_i, y_i) \mid 1 \le i \le n\} \cup\\
& \{used(x_i) \mid 1 \le i \le n\}\\
& \{in\_state\_\langle n\rangle(p, x_1, \ldots, x_n)\}\\
& \{\neg add\_added(p,a)\}\\
add(\psi_{-n}) =& \{fixed\} \cup \{del\_added(p,a)\}\\
del(\psi_{-n}) =& \{in\_state\_\langle n\rangle(p, x_1, \ldots, x_n)\}\\
cost(\psi_{-n}) =& C_{\psi_-}
\end{aligned}$$

CLOSE ACTION ($\mathcal{A}_\omega$) The application of a close action concludes the reparation of an action. It deletes the information about the current action and the objects used, and updates the action as already checked. The close action scheme has a single parameter a of type o_act and a *forall* effect to remove the *used* predicates, where x is of type o_item:

$$\begin{aligned}
par(\omega) =& (a)\\
pre(\omega) =& \{current\_action(a)\} \cup \{open\}\\
add(\omega) =& \{checked(a)\}\\
del(\omega) =& \{\texttt{forall}(x, used(x))\} \cup\\
& \{current\_action(a)\} \cup \{open\}\\
cost(\omega) =& 0
\end{aligned}$$

In summary, the new action schemes $\mathcal{A}' = \mathcal{A}_\alpha \cup \mathcal{A}_\phi \cup \mathcal{A}_{\psi_+} \cup \mathcal{A}_{\psi_-} \cup \mathcal{A}_\omega$, are defined as $\mathcal{A}_\alpha = \{\alpha_a \mid a \in \mathcal{A}\}$, with an action for every action in the original domain; $\mathcal{A}_\phi = \{\phi\}$, the FIX ACTION scheme; the ADD-FIX action schemes $\mathcal{A}_{\psi_+} = \{\psi_{+_n} \mid n\, arity\, of\, p(t) \in \mathcal{P}\}$; the DEL-FIX action schemes $\mathcal{A}_{\psi_-} = \{\psi_{-_n} \mid n\, arity\, of\, p(t) \in \mathcal{P}\}$, both with an action scheme for each arity of the predicates in the original domain; and the CLOSE ACTION scheme $\mathcal{A}_\omega = \{\omega\}$.

PROBLEM INSTANCE (I') The problem instance is compiled as $I' = \langle \mathcal{O}, \mathcal{I}', \mathcal{G}' \rangle$. Predicates in $\mathcal{P}_{access}$ are instantiated to represent static domain information about the original predicates and actions, and the $in\_state\_\langle n \rangle$ reformulation of the original initial state. $\mathcal{I}'$ is then defined as follows, where the function $all(\mathcal{H}, o)$ returns all types (primitive type and all super-types) of an object in a type hierarchy:

$$\begin{aligned}
\mathcal{I}' =& \{in\_state\_\langle n \rangle(p,t) \mid p(t) \in \mathcal{I}\} \cup\\
& \{functor(p) \mid p \in \mathcal{P}\} \cup\\
& \{pred\_\langle n \rangle(p, t_{type_1}, \ldots, t_{type_n}) \mid p \in \mathcal{P}\} \cup\\
& \{type(o, t_{type}) \mid o \in \mathcal{O}, t_{type} \in all(\mathcal{H}, o)\} \cup\\
& \{goal\_\langle n \rangle(p,t) \mid p(t) \in \mathcal{G}\} \cup\\
& \{add\_eff(p, name(a)) \mid p(t) \in add(a), a \in \mathcal{A}^-\} \cup\\
& \{del\_eff(p, name(a)) \mid p(t) \in del(a), a \in \mathcal{A}^-\}
\end{aligned}$$

The set of goals is replaced by the $in\_state\_\langle n \rangle$ reformulation of the goals of the original problem. Figure 5 shows a partial example of the compiled problem.

$$\mathcal{G}' = \{in\_state\_n(p,t) \mid p(t)\} \in \mathcal{G}\}$$

```
(:init
    (functor clean)
    (pred_1 clean t_container)
    (type t_container shot01)
    (in_state1 clean shot01)
    (in_state1 empty shot01)
    (add_eff contains fill-shot)
    (del_eff used clean-shot)
    ...)
(:goal (in_state2 contains shot01 cocktail4))
```

Figure 5: Partial example of the compiled problem.

We define the total cost of a solution as the sum of the reparations made in the domain and include a metric in the problem instance to minimize this value. Our aim is to solve the uninformed repairing problem and find the minimum repairing set $\hat{R}$ so that $\Pi^{\hat{R}}$ is solvable. Figure 6 shows a part of an expected solution for the extended planning task in the barman domain, where the repair actions are highlighted.

```
(clean-shot shot01 ingredient01 left right)
(fix clean-shot clean)
(add-fix-1 clean-shot clean t_container shot01)
(close clean-shot)
(fill-shot shot01 ingred2 left right dispenser2)
(close fill-shot)
(pour-shot-used-shaker shot01 ingred2 shaker1 left l1 l2)
(close pour-shot-to-used-shaker)
(clean-shot shot01 ingred2 left right)
(add-fix-1 clean-shot clean t_container shot01)
(close clean-shot)
```

Figure 6: Excerpt of the solution plan.

The *clean-shot* action is repaired with the *clean* functor, which is after used to apply the reparation with the proper object. We can obtain the repairing set $\hat{R}$ from this plan by extracting the repaired action and the predicate used. If the compilation is performed on a fully specified task, the resulting plan will achieve the goals without requiring any repairs.

Theoretical Properties

The compiled task Π' is complete and sound.

Theorem 1 (Completeness). *Given an uninformed repairing problem with unsolvable task $\Pi^- = \langle D^-, I \rangle$ with $D^- = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{A}^- \rangle$, there is always a solution π' for the extended task Π'.*

Proof. When an action $a_i^- \in \mathcal{A}^-$ is applied, it introduces the artificial *open* proposition, allowing the application of a FIX action. This action can select any predicate symbol in $\mathcal{P}$ to fix a_i^-. Actions ADD-FIX and DEL-FIX are applicable if a_i^- has been fixed, adding or removing the predicate $p(t)$ used in the reparation from the state. The arguments of $p(t)$ are parameters of the current action, $t \subseteq par(a_i^-)$. CLOSE action updates the action as *checked* and closes the phase to repeat the process with a_{i+1}^-. Successive applications of these repair actions over $\mathcal{A}^-$ can add fixes for all predicates, potentially adding or deleting any subset of facts in the current state, including the goals. □

Theorem 2 (Soundness). *Given an uninformed repairing problem with unsolvable task $\Pi^- = \langle D^-, I \rangle$ with $D^- = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{A}^- \rangle$, the repaired set $\hat{R}$ obtained from any solution to the extended task Π' makes the planning task $\Pi^{\hat{R}} = \langle D^{\hat{R}}, I \rangle$, with $D^{\hat{R}} = D^- \oplus \hat{R}$ solvable.*

Proof. Plans for Π' are composed of the original (compiled) actions in $\mathcal{A}^-$ ($\mathcal{A}_\alpha$), interleaved with actions to repair and close them ($\mathcal{A}_\phi, \mathcal{A}_{\psi_+}, \mathcal{A}_{\psi_-}, \mathcal{A}_\omega$) if needed. We keep the original set of goals, so any plan for Π' is a plan for $\Pi^{\hat{R}}$. □

Bias for Domain Reparation

There are many changes to a domain that can make the planning task solvable. However, some of these modifications can sometimes be undesirable. To guide the reparation process, we impose restrictions on the set of repair operators, increasing the cost of the least restrictive. We have identified four main problems related to the planning task reparation, which we list below along with proposed solutions.

Goals. ADD-FIX actions are defined with zero cost, so it would be trivial to make the task solvable by adding the goal predicate as effect of an action. To address that, we duplicate ADD-FIX actions for each arity n, ψ_{+n}. These actions have the same parameters and effects as the corresponding ψ_{+n} action, but they differ in preconditions and costs. We penalize the application of ψ_{+n}^{g} when the proposition being repaired is a goal, in order to avoid making the problem too simple to solve.

$$\begin{aligned} pre(\psi_{+n}^{g}) =& pre(\psi_{+n}) \cup \{goal\_\langle n\rangle(p, x_1, \ldots, x_n)\} \\ cost\psi_{+n}^{g}) =& C_g \end{aligned}$$

Common ADD-FIX actions ψ_{+n} that consider not-goal atoms maintain zero cost, so a plan that repairs the *clean-shot* action and then prepares the cocktail will have a lower cost than a plan that adds the goal directly.

Adding deleted atoms. In some cases, fixing an action by adding an effect that already coincides with a *del* effect can be incorrect. For example, the action that fills the shot already includes a *(not (clean ?s))* effect, so adding the same effect as a positive would create a conflict. Therefore, we inform the problem with current action effects and introduce two types of FIX ACTIONS: ϕ^- and ϕ^+. Both types have the same general scheme for fix actions, but we restrict in preconditions this kind of reparation.

$$pre(\phi^-) = pre(\phi) \cup \{\neg del\_eff(p, a)\}$$

However, in some domains it is necessary to delete information to update it (for example, the position of an object):

$$pre(\phi^+) = pre(\phi) \cup \{del\_eff(p, a)\}$$

To prioritize these actions, we set $cost(\phi^+) > cost(\phi^-)$.

Repair effects in the same action. The planner is likely to include all necessary positive effects in a single action without considering other actions. To promote the use of the rest of the actions, the set of FIX ACTIONS is further specialized to cover different combinations of delete effects and whether the action has already been fixed or not. This results in four fix actions:

- Actions ϕ_0^-: the predicate symbol used to repair the action does not appear also as a negated effect and the action was not repaired before:

$$pre(\phi_0^-) = pre(\phi^-) \cup \{\neg patched(a)\}$$

- Actions ϕ_1^-: the predicate symbol used to repair the action does not appear also as a negated effect, but the action has already been repaired before:

$$pre(\phi_1^-) = pre(\phi^-) \cup \{patched(a)\}$$

- Actions ϕ_0^+: the predicate symbol is a negated effect of the action, and the action has not been repaired before:

$$pre(\phi_0^+) = pre(\phi^+) \cup \{\neg patched(a)\}$$

- Actions ϕ_1^+: the predicate symbol to repair the action appears as a negated effect and the action was repaired before:

$$pre(\phi_1^+) = pre(\phi^+) \cup \{patched(a)\}$$

The different costs are distributed as $cost(\phi_1^+) > cost(\phi_0^+) > cost(\phi_1^-) > cost(\phi_0^-)$. By using the four defined types of fix actions with different costs, we want to prevent the situations identified as undesirable and to promote the use of all available domain actions.

Use of repaired actions. To avoid excessive use of fixed actions in the plan, we penalize their usage. We limit the applicability of CLOSE ACTIONS ω to only those cases where the action has not been patched before:

$$pre(\omega) = pre(\omega) \cup \{\neg patched(a)\}$$

Additionally, we introduce a new action ω^p for the opposite scenario, where the action being closed is a fixed action:

$$pre(\omega^p) = pre(\omega) \cup \{patched(a)\} \cup \{fixed\}$$

For them $C_\omega = 0$ and $C_{\omega^p} > 0$.

Our objective is to minimize the cost of reparations, while also penalizing the use of repaired actions. By incorporating these diverse costs, we aim to find refined reparations that closely align with the underlying domain, which corresponds to the user's mental model.

After considering the aforementioned bias, the resulting set of action schemes $\mathcal{A}'$ is $\mathcal{A}_\alpha \cup \mathcal{A}_\phi \cup \mathcal{A}_{\psi_+} \cup \mathcal{A}_{\psi_-} \cup \mathcal{A}_\omega$, where $\mathcal{A}_\alpha = \{\alpha_a \mid a \in \mathcal{A}\}$, $\mathcal{A}_\phi = \{\phi_0^-, \phi_1^-, \phi_0^+, \phi_1^+\}, \mathcal{A}_{\psi_+} = \{\psi_{+n}, \psi_{+n}^{g} \mid n \, arity \, of \, p(t) \in \mathcal{P}\}, \mathcal{A}_\omega = \{\omega, \omega^p\}$.

Experiments

We evaluate our approach to provide repair suggestions for an effect-incomplete planning domain D^-. To do so, we assume the existence of an underlying planning task $\Pi = \langle D, I\rangle$ and compare the resulting solutions.

We select seven domains from the International Planning Competition (IPC): TRANSPORT, BLOCKSWORLD, SATELLITE, CHILDSNACK, GOLDMINER, ROVERS and BARMAN, which have 3, 4, 5, 6, 7, 9 and 12 actions, respectively. For each domain, we use a PDDL problem generator[3] to create 10 problem instances of increasing difficulty (Seipp, Torralba, and Hoffmann 2022). We then generate 4 different effect-incomplete tasks for each problem by incrementally deleting random effects from the domain actions. We ensure that each deleted effect made the task unsolvable. In total, we obtain 40 effect-incomplete planning tasks for each domain, with a maximum of 4 deleted effects per task.

[3]https://github.com/AI-Planning/pddl-generators

We solve the generated tasks using the proposed compilation and extract the corresponding repairing set $\hat{R}$ from the resulting plan. We use Fast-Downward (Helmert 2006) planning system, in satisficing and optimal configurations. For the satisficing configuration, we use *lama* (Richter and Westphal 2010). For the optimal configuration, we use *seq-opt-lmcut*. The experiments were run on an Ubuntu machine with Intel(R) Xeon(R) CPU X3470 running at 2.93GHz 16GB memory and a time limit of 30 minutes.

Metrics. We assume the underlying domain D to determine the quality of the solution using *precision* and *recall* as metrics, typically used to measure differences between domain models (Davis and Goadrich 2006). *Precision* $= \frac{tp}{tp+fp}$ computes the number of the reparations correctly placed in the model (*true positives*) taking into account the number of reparations appearing in the generated model that should not appear (*false positives*). *Recall* $= \frac{tp}{tp+fn}$ is similar but for *false negatives*, i.e. it computes the fixes that should appear in the generated action model but are missing. We consider two different ways of computing these metrics, M_1 and M_2:

- In M_1 a reparation is a true positive if the predicate-action pair is correct (i.e., if the predicate *clean* is added as an effect in the *clean-shot* action).
- In M_2 a reparation is a true positive if the predicate is correct (i.e., if the predicate *clean* is added as an effect in *any* action).

Note that M_1 is more restrictive, as it requires to add the predicate as effect to the correct action compared to the underlying domain. We empirically established the following costs to bias the search towards more desirable plans: $cost(\omega^+) = 5$, $cost(\phi_0^-) = 30$, $cost(\phi_0^+) = 50$, $cost(\phi_1^-) = 250$, $cost(\phi_1^+) = 450$, $cost(\psi_n^g) = 150$. This cost setting have been applied to all domains.

The coverage is presented in Table 1, where cardinal numbers used to label the columns (#1, #2, etc.) represent the number of effects removed from the domain. For each domain and number of removed effects, we generate 10 tasks, except for the BLOCKSWORLD domain. In that case, it was not possible to remove 3 and 4 effects simultaneously and still ensure that each of them would make the task unsolvable. Therefore, the total number of tasks in this domain is 9, as indicated to the right of the slash.

Table 2 shows the metric scores for the optimal and satisficing configurations. Total times are not included in the table, but we provide a summary of them in the text. For the optimal configuration, the total coverage is 55.03%, with 50.32% of the solved tasks taking less than 3 seconds to solve, while 24.14% exceeded one minute. In terms of the metrics, for M_1 the average precision and recall are higher than 0.75 in 59.26% and 55.56% of the cases, respectively. For M_2, the average precision and recall are higher than 0.75 in 85.19% and 66.67% of the cases, respectively. This higher precision in M_2 indicates that the correct predicate is often repaired in other actions. For the satisficing configuration, the total coverage is 89.56%, with 67.47% of the solved tasks taking less than 3 seconds to report the best plan in the given time, and only 17.26% exceeding one minute. For M_1 the average precision and recall are higher than 0.75 in 32.14% and 42.86% of the cases, respectively. For M_2, they are higher than 0.75 in 67.86% and 75% of the cases, respectively.

	#1		**#2**		**#3**		**#4**	
Domain	**Opt.**	**Sat.**	**Opt.**	**Sat.**	**Opt.**	**Sat.**	**Opt.**	**Sat.**
TRANSPORT	6	8	5	6	6	6	4	6
BLOCKS	10	9	10	10	8/9	9/9	7/9	9/9
SATELLITE	7	10	6	10	5	10	4	10
CHILDSNACK	4	10	2	10	1	10	0	9
GOLDMINER	10	10	10	10	10	10	10	10
ROVERS	5	10	5	10	4	10	4	9
BARMAN	4	9	2	7	3	7	1	5
Coverage	46	66	40	63	37	62	30	58
#Tasks	70		70		69		69	

Table 1: Coverage results for the benchmark configuration.

Solving the compiled repair task is more difficult for domains with a higher number of actions, as is the case of ROVERS and BARMAN. Since there are many potential ways to repair the domain, both precision and coverage decrease. The low scores in GOLDMINER are due to the fact that problems in this domain have a single goal, which is a 0-arity predicate. During the experiments, when we randomly deleted effects from actions, the action that achieves the goal was broken in most cases (as if the user had forgotten to include actions that generate the goal). Therefore, the cheaper plan consisted of just adding the goal as an effect, even if there were more actions broken.

Regarding planner's performance, the optimal configuration, as expected, produces higher precision but lower coverage. To address the issue of unwanted reparations, we use a cost-based compilation method, which introduces a bias in favor of plans that are more likely to avoid them. However, it is important to note that optimal plans do not guarantee optimal reparations. If rapid responses are a priority and the precision in terms of the predicate-action pair is not a concern, then the satisficing configuration may be a feasible option, as M_2 using this configuration also reports good results.

To evaluate the performance of our approach when negative effects are missing, we conducted some experiments in the TIDYBOT domain, in which it is necessary to remove facts from the state to achieve the goals. We observed that *del-fix* reparations are more likely to be applied when it is inevitable to delete a fact to achieve the goals. Otherwise, reparations will be made by adding positive effects to any of the domain actions.

Related Work

Previous works addressed similar problems in an effort to assist users in designing planning tasks. Some approaches rely on a properly specified domain and problem and also assume an initial plan, which is later modified or improved in collaboration with humans. This is known as *mixed-initiative planning* (Veloso, Mulvehill, and Cox 1997; Howey, Long, and Fox 2004; Chakraborti et al. 2017; Lin and Bercher 2021). When the planning task is unsolvable and there is

		#1		#2		#3		#4	
Conf.	**Domain**	**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**
Opt.	TRANSPORT	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	0.96 ± 0.10	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
		1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
	BLOCKS	0.90 ± 0.32	0.90 ± 0.32	0.95 ± 0.16	0.95 ± 0.16	0.88 ± 0.17	0.88 ± 0.17	0.82 ± 0.17	0.71 ± 0.22
		1.00 ± 0.00	1.00 ± 0.00	0.95 ± 0.16	0.95 ± 0.16	0.96 ± 0.15	0.96 ± 0.12	0.87 ± 0.17	0.83 ± 0.22
	SATELLITE	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
		1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
	CHILDSNACK	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	-	-
		1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	-	-
	GOLDMINER	0.30 ± 0.42	0.40 ± 0.52	0.10 ± 0.21	0.10 ± 0.21	0.10 ± 0.21	0.07 ± 0.14	0.10 ± 0.21	0.05 ± 0.11
		0.30 ± 0.42	0.40 ± 0.52	0.70 ± 0.42	0.40 ± 0.21	0.90 ± 0.21	0.33 ± 0.00	0.90 ± 0.21	0.25 ± 0.00
	ROVERS	0.20 ± 0.45	0.20 ± 0.45	0.90 ± 0.22	0.90 ± 0.22	0.71 ± 0.34	0.67 ± 0.38	0.50 ± 0.33	0.31 ± 0.13
		0.80 ± 0.45	0.80 ± 0.45	1.00 ± 0.00	1.00 ± 0.00	0.83 ± 0.33	0.75 ± 0.32	0.83 ± 0.19	0.56 ± 0.13
	BARMAN	0.50 ± 0.58	0.50 ± 0.58	0.25 ± 0.35	0.25 ± 0.35	0.61 ± 0.10	0.56 ± 0.19	0.00 ± 0.00	0.00 ± 0.00
		0.50 ± 0.58	0.50 ± 0.58	0.50 ± 0.00	0.75 ± 0.35	1.00 ± 0.00	0.89 ± 0.19	1.00 ± 0.00	0.50 ± 0.00
Sat.	TRANSPORT	1.00 ± 0.00	1.00 ± 0.00	0.95 ± 0.13	1.00 ± 0.00	0.88 ± 0.14	1.00 ± 0.00	0.97 ± 0.08	1.00 ± 0.00
		1.00 ± 0.00	1.00 ± 0.00	0.95 ± 0.13	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
	BLOCKS	0.89 ± 0.33	0.89 ± 0.33	0.95 ± 0.16	0.95 ± 0.16	0.62 ± 0.27	0.67 ± 0.29	0.73 ± 0.17	0.64 ± 0.22
		1.00 ± 0.00	1.00 ± 0.00	0.95 ± 0.16	0.95 ± 0.16	0.75 ± 0.20	0.76 ± 0.25	0.89 ± 0.17	0.76 ± 0.20
	SATELLITE	0.87 ± 0.28	1.00 ± 0.00	0.72 ± 0.37	0.90 ± 0.21	0.78 ± 0.29	0.90 ± 0.16	0.91 ± 0.16	0.98 ± 0.08
		0.88 ± 0.25	1.00 ± 0.00	0.95 ± 0.16	1.00 ± 0.00	0.94 ± 0.12	0.97 ± 0.11	0.98 ± 0.06	1.00 ± 0.00
	CHILDSNACK	0.73 ± 0.37	0.90 ± 0.32	0.63 ± 0.34	0.80 ± 0.26	0.44 ± 0.30	0.67 ± 0.27	0.46 ± 0.17	0.75 ± 0.22
		0.85 ± 0.34	0.90 ± 0.32	0.87 ± 0.22	0.90 ± 0.21	0.73 ± 0.28	0.83 ± 0.24	0.85 ± 0.19	0.92 ± 0.18
	GOLDMINER	0.30 ± 0.42	0.40 ± 0.52	0.10 ± 0.21	0.10 ± 0.21	0.10 ± 0.21	0.07 ± 0.14	0.10 ± 0.21	0.05 ± 0.11
		0.30 ± 0.42	0.40 ± 0.52	0.70 ± 0.42	0.40 ± 0.21	0.90 ± 0.21	0.33 ± 0.00	0.90 ± 0.21	0.25 ± 0.00
	ROVERS	0.60 ± 0.52	0.60 ± 0.52	0.65 ± 0.41	0.60 ± 0.39	0.47 ± 0.32	0.47 ± 0.32	0.35 ± 0.16	0.33 ± 0.13
		0.83 ± 0.36	0.90 ± 0.32	0.85 ± 0.34	0.85 ± 0.34	0.67 ± 0.26	0.63 ± 0.25	0.81 ± 0.19	0.64 ± 0.13
	BARMAN	0.28 ± 0.43	0.44 ± 0.53	0.25 ± 0.24	0.43 ± 0.35	0.50 ± 0.40	0.57 ± 0.32	0.23 ± 0.22	0.40 ± 0.38
		0.62 ± 0.46	0.78 ± 0.44	0.33 ± 0.19	0.64 ± 0.38	0.63 ± 0.36	0.81 ± 0.26	0.54 ± 0.27	0.78 ± 0.22

Table 2: Average Precision (P) and Recall (R) scores for optimal (Opt.) and satisficing (Sat.) configurations. Cardinal numbers in the columns indicate the number of removed effects from the domain. Empty cells indicate that no plan was found in the given time window. For each domain, the first and second lines represent the results for metrics M_1 and M_2, respectively.

no suggested plan, some works focus on generating explanations for the unsolvability of a given planning problem. One approach is to identify unreachable subgoals (Sreedharan et al. 2019), derived from abstract and solvable models using planning landmarks (Hoffmann, Porteous, and Sebastia 2004). Another approach, based on counterfactuals theory (Ginsberg 1985), enables authors to explain why a plan fails and how to solve it, but only considers changes to the initial state (Göbelbecker et al. 2010). Planning with incomplete domains or approximate domain models (Garland and Lesh 2002; McCluskey, Richardson, and Simpson 2002; Simpson, Kitchin, and McCluskey 2007; Nguyen, Sreedharan, and Kambhampati 2017) considers domains that are not properly specified, but assumes that the model is filled with annotations or statements about where it has been incompletely specified. Helping users in the modelling of planning tasks has also been addressed through learning approaches, which entails the generation of planning models from a set of plan traces from several plan executions (Aineto, Celorrio, and Onaindia 2019; Lamanna et al. 2021). Related work in HTN domains solves the case when there is no valid decomposition tree via task insertion, including subtasks to refine incomplete methods (Xiao et al. 2020).

Conclusions and Future Work

In this work we present an approach based on automated planning to repair unsolvable planning tasks caused by missing action effects. We propose a compilation of the unsolvable task into a new extended task, which includes operators to repair domain actions by linking them to new positive or negative effects. The solution is a plan that achieves the original goals and incorporates the modifications needed to make the task solvable. The repair process is guided using a cost-based approach that penalizes undesired reparations. Our method is able to generate appropriate reparations in a few seconds in most of the scenarios, making it potentially helpful to support users in designing planning tasks and a significant advancement in the field of explainable planning.

One strength of our approach is that it can obtain a fairly accurate reparation without requiring additional information from the user, only a domain and a single problem. However, this may also have the drawback of generating reparations that are over-fitted to the given problem, compromising the ability to generalize. We believe that incorporating multiple problem instances or automatically generating problems to identify possible flaws in the domain pose interesting challenges that can motivate future research.

Acknowledgements

This work has been partially funded by PID2021-127647NB-C21 and PDC2022-133597-C43 projects, MCIN/AEI/10.13039/501100011033/ and by "ERDF A way of making Europe". Also by the Madrid Government under the Multiannual Agreement with UC3M in the line of Excellence of University Professors (EPUC3M17) in the context of the V PRICIT (Regional Programme of Research and Technological Innovation).

References

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning action models with minimal observability. *Artif. Intell.*, 275: 104–137.

Chakraborti, T.; Sreedharan, S.; and Kambhampati, S. 2020. The Emerging Landscape of Explainable Automated Planning & Decision Making. In *Proceedings of IJCAI 2020*, 4803–4811. ijcai.org.

Chakraborti, T.; Sreedharan, S.; Zhang, Y.; and Kambhampati, S. 2017. Plan Explanations as Model Reconciliation: Moving Beyond Explanation as Soliloquy. In *Proceedings of IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 156–163. ijcai.org.

Davis, J.; and Goadrich, M. 2006. The relationship between Precision-Recall and ROC curves. In *Proceedings of ICML 2006, Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, volume 148, 233–240. ACM.

Fox, M.; Long, D.; and Magazzeni, D. 2017. Explainable Planning. *CoRR*, abs/1709.10256.

Garland, A.; and Lesh, N. 2002. Plan Evaluation with Incomplete Action Descriptions. In *Proceedings of AAAI 2002, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, 461–467. AAAI Press / The MIT Press.

Ginsberg, M. L. 1985. Counterfactuals. In *Proceedings of IJCAI 1985. Los Angeles, CA, USA, August 1985*, 80–86. 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 ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, 81–88. AAAI.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26: 191–246.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered Landmarks in Planning. *J. Artif. Intell. Res.*, 22: 215–278.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *Proceedings of ICTAI 2004, 15-17 November 2004, Boca Raton, FL, USA*, 294–301. IEEE Computer Society.

Kambhampati, S. 2007. Model-lite Planning for the Web Age Masses: The Challenges of Planning with Incomplete and Evolving Domain Models. In *Proceedings of AAAI 2007, July 22-26, 2007, Vancouver, British Columbia, Canada*, 1601–1605. AAAI Press.

Lamanna, L.; Saetti, A.; Serafini, L.; Gerevini, A.; and Traverso, P. 2021. Online Learning of Action Models for PDDL Planning. In *Proceedings of IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, 4112–4118. ijcai.org.

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 IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, 4152–4159. ijcai.org.

Lin, S.; Grastien, A.; and Bercher, P. 2023. Towards Automated Modeling Assistance: An Efficient Approach for Repairing Flawed Planning Domains. In *Proceedings of AAAI 2023, Washington, USA*.

Linares López, C.; Jiménez Celorrio, S.; and García-Olaya, A. 2015. The deterministic part of the seventh International Planning Competition. *Artif. Intell.*, 223: 82–119.

McCluskey, T. L.; Richardson, N. E.; and Simpson, R. M. 2002. An Interactive Method for Inducing Operator Descriptions. In *Proceedings of AIPS 2002, April 23-27, 2002, Toulouse, France*, 121–130. AAAI.

McCluskey, T. L.; Vaquero, T. S.; and Vallati, M. 2017. Engineering Knowledge for Automated Planning: Towards a Notion of Quality. In *Proceedings of K-CAP 2017, Austin, TX, USA, December 4-6, 2017*, 14:1–14:8. ACM.

Nguyen, T.; Sreedharan, S.; and Kambhampati, S. 2017. Robust planning with incomplete domain models. *Artif. Intell.*, 245: 134–161.

Pednault, E. P. D. 1989. ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. In *Proceedings of KR 1989. Toronto, Canada, May 15-18 1989*, 324–332. Morgan Kaufmann.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *J. Artif. Intell. Res.*, 39: 127–177.

Seipp, J.; Torralba, Á.; and Hoffmann, J. 2022. PDDL Generators. https://doi.org/10.5281/zenodo.6382173.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning domain definition using GIPO. *Knowl. Eng. Rev.*, 22(2): 117–134.

Sreedharan, S.; Srivastava, S.; Smith, D. E.; and Kambhampati, S. 2019. Why Can't You Do That HAL? Explaining Unsolvability of Planning Tasks. In *Proceedings of IJCAI 2019, Macao, China, August 10-16, 2019*, 1422–1430. ijcai.org.

Veloso, M. M.; Mulvehill, A. M.; and Cox, M. T. 1997. Rationale-Supported Mixed-Initiative Case-Based Planning. In *Proceedings of IAAI 1997, July 27-31, 1997, Providence, Rhode Island, USA*, 1072–1077. AAAI Press / The MIT Press.

Xiao, Z.; Wan, H.; Zhuo, H. H.; Herzig, A.; Perrussel, L.; and Chen, P. 2020. Refining HTN Methods via Task Insertion with Preferences. In *Proceedings of AAAI 2020, New York, NY, USA, February 7-12, 2020*, 10009–10016. AAAI Press.