Planning Domain Repair Using Preferred Plans and Background Knowledge

Thomas Eckstein
Institute of Software Engineering and Artificial Intelligence, Graz University of Technology, Austria

Gerald Steinbauer-Wagner
Institute of Software Engineering and Artificial Intelligence, Graz University of Technology, Austria

Abstract

Planning domain repair assists developers with the tedious task of finding and fixing errors in planning domains. We present Domain Repair for PDDL (DrPDDL), a novel automated debugger supporting lifted STRIPS domains with typing. As input, it takes a faulty domain, problem instances with corresponding preferred plans – plans the user desires but which may be invalid in the current domain – and optional State Integrity Axioms (SIAs) that must hold across all reachable states. Unlike prior tools, DrPDDL can also repair domains when the preferred plans are already valid. The debugger uses a pipeline of consecutive algorithms. It first relaxes the domain to fix failing action preconditions and SIA violations, and subsequently constricts it as much as possible while ensuring preferred plan validity. DrPDDL can add or remove positive and negative effects, as well as positive preconditions. We evaluated our debugger by automatically inserting random errors into International Planning Competition domains, generating thousands of test instances. The results demonstrate that DrPDDL successfully detects and repairs a high percentage of these errors.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Computing methodologies → Planning and scheduling

Keywords and phrases Planning, Planning Domain Definition Language (PDDL), Debugging, Preferred Plans, Background Knowledge

Digital Object Identifier 10.4230/OASIcs.DX.2026.10

1 Introduction

AI Planning is a field of research dedicated to exploring ways to solve problems commonly occurring in robotics, intelligent agents, and logistics that involve finding a sequence of actions. The Planning Domain Definition Language (PDDL) [5] is a popular language for modeling such problems so that a planner can solve them. The appeal behind planning is that the user can model a domain in a declarative way by stating a set of fluents along with actions and their effects. Planning adds another abstraction layer that eliminates the need for deriving a concrete algorithm for a problem, so that the developer doesn't need to care whether to use a greedy or exhaustive search strategy, for example.

However, since many developers aren't very familiar with planning or declarative programming, they may make mistakes that are hard to find when creating planning domains. We think that the many peculiarities of planning hinder its acceptance and use by a broader community of developers.

Therefore, we want to help developers and students with crafting planning domains by designing a debugger for PDDL. This way, developers can rely on a tool that checks the correctness of a planning domain while also giving suggestions for fixes, relieving the frustration that arises when testing a domain that just doesn't work as intended. So far, existing debuggers for PDDL, such as those designed by [4, 7, 11], can only handle problem instances where no plan can be found at all. These debuggers assume that the intended behavior of the domain is that a given set of problem instances should be solvable. However,

current PDDL debuggers are not able to detect and repair errors when the intended behavior is more complex. It often occurs that a solution is found, but the plan is incorrect because the domain doesn't behave as the developer intended.

For example, we consider a simple packet delivery domain. Multiple locations are connected via a graph, and one or more robots can move between them, as illustrated in Figure 1. Each robot can pick up and carry at most one packet at a time and drop it at another location. The goal is for all packets to be delivered to their destination.

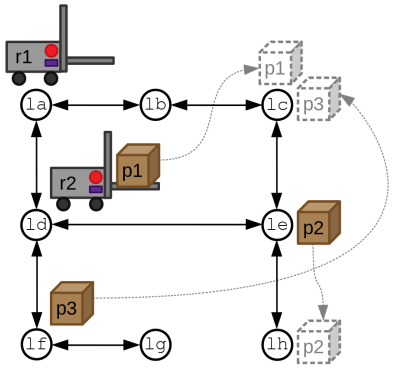

Figure 1 A possible configuration of the delivery robot domain with 8 locations, 2 robots, and 3 packets. Robot `rp` carries packet `p2`. The destinations of the packets stated in the goal are also shown.

In our example, the developer forgot the negative effect `(not (isat ?r ?from))` that removes the robot's old location in the `move` action. The developer tests the domain using a problem instance where the robot starts at location `la` and needs to bring a packet from `lb` back to `la`. The planner gives a solution that is obviously wrong since the robot must move back to `la` after picking up the packet:

```
1  (move r1 la lb)
2  (pickup r1 p1 lb)
3  (drop r1 p1 la)
```

However, the developer prefers to obtain the following plan:

```
1  (move r1 la lb)
2  (pickup r1 p1 lb)
3  (move r1 lb la)
4  (drop r1 p1 la)
```

The domain developer also has the background knowledge that a robot can only be in one location at a time. He wishes to automatically verify whether this property holds for the domain, and otherwise, a fix should be suggested so that the property holds.

This example leads us to the idea of building a debugger for PDDL domains that fixes the domain by not only making problem instances solvable, but also considers additional knowledge of the developer about the domain. The developer should have more freedom in specifying the intended behavior of the domain, allowing a wider range of errors to be found.

In order to assist PDDL developers with writing correct planning domains, our goal is to design and implement a novel debugger for PDDL domains. Instead of an interactive tool like many debuggers found in IDEs, we want a tool that requires little user intervention and finds errors in a domain automatically, while at the same time attempting to repair the domain. In contrast to existing PDDL debuggers, it should be able to detect a wider range of errors, such as missing preconditions, missing negative effects, or unnecessary positive

effects. The debugger should take the domain as input, as well as multiple pairs of problem instances and preferred plans. Additionally, background knowledge about the domain can be provided. The debugger should change the domain so that all given plans are valid and the background knowledge is respected by the domain. One major challenge to overcome is the complexity of planning domains, because the number of possible effects, or preconditions, grows rapidly with each new parameter. When an error is encountered, multiple different adjustments can be made, followed by another set of adjustments, leading to an exponential cascade of solutions. Another obstacle is the large set of features present in modern PDDL.

The main contributions of the paper are a clear notation for PDDL debugging using preferences and background knowledge, a methodology for debugging PDDL if such information is given, and an implementation of the approach as a tool [1].

The remainder of the paper is organized so that we first discuss related work in the validation and debugging of PDDL domains. The next section introduces the notations, considered errors, and basic methodology. This is followed by a detailed description of the developed approach. In the next section, an experimental evaluation with common planning domains is presented, while in the final section we draw some conclusions and outline future work.

2 Related Research

The problem of debugging, validating, and repairing planning domains is an active topic of research with many contributions in recent years. In this section, we present various approaches made towards providing assistance to PDDL developers.

VAL (Plan Validator), originally developed by [9] to verify planners, checks if a given plan solves a specific planning problem. It simulates the initial state and evaluates the plan step-by-step, checking preconditions, applying effects, and verifying the final goal. If a precondition fails, it marks the plan invalid and reports a detailed violation explanation. Although VAL can be repurposed for domain debugging by assuming the plan is correct and the domain is faulty, it lacks automation; a human developer must still manually determine and apply the necessary domain repairs.

Another domain repair tool, presented by [4], addresses planning problems that fail due to over-constrained preconditions. Using a domain and problem instances as input, it iteratively modifies preconditions to relax the domain until a solution is found. Driven by the Delta Debugging algorithm, the tool alters precondition components and invokes a planner to find a local minimum of removed constraints. This black-box approach requires neither human intervention nor preferred plans, allowing it to support advanced PDDL features like temporal planning. However, it is strictly limited to fixing a single error type: additional preconditions.

Domain repair for incomplete positive effects was addressed by [8]. When a planning problem has no solution, their algorithm adds positive effects to relax the domain and make the goal reachable. They encode this repair process as a planning problem itself, using auxiliary actions and fluents that dynamically modify actions. The planner returns both a valid plan and the required domain modifications. To counter the planner's natural bias toward excessive repair (adding extraneous effects to shorten plans), the authors penalize each repair made. A user interface was later introduced in [7].

The most advanced debugger we are aware of was presented by [10] and [11]. In their first paper, they mapped domain repair for grounded STRIPS domains to a diagnosis problem solved via a Minimal Hitting Sets algorithm. The debugger relaxes the domain to validate

[1] The implementation is publicly available under `https://git.ist.tugraz.at/ais/pddl-debugger`

an incorrect preferred plan by removing preconditions, adding positive effects, or removing negative effects. In their second paper, they extended the tool to support lifted STRIPS domains, multiple problem-plan pairs, and negative preconditions (allowing the removal of negative preconditions/positive effects or the addition of negative effects). However, this debugger still fails to address scenarios where the preferred plan is already valid, but the planner returns a different, incorrect plan as the solution.

An exploratory study by Gragera and Pozanco investigated the utility of Large Language Models (LLMs) like ChatGPT for debugging planning domains. In each test case, they prompted the LLM with an unsolvable planning problem – consisting of a domain and a problem instance – alongside a brief task description. Their findings indicated that, given their current capabilities, LLMs fail to provide meaningful assistance to planning domain developers.

Another notable tool created to assist PDDL developers is PDDL Studio [15]. It is a complete IDE for PDDL that is capable of syntax highlighting and marking compile errors. Furthermore, it provides a graphical interface for various planners. A similar collection of tools is available as an add-on for the popular text editor Visual Studio Code [3]. It also integrates features from planutils, a collection of tools for planning created by Muise [14].

An exploratory study by Sleath et al. systematically cataloged PDDL domain errors to evaluate PDDL parsers, creating a repository of both syntactical and semantical mistakes [17]. Syntactic errors include basic formatting slips like missing closing parentheses, while semantic errors cover logical flaws such as attempting to access an undefined variable. However, this repository fails to capture higher-level design errors, such as missing effects or redundant preconditions.

Regarding integrity constraints for situation calculus, the work presented in [16] has laid the groundwork for proving that properties hold for all situations reachable from the initial situation using induction as a proof technique.

3 Methodology

3.1 Basic Notations

Figure 2 depicts the possible results of running a planner on a planning problem. The planner may **fail** in providing meaningful results due to internal software errors, syntax errors in the domain model, or algorithmic errors. The planner may not be able to provide meaningful results due to **limited computational resources** or domain complexity. The planner may terminate normally (**completes**) and present a valid result regarding the given planning problem. The planner completes and presents a valid plan for the problem (**solves**). The planner completes and states that no valid plan exists for the given planning problem (**rejects**).

A plan is **valid** if it fulfills all requirements given by the domain and problem description, meaning that at each step, the respective preconditions hold and the goals are met in the end. Conversely, a plan is **invalid** if a precondition is violated or the goal is not met. A plan is **correct** if it is the solution to a concrete planning problem and can be executed in the corresponding world showing the intended behavior. An **incorrect** plan is either infeasible or does not achieve the goals intended in the corresponding world. For example, a robot may try to pick up an object that is not in place. Please note that a valid plan isn't necessarily correct, while a correct plan isn't necessarily valid given a domain model.

Figure 3 illustrates how relaxing or constricting a domain shifts its boundaries, allowing us to formally define soundness and completeness in PDDL modeling. A domain is **sound** if all valid plans are correct, and it is **unsound** if an invalid plan is permitted. **Constricting** a

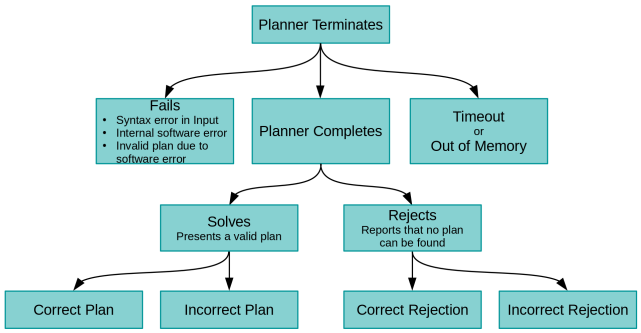

Figure 2 The potential outcomes of running a planner on a planning problem.

domain by adding preconditions, removing positive effects, or adding negative effects restricts the set of valid plans and increases soundness. Conversely, a domain is **complete** if all correct plans are valid, and it is **incomplete** if a correct plan is rejected. **Relaxing** a domain by removing preconditions, adding positive effects, or removing negative effects expands the valid plan space and increases completeness.

A domain is correct only when it is simultaneously sound and complete. If it is incorrect, modeling errors cause it to report incorrect plans or false rejections, failing to accurately reflect the real-world behavior or the modeler's intent.

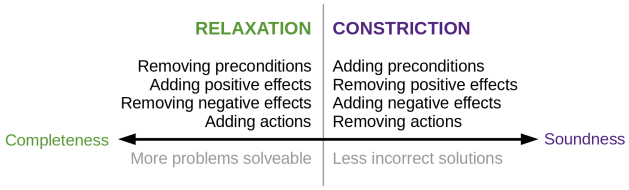

Figure 3 Relation of relaxation and constriction of a domain to changes in the domain.

3.2 Error Catalog and Fixes

We identified 3 error types related to modeling: (1) syntax errors in the domain model and the problem instance, (2) errors in the problem instance, and (3) errors in the domain model. We assume that the first type is handled by the planner. The second type is beyond the scope of this work to use preferred plans and background knowledge for domain debugging. Table 1 provides a list of modeling errors we aim to automatically debug, along with a detailed description and the potential consequences.

We define a **fix** as an atomic modification to a domain. We focus on altering only the preconditions and effects of domains. There are other types of fixes, such as altering parameters of actions, adding or removing fluents, and creating or deleting actions. This is beyond the scope of this work. Applying a fix fix to a domain d gives a new domain d': $d' = apply\_fix(d, fix)$ We also allow multiple fixes to be applied to a domain: $d' = apply\_fix(apply\_fix(d, fix_1), fix_2) = apply\_fix(d, \{fix_1, fix_2\})$

The application of a fix to a domain is idempotent, meaning that the domain doesn't change if the same fix is applied one more time: $apply\_fix(d, fix) = apply\_fix(apply\_fix(d, fix), fix)$

Table 2 lists the types of fixes considered and the notation used to denote them.

Table 1 Catalog of considered modeling errors and their consequence on the planning results: domain relaxation (RE), domain constriction (CO), jeopardize domain correctness (CR).

error type	error description	consequence
precondition missing	a literal is missing in the precondition	RE
additional precondition	an additional incorrect precondition is present	CO
positive effect missing	a positive literal is missing in the precondition	CO
additional positive effect	an additional incorrect positive literal is present	RE
additional negative effect	an additional incorrect negative literal is present	CO
negative effect missing	a negative literal is missing in the precondition	RE
action parameter missing	a action parameter is missing and most likely a precon.	RE
fluent missing	a fluent is missing and modeling of the related impact	CC
action missing	an action is missing and modeling of the related behavior	CC
parameter mismatch precon.	wrong parameter used in precondition	CC
parameter mismatch effect	wrong parameter used in effect	CC
incorrect effect polarity	mismatch between positive and negative effect	CC

Table 2 The types of fixes considered, along with the notation we introduce to denote them.

Fix	Meaning
$\langle a.C + c \rangle$	Adding precondition c to the action a.
$\langle a.C - c \rangle$	Removing precondition c from the action a.
$\langle a.E^+ + e \rangle$	Adding positive effect e to the action a.
$\langle a.E^+ - e \rangle$	Removing positive effect e from the action a.
$\langle a.E^- + e \rangle$	Adding negative effect e to the action a.
$\langle a.E^- - e \rangle$	Removing negative effect e from the action a.

3.3 Background Knowledge

We express background knowledge as a set of state integrity axioms (SIA) that hold if a situation is consistent in the modeled world. If an action is taken in a consistent situation leading to an inconsistent situation, then an error must have been made in modeling that action. Background knowledge is intrinsic to the planning problem itself, but it isn't necessary to incorporate it in the planning domain directly, as discussed by Mueller [13]. For instance, it is reasonable that a robot can only be in one location at once in the delivery robot domain.

We state that a domain model D is incorrect if there exists a valid plan that leads to a state where an SIA doesn't hold. The SIA must hold for all valid plans. This means that for all valid states, all possible actions must lead to another state for which the SIA holds, which is similar to concepts presented by Reiter [16].

$$\begin{aligned} &\mathit{correct}(D) \rightarrow [\forall I, G, \pi : \mathit{valid}(D, I, G, \pi) \\ &\rightarrow \forall a_i \in \pi, s_i : s_i = \gamma(s_{i-1}, a_{i-1}) \wedge \mathit{SIA}(s_i)] \end{aligned} \tag{1}$$

An SIA can be expressed using a syntax similar to logic expressions in PDDL of the form `(operator (fluent1 term1_1 term1_n) (fluentN termN_1 termN_N))`. For instance, the SIA `(always (f t1 t2))` encodes the constraint that at least one atomic formula of `f` needs to be part of the state S: $\forall t_1 \in O : \exists t_2 \in O : \langle f, t_1, t_2 \rangle \in S$. An example is `(always (isat r1 ?))` encodes that robot `r1` needs to be at least at one location at all times, where `?` encodes a wildcard. Consistency checks can be executed efficiently using the techniques proposed by Giacomo and Mancini [2]. Besides `always`, the debugger supports `not`, `binary`, `imlies`, `nand`, `xor`, and `and` in SIA.

Hypotheses for the violation are reported as a set of tuples $\langle positive, prev, modified \rangle$ where *positive* is true if the atom is present, but shouldn't be, and vice versa, *prev* holds the atom if established already before the evaluated action, and *modified* holds the atom if established by the evaluated action. For instance, if in the action `move` the positive effect for the target plocation is missing, the following hypotheses are given: $\{\langle positive : \bot, modified : \texttt{(at r1 la)} \rangle, \langle positive : \bot, prev : \texttt{(at r1 ?)} \rangle\}$.

3.4 PDDL Debugger

DrPDDL takes a PDDL domain, a set of State Integrity Axioms, a list of at least one pair of a problem instance and preferred plan as well as a set of fixes to ignore as input. It first checks whether any action preconditions or SIAs are violated when evaluating the plans. It reports these violations and attempts to fix them by adding or removing preconditions and effects. Under the assumption that any behavior outside of the behavior represented in the preferred plans is unwanted behavior, DrPDDL constricts the domain as much as possible while still ensuring that the plans are valid and no SIAs are violated. Finally, it gives a list of fixes as output, as shown in Algorithm 1.

Currently, our debugger is limited to PDDL domains, making use of only the `strips` and `typing` requirements. It also cannot add new actions, remove actions, or add new parameters to actions, since new parameters would require the provided plans to be revised during the process of debugging them. It is also limited to debugging planning domains only. It cannot debug problem instances by explaining how the initial situation needs to be altered for the goal to become achievable.

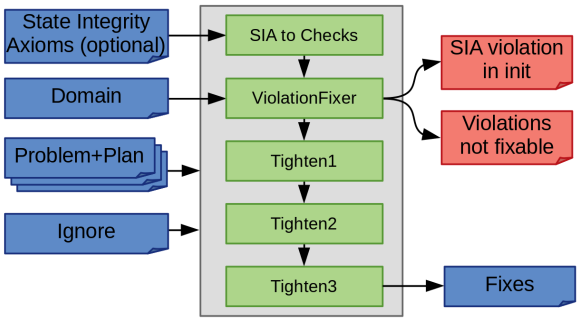

Figure 4 A flowchart illustrating the input, output, and components of the proposed debugger.

Internally, the debugger makes use of various algorithms, as shown in Figure 4. At first, it loads and parses the input. The SIAs are translated into a list of checks. A check is a short procedure that is executed after each step in the plans and checks for a specific change regarding an SIA. If a violation is found, hypotheses about the cause of the violation are reported.

The ViolationFixer is responsible for changing a domain so that it doesn't violate any preconditions of the plans or SIAs during the evaluation of those plans. While fixing violations of preconditions leads to a domain relaxation, fixing a violation of an SIA may leads to either a relaxation or a constriction. It uses a breadth-first search strategy where the plans are validated, and when a violation is encountered, one or more fixes are proposed. It succeeds if a set of fixes is found so that all plans are valid, and it fails if all branches have been searched without success.

Algorithm 1 Main function of our debugger that calls all core algorithms, incrementally refining the domain.

```
1: function Main(d, plans, SIAs, ignore)
2:     checks ← SiaToChecks(SIAs)
3:     d ← ViolationFixer(d, plans, checks, ignore)
4:     d ← Tighten1(d, plans, checks, ignore)
5:     d ← Tighten2(d, plans, checks, ignore)
6:     d ← Tighten3(d, plans, checks, ignore)
7:     return d
8: end function
```

ViolationFixer (Algorithm 2) takes a domain d, problem-plan pairs *plans*, a set of checks *checks*, and a set of ignored fixes *ignore* as parameters. It initializes a queue with the initial domain d and creates a set for visited domains. The algorithm loops until the queue is empty, extracting a domain d' from the front. If d' was already visited, it is ignored. Otherwise, the plans are validated using d'. If the plans are valid, the algorithm terminates and returns d'. Otherwise, it calculates possible fixes for the violation, applies them to d', and pushes the resulting domains back to the queue. This breadth-first search strategy inherently favors domains requiring fewer fixes. If the queue becomes empty because no further fixes can be generated, the algorithm fails. Figure 5 outlines this control flow.

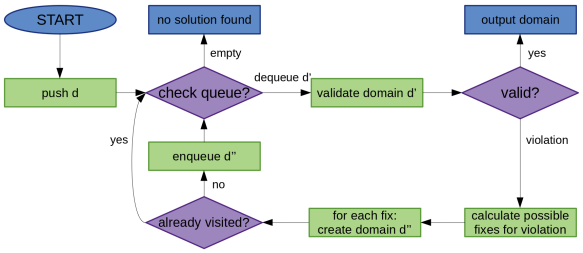

Figure 5 A flowchart showing the steps of the ViolationFixer algorithm.

The method for calculating fixes is determined by the specific type of violation encountered. When a violation occurs within a check, the algorithm calls the function CalcFixes1 for each component of that violation. This function generates distinct sets of fixes, each of which is applied to the domain before being added to the queue.Alternatively, if the violation is found within a goal or a precondition, the fixes are determined by the function CalcFixes2. Furthermore, if the issue stems specifically from a precondition, an additional domain where that precondition has been entirely removed is also generated and enqueued.

In order to derive fixes from these components, the following rules are applied, as reflected in Algorithm 3 in the same order. If an atom is already present, a negative effect of removing it is introduced. For example, if the SIA (xor A B) is violated because A was already set when B is applied, a possible fix is adding the effect (not A). If an atom were added to the violating action, the positive effects of creating it are removed. In the (xor A B) example, another fix is removing the effect (B) so only A remains. If both cases above are true, the Cartesian product of the fixes is returned, applying both a negative effect addition and a positive effect removal simultaneously. If B and A" are set while A' is present, the fix set is $\{\langle a.E^- + \text{`(not A')`}\rangle, \langle a.E^+ - \text{`(A")`}\rangle\}$. If an atom is already absent, either a positive effect creating it or a precondition testing for it is added. For example, if (xor A B) is violated when B is removed because A is absent, a fix is adding the positive effect (A). If an atom was removed in the violating action, either the negative effect removing it is deleted, or a positive

Algorithm 2 ViolationFixer. It takes a domain, a list for problem-plan pairs, a list of checks and a list of fixes to ignore as parameters and returns a list of fixes necessary to uphold all action preconditions and SIAs.

```
1: function ViolationFixer(d, plans, checks, ignore)
2:   queue ← ⟨⟩
3:   enqueue(d)
4:   searched ← ∅
5:   while d′ = dequeue() do
6:     if d′ ∈ searched then
7:       continue
8:     end if
9:     searched ← searched ∪ {d′}
10:    result ← Validator(d′, plans, checks)
11:    if result = Valid then
12:      return d′
13:    end if
14:    if result occured in check then
15:      for all c in result.violation.components do
16:        for all f in CalcFixes1(c, result.s) do
17:          enqueue(apply_fix(d′, f \ ignore))
18:        end for
19:      end for
20:    else
21:      r ← result.precondition(result.s)
22:      for all f in CalcFixes2(d′, result.s, r) do
23:        enqueue(apply_fix(d′, f \ ignore))
24:      end for
25:      if result occured in precondition then
26:        enqueue(apply_fix(d′, {⟨result.s.a.C−
27:        result.pc⟩} \ ignore))
28:      end if
29:    end if
30:  end while
31:  return Null
32: end function
```

effect adding an alternative atom is introduced (e.g., removing (not B) or adding (B')). If both absent/removed cases are true, the Cartesian product of those fixes is returned. If (or A B) is violated because A is absent when the action removes B, and a "blind" negative effect also removes A, the fix set is $\{\langle a.E^- - \text{`(not A)'}\rangle, \langle a.E^+ + \text{`(A)'}\rangle\}$.

When an action precondition or planning goal is violated, the function in Algorithm 3 seeks fixes to satisfy it by traversing the plan in reverse, starting immediately before the violation. At each step, the algorithm attempts to introduce a positive effect that creates the required atom. However, if it encounters an action containing a negative effect that explicitly removes that atom, an alternative fix is generated to delete that negative effect (regardless of whether the atom was present prior to that action). Once this occurs, the reverse loop terminates and returns the accumulated set of fixes.

The Tighten1 algorithm, outlined in Algorithm 5, constricts a domain by removing as many positive effects as possible without compromising its validity. The simplest approach would be to eliminate all positive effects whose created atoms are never subsequently queried by a precondition. However, multiple positive effects can generate the exact same atom for a downstream precondition, meaning redundant effects can be safely removed. As illustrated in Figure 6 and Figure 7 left, this challenge can be transformed into a Minimal Hitting Set problem. Here, each evaluated precondition contributes a set containing all positive effects capable of creating its required atom. By solving for the minimal hitting set, the algorithm identifies the absolute minimum collection of effects necessary to satisfy all preconditions. Any positive effects outside this set are then removed from the domain.

Algorithm 3 Function CalcFixes1 used by the ViolationFixer algorithm. It takes a component v of a violation and a step s and returns a set of all fixes that can be used to prevent the violation.

```
1: function CalcFixes1(v, s)
2:   if v.positive then
3:     if v.prev ≠ Nil ∧ v.mo = Nil then
4:       F = {{⟨v.a.E⁻ + x⟩}|x ∈ EC(v.a, v.p)}
5:       return F
6:     else if v.prev = Null ∧ v.m ≠ Null then
7:       F = {{⟨v.a.E⁺ − e⟩}|e ∈ v.a.E⁺ ∧
8:       e(s) = v.m}
9:       return F
10:    else if v.prev ≠ Null ∧ v.m ≠ Null then
11:      F1 ← {⟨v.a.E⁻ + x⟩|x ∈ EC(v.a, v.p)}
12:      F2 ← {⟨v.a.E⁺ − e⟩ |e ∈ v.a.E⁺ ∧
13:      e(s) = v.m}
14:      return F1 × F2
15:    end if
16:  else
17:    if v.prev ≠ Nil ∧ v.m = Nil then
18:      F1 ← {{⟨v.a.E⁺ + x⟩}|x ∈ EC(v.a, v.p)}
19:      F2 ← {{⟨v.a.C + x⟩}|x ∈ EC(v.a, v.p)}
20:      return F1 ∪ F2
21:    else if v.prev = Nil ∧ v.modified ≠ Nil then
22:      F1 ← {{⟨v.a.E⁻ − e⟩}|e ∈ v.a.E⁻ ∧
23:      e(s) = v.m}
24:      F2 ← {{⟨v.a.E⁺ + x⟩}|x ∈ EF(v.a, v.m)}
25:      return F1 ∪ F2
26:    else if v.prev ≠ Nil ∧ v.m ≠ Nil then
27:      F1 ← {⟨v.a.E⁺ + x⟩|x ∈ EF(v.a, v.p)}
28:      F2 ← {⟨v.a.C + x⟩|x ∈ EF(v.a, v.p)}
29:      F3 ← {⟨v.a.E⁻ − e⟩|e ∈ v.a.E⁻ ∧
30:      e(s) = v.m}
31:      return (F1 ∪ F2) × F3
32:    end if
33:  end if
34: end function
```

Algorithm 4 Function CalcFixes2 used by the ViolationFixer algorithm. It takes a domain, step and atom queried by a precondition as argument and returns a list of fixes in order for the precondition to hold.

```
1: function CalcFixes2(d, s, r)
2:   F ← ∅
3:   while s.num > 0 do
4:     s ← s − 1
5:     if ∃e ∈ s.a.E⁻ : e(s) = r then
6:       F ← F ∪ {{⟨s.a.E⁻ − e⟩}}
7:       break
8:     end if
9:     for all x in EE(r, s.args) do
10:      fixes ← fixes ∪ {{⟨s.a.E⁺ + x⟩}}
11:    end for
12:  end while
13:  return F
14: end function
```

Using a Minimal Hitting Set algorithm, it is also possible to mark effects that must be included in the resulting hitting set, as shown in Figure 7 right. We use this feature to mark all effects contained in the ignore set as well as all effects in the initial situations, since they should not be removed. In the end, the resulting effects are removed one by one, with a call to the Validator algorithm being made each time. If the removal of one effect results in an invalid domain, the removal will be reversed.

The function ExtendPlan simplifies the evaluation of the initial situation and goal. It constructs a new action init from the initial situation of a given problem and prepends it to the beginning of the plan. This action has no parameters or preconditions, and the atoms of the initial situation make up its positive effects. Next, it constructs another action `goal` from the problem's goal and appends it to the end of the plan. It has no parameters or effects, but uses the planning goal as its precondition. Finally, the function returns the new plan, an approach also used in plan-space planning [6].

Algorithm 5 TIGHTEN1. It takes a domain, a list of problem-plan pairs, a list of checks compiled from the SIAs and a set of changes to be ignored and it then removes as many positive effects as possible from the domain.

```
1: function TIGHTEN1(d, plans, checks, ignore)
2:   sets ← ∅
3:   for all ⟨problem, plan⟩ in plans do
4:     for all e in problem.init do
5:       ignore ← ignore ∪ {e}
6:     end for
7:     plan ← EXTENDPLAN(plan, problem)
8:     S ← ∅
9:     for all s in plan do
10:      for all c in s.a.C do
11:        r ← c(s)
12:        sets ← sets ∪ S_r
13:      end for
14:      for all e in s.a.E do
15:        r ← e(s)
16:        if e is positive then
17:          S_r ← S_r ∪ {e}
18:        else
19:          S_r ← ∅
20:        end if
21:      end for
22:    end for
23:  end for
24:  removeable ← d.E⁺ \ MHS(sets, ignore)
25:  for all e in removeable do
26:    d′ ← apply_fix(d, ⟨e.a.E⁺ − e⟩)
27:    if VALIDATOR(d′, plans, checks) then
28:      d ← d′
29:      yield ⟨e.a.E⁺ − e⟩
30:    end if
31:  end for
32: end function
```

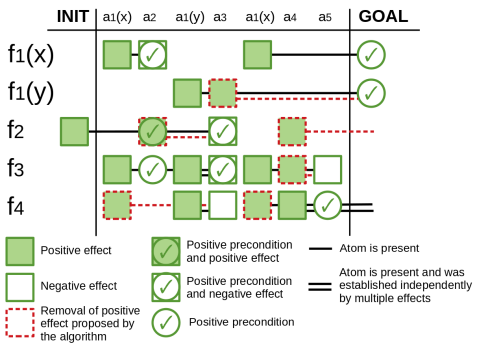

Figure 6 Example scenario demonstrating the TIGHTEN1 algorithm.

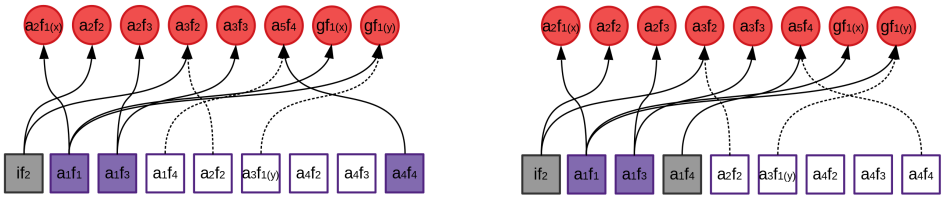

Figure 7 Left: The graph constructed from the scenario illustrated in Figure 6 as well as a possible set of effects that can be removed, marked with white background. Effects in the initial situation are by default blacklisted. Right: Another possible solution for the scenario in Figure 6 with the effect $a_1 f_4$ also blacklisted this time. Another possible solution for the example scenario with one additional effect blacklisted.

The TIGHTEN2 algorithm is a greedy procedure that adds as many negative effects to a domain as possible to constrict it. To preserve plan validity, effects are introduced only where existing atoms are present but not subsequently required by a precondition. The algorithm uses a heuristic based on the proximity of removable atoms to the latest preconditions to select the best candidate effect. Once added, the VALIDATOR checks for State Integrity Axiom (SIA) violations. If a violation is found, the effect is removed and placed in an ignore set. As defined in Algorithm 6, this process repeats until no further effects can be found. Figure 8 illustrates this behavior for a hypothetical domain and plan.

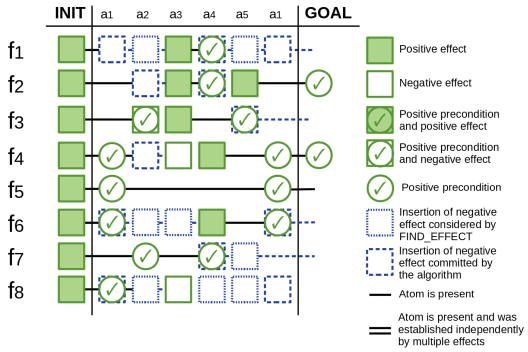

Figure 8 A hypothetical domain and plan showing the set of negative effects the TIGHTEN2 algorithm adds.

In each iteration, the most suitable negative effect to be added is calculated using the function in Algorithm 7. This function utilizes mutable records passed by reference. First, the EXPRENUMERATOR calculates the entire set of addable negative effects for all actions and fluents. For each effect, a record is created containing the effect itself, a score initialized to zero, and a validity field initialized to $\top$. Finally, these records are stored in a list and bucketized by their corresponding action.

Thereafter, the algorithm iterates over each problem–plan pair and each step of the plan, including the initial situation and goal state. For every atom in the current situation, it stores the creation step, the step in which it was last used as a precondition, and a list of encountered candidate negative effects. At each step, the action's preconditions are evaluated. When an atom is queried, its last-use index is updated, and any previously encountered candidate negative effects for that atom are marked invalid, since they would remove the atom and invalidate the plan. The action's effects are then applied. Positive effects create new atom records with the current step as their creation and last-use index, while negative effects

remove the corresponding atom records. Finally, candidate negative effects of the action are considered. If the resulting atom exists in the current situation and was created before the current step, the candidate is added to the atom's effects. Its score is also increased according to a heuristic based on the distance to the atom's last use, with higher scores assigned to atoms removed shortly after their last use. Finally, the candidate with the highest score is picked out of the set of all valid candidates with scores higher than 0. If no such candidate exists, NULL is returned.

Algorithm 6 TIGHTEN2 algorithm. It takes a domain, a list of problem-plan pairs, a list of checks compiled from the SIAs and a set of changes to be ignored and adds as many negative effects as possible to the domain.

```
1: function TIGHTEN2(d, plans, checks, ignore)
2:   while e ← FINDEFFECT(d, plans, checks, ignore) do
3:     d′ ← apply_fix(d, ⟨e.a.E⁻ + e⟩)
4:     if VALIDATOR(d′, plans, checks) then
5:       d ← d′
6:       yield ⟨e.a.E⁻ + e⟩
7:     else
8:       ignore ← ignore ∪ {e}
9:     end if
10:  end while
11: end function
```

The TIGHTEN3 algorithm (Algorithm 8) adds as many preconditions as possible while preserving the validity of the given plans. It processes all plans and computes, for each action occurrence, the set of preconditions that could be added based on the current situation. These candidate sets are grouped by action. For each action, all collected sets are intersected, yielding the preconditions that hold whenever the action is executed. A precondition is reported unless it is already present in the action, belongs to the goal, or is contained in the *ignore* set. Figure 9 shows how the TIGHTEN3 algorithm adds preconditions in a hypothetical scenario as well as the resulting data structure.

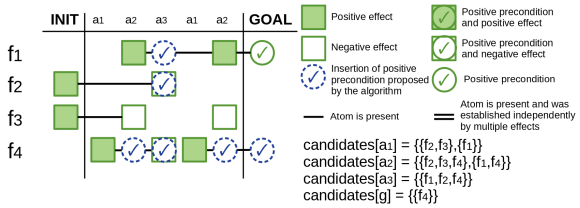

Figure 9 An example plan and domain showing the set of preconditions the TIGHTEN3 algorithm can add. The resulting data structure is also shown. For each step, the set of all possible additional preconditions is calculated that is then bucketized by action. The result is obtained using the intersection of the sets of preconditions within each bucket. Example scenario demonstrating the Tighten4 algorithm.

4 Evaluation

4.1 Metrics

To evaluate the effectiveness of our proposed PDDL debugger, we first define appropriate metrics. We evaluate how many errors our debugger can detect and repair in a faulty domain. Each testcase consists of a domain containing a set of errors M, for which the debugger

Algorithm 7 The FINDEFFECT function is used by the TIGHTEN2 algorithm. It calculates all sensible negative effects that can be added to a domain and returns the best-suited fix.

```
1: function FindEffect(d, plans, checks, ignore)
2:   candidates ← ∅
3:   for all a in d do
4:     for all f in d.F do
5:       for all x in ExprEnumerator(a, f) do
6:         candidates_a ← candidates_a ∪ {⟨e : x, score : 0, valid : ⊤⟩}
7:       end for
8:     end for
9:   end for
10:  for all ⟨problem, plan⟩ in plans do
11:    plan ← ExtendPlan(plan, problem)
12:    S ← ∅
13:    for all s in plan do
14:      for all c in s.a.C do
15:        r ← c(s)
16:        S_r.last_use ← s.num
17:        for all e in S_r.encountered do
18:          e.valid ← ⊥
19:        end for
20:      end for
21:      for all e in s.a.E do
22:        r ← e(s)
23:        if e is positive then
24:          S_r ← ⟨origin : s.num, last_use : s.num, encountered : ∅⟩
25:        else
26:          remove S_r
27:        end if
28:      end for
29:      for all candidate in candidates_a do
30:        r ← candidate.e(s)
31:        if r ∈ S ∧ S_r.origin < s.num then
32:          add candidate to S_r.encountered
33:          candidate.score ← candidate.score + 1/(s.num − S_r.last_use + 1)^2
34:        end if
35:      end for
36:    end for
37:  end for
38:  remove from candidates where ¬candidate.valid ∨ candidate.score = 0
39:  if candidates = ∅ then
40:    return Null
41:  else
42:    return argmax_{candidate∈candidates} candidate.score
43:  end if
44: end function
```

returns a set of fixes F. The correctly identified errors are the true positives $M \cap F$. Since the number of errors differs between testcases, we measure performance using the recovery rate *rec* defined in Definition 1. False positives are given by $F \setminus M$. We report the false positive rate from Definition 2, which measures the fraction of reported incorrect fixes. Additionally, we record the runtime t required to process each testcase.

▶ **Definition 1.** *The recovery rate is defined as the rate of correctly identified errors among the introduced errors.*

$$rec = \frac{|M \cap F|}{|M|}$$

▶ **Definition 2.** *The false positive rate is defined as the rate of false positives among all reported errors. If no errors are reported, fp is 0.*

$$fp = \frac{|F \setminus M|}{max(|F|, 1)}$$

Algorithm 8 TIGHTEN3. It takes a domain, a list of problem-plan pairs, a list of checks compiled from the SIAs and a set of changes to be ignored as input and adds as many preconditions as possible to the domain.

```
 1: function TIGHTEN3(d, plans, checks, ignore)
 2:   buckets ← ∅
 3:   for all ⟨problem, plan⟩ in plans do
 4:     plan ← EXTENDPLAN(plan, problem)
 5:     S ← ∅
 6:     for all s in plan do
 7:       for all r in S do
 8:         fixes ← FIXENUMERATOR(r, s.args)
 9:         buckets_{s.a} ← buckets_{s.a} ∪ {fixes}
10:       end for
11:       apply s to S
12:     end for
13:   end for
14:   for all ⟨a, candidates⟩ in buckets do
15:     matches ← ⋂_{candidate ∈ candidates} candidate
16:     for all c in matches do
17:       if c ∉ a.C ∧ c ∉ ignore then
18:         yield ⟨a.C + c⟩
19:       end if
20:     end for
21:   end for
22: end function
```

When running a large number of similar testcases, we can just calculate the average recovery rate rec_μ as well as the average false positive rate fp_μ. Similarly, we calculate the average runtime t_μ and its standard deviation t_σ.

To evaluate the core design decisions, we investigate the impact of background knowledge on result quality, determine which error types are easiest to detect and repair, assess the benefit of preemptively blacklisting false positives, analyze how the recovery rate varies with the number of errors present, and compare the results for domains that are valid and invalid with respect to the preferred plans. The last question targets scenarios where a preferred plan is valid, but the planner returns an incorrect solution. To test these, our debugger's interface allows adjusting the following meta-parameters: the sequence of algorithms used, whether to use all, some, or no SIAs, and which fixes to exclude from the search space. To answer these questions, we considered the following debugger configurations:

- **normal**: All plans available, all algorithms used, all SIA used, no fixes excluded.
- **no tighten**: All plans available, only VIOLATIONFIXER used, all SIA used, no fixes excluded.
- **no SIA**: All plans available, all algorithms used, no SIAs used, no fixes excluded.
- **with ignored**: First, the debugger is executed the same way as in **normal**. Then, the set of false positives is calculated, and the debugger is run again with that set being excluded.
- **single**: Only one plan available at a time, all algorithms used, all SIA used, no fixes excluded. This results in the number of runs being multiplied by the number of plans available for that domain.

4.2 Setup

We rely on realistic domains written by various developers. The archive of the International Planning Competition (IPC) [12] offers a large set of domains of various complexities. It also contains dozens of problem instances for each domain. The more challenging aspect is to introduce errors into those domains that are likely to occur when the domain is written by a human. We filter out all domains with PDDL features not supported by DrPDDL, meaning that we only keep domains that only have both `strips` and `typing` as requirements.

Subsequently, we use the Optic planner [1] to generate plans for these domains. We discard any domain where fewer than 10 problem instances yielded valid plans due to timeouts, further narrowing down the set of suitable domains. Then we sort out domains that are very similar in their structure and purpose, such as `driverlog-strips-hand-coded` and `driverlog-numeric-hard-automatic`, narrowing down to 10 domains listed in Table 3. We manually specify the background knowledge for each domain. To make sure that background knowledge is correct, we validate each domain using the previously generated plans.

Table 3 The domains used along with metrics regarding their complexity. It shows the name, the number of actions $|A|$, number of fluents $|F|$, total number of preconditions $|C|$, the total number of positive and negative effects $|E^+|$ and $|E^-|$ respectively, the number of steps in all plans $|P|$ as well as the number of SIAs $|SIA|$.

Domain Name	$\|F\|$	$\|A\|$	$\|C\|$	$\|E^+\|$	$\|E^-\|$	$\|P\|$	$\|SIA\|$
2000-blocks-strips-typed	5	4	9	9	9	336	8
2000-freecell-strips-typed	11	10	61	26	30	243	7
2000-logistics-strips-typed	3	6	12	6	6	360	6
2002-driverlog-strips-automatic	6	6	14	7	7	303	6
2002-satellite-strips-automatic	8	5	15	5	4	398	3
2002-zenotravel-strips-automatic	4	5	14	7	7	189	6
2006-pipesworld-propositional	15	6	57	26	26	346	12
2006-rovers-propositional	25	9	45	17	13	313	12
2006-storage-propositional	8	5	18	10	10	158	8
2006-tpp-propositional	7	4	17	7	7	566	3

To test a range of error types in each domain, we decided to automatically add errors, allowing for the generation of hundreds of faulty mutants from one domain.

Regarding the types of errors, we will use the error catalog presented in Section 3 as a reference. In our quantitative evaluation, we will only consider errors our debugger is suited for, which are: **pre rem** - missing precondition, **pre add** - additional precondition, **pre rep** - error in a precondition, **pos rem** - missing positive effect, **pos add** - additional positive effect, **pos rep** - error in a positive effect, **neg rem** - missing negative effect, **neg add** - additional negative effect, **neg rep** - error in a negative effect, and **polarity** - wrong polarity of an effect.

As for missing actions, fluents, or action parameters, we decide to exclude those error types from our quantitative evaluation and will briefly discuss them in a qualitative evaluation.

We start by creating a list of all reasonable errors grouped by type for each domain. Additionally, we determine the set of dynamic fluents for each domain. We call a fluent dynamic if it changes after the initial situation, meaning that there's at least one effect for that fluent.

Error types **pre rem**, **pos rem**, **neg rem**, and **polarity** are easy to generate because it involves iterating over all preconditions and effects and then removing that element or changing its polarity.

Then, we inject errors of the types **pre rep**, **pos rep**, and **neg rep** by iterating over each parameter and checking if it can be replaced by another of the same type. For example, we can replace a precondition `(edge ?from ?to)` in the action `move` with `(edge ?to ?to)`, `(edge ?to ?from)`, or `(edge ?from ?from)` since the parameters `?from` and `?to` share the same type.

For error types with element additions, namely **pre add**, **pos add**, and **neg add**, we generate them by iterating over each action in the domain. For each fluent, we make use of the ExprEnumerator to generate syntactically valid expressions based on the parameters of the action. Additionally, it is reasonable to add an effect for a fluent only if that fluent is dynamic. For example, we exclude errors such as the effect `(not (edge ?from ?to))` since this fluent doesn't change in the original domain.

Having obtained a list of reasonable errors, we generate faulty domains by randomly picking a set of errors of a given type from that list and inserting them into the domain.

We run the following sets of testcases for each IPC domain. We generated 100 domains with 2 errors for each mode specified above. For the mode **single**, we run the debugger for each problem instance. Since we use 10 problem instances per domain, the number of runs is 1000 instead of 100 for that mode. For each error type, we generate 100 domains with one error and test it in the **normal** mode. For all n between 1 and 15, generate 100 domains with n errors and test it in the **normal** mode.

We ensure that the generated faulty domain is valid in each testcase. The testcases are executed on a machine with an Intel i7-4770k CPU and 16GiB memory running on Ubuntu 18.04 and using OpenJDK17. In order to save time, the test cases are run in parallel on 6 threads.

4.3 Results

Recovery rates vary substantially across domains (Table 4), ranging from 52% for tpp to 96% for logistics, despite their similar sizes (Table 3). We observe no clear relationship between domain size and recovery rate. Simpler domains such as blocks and logistics achieve lower false positive rates f_{p_μ} than more complex domains like pipesworld and rovers, likely due to their smaller space of possible fixes. Runtime (t_μ) also varies by domain, with zenotravel being the fastest and tpp the slowest, although most testcases finish within one second. The relatively large t_σ indicates that runtime depends largely on the structure of individual faulty domains.

Table 4 Average recovery rate rec_μ, average false positive rate fp_μ, average runtime t_μ, and its standard deviation t_σ. The results are averaged over testcases with 2 errors, all error types in normal execution mode.

Domain	n	$rec_\mu[\%]$	$fp_\mu[\%]$	$t_\mu[ms]$	$t_\sigma[ms]$
blocks	100	76	12	34	38
freecell	100	73	85	266	56
logistics	100	98	2	30	16
driverlog	100	86	48	32	26
satellite	100	59	59	43	23
zenotravel	100	65	52	14	7.96
pipesworld	100	80	98	148	46
rovers	100	74	90	226	49
storage	100	83	77	18	14
tpp	100	55	81	71	29

Table 5 shows that recovery rates are higher for invalid than for valid faulty domains (80% vs. 60%). We assume errors are easier to identify when they cause a concrete plan failure. Notably, when the TIGHTEN algorithms are disabled and only VIOLATIONFIXER is used, the debugger still achieves a 28% recovery rate on valid domains. Since no precondition violations occur in this setting, these errors are detected solely through the SIAs, while maintaining a low false positive rate of 6%.

Table 5 The results for all domains grouped by execution mode and faulty domain validity. The first row of each execution mode shows all corresponding testcases. The second and third rows shows only testcases where the faulty domain is valid or invalid. The results refer to testcases with 2 mixed errors. The table shows the average recovery rate rec_μ, average false positive rate fp_μ, the average runtime t_μ, and its standard deviation t_σ.

Mode	Valid?	n	$rec_\mu[\%]$	$fp_\mu[\%]$	$t_\mu[ms]$	$t_\sigma[ms]$
normal	$\top, \bot$	1000	75	60	88	93
	$\top$	263	60	71	90	91
	$\bot$	737	80	57	88	94
no tighten	$\top, \bot$	1000	53	9	21	26
	$\top$	263	28	6	15	13
	$\bot$	737	62	11	23	29
no sia	$\top, \bot$	1000	70	63	56	60
	$\top$	263	55	74	60	63
	$\bot$	737	75	59	54	60
with ignored	$\top, \bot$	1000	75	27	89	284
	$\top$	263	59	35	75	154
	$\bot$	737	80	24	94	317
single	$\top, \bot$	10000	64	66	20	54
	$\top$	2630	54	76	19	87
	$\bot$	7370	68	62	20	35

As shown in Table 5, providing all plans instead of only a single plan improves the recovery rate by 11 percentage points (75% vs. 64%). The effect is moderate because most problem instances in the test set already use all actions in the domain. In testcases where only the VIOLATIONFIXER is run and the TIGHTEN algorithms are skipped, the recovery rate is just 53%, but the false positives rate is only 9%. This is due to the fact that the VIOLATIONFIXER only makes changes when a violation is encountered, while the TIGHTEN algorithms constrict the domain as much as possible.

Regarding the usefulness of SIAs in general, rec_μ is 5 percentage points lower when no SIAs are given versus testcases with all SIAs, as shown in Table 5. fp_μ is also only slightly influenced, while the usage of SIAs increases the runtime of the testcases by 57%. As it turns out, blacklisting false positives obtained in the first run doesn't improve the recovery rate in the second round on average, as evident from the results in Table 5.

Table 6 compares recovery rates across error types and distinguishes between valid and invalid faulty domains. Added preconditions cannot be detected in valid domains (0% recovery), but are removed in 88% of cases when they cause violations. Missing preconditions always yield valid domains and are recovered in 84% of cases, while modified preconditions are repaired in 80% overall and 90% when violations occur. Added positive effects are repaired in 87% of cases. Missing positive effects are recovered in 89% of invalid domains but never when the domain remains valid. Likewise, modified positive effects are recovered much more often in invalid than valid domains (94% vs. 41%). Additional negative effects are the hardest to repair, with a recovery rate of 41%. However, recovery differs sharply between valid and

Table 6 Results separated by error type. In each testcase, one error of the corresponding type is introduced into the domain. All testcases are run with normal execution mode. Average recovery rate rec_μ, average false positive rate fp_μ, the average runtime t_μ, and its standard deviation t_σ are shown. n reflects the number of distinct errors and thus faulty domains that could be generated for the corresponding error type.

Error	Valid?	n	$rec_\mu[\%]$	$fp_\mu[\%]$	$t_\mu[ms]$	$t_\sigma[ms]$
pre add	$\top, \bot$	510	78	87	112	86
	$\top$	56	0	100	100	89
	$\bot$	454	88	85	114	86
pre rem	$\top, \bot$	262	84	83	112	88
	$\top$	262	84	83	112	88
	$\bot$	0	0	0	0	0
pre rep	$\top, \bot$	357	80	86	116	84
	$\top$	53	22	96	93	90
	$\bot$	304	90	84	120	82
pos add	$\top, \bot$	297	87	89	192	576
	$\top$	297	87	89	192	576
	$\bot$	0	0	0	0	0
pos rem	$\top, \bot$	120	84	78	101	84
	$\top$	7	0	100	116	92
	$\bot$	113	89	77	100	84
pos rep	$\top, \bot$	55	89	77	137	88
	$\top$	6	41	85	137	84
	$\bot$	49	94	76	137	88
neg add	$\top, \bot$	331	41	89	152	186
	$\top$	186	3	92	142	81
	$\bot$	145	91	86	165	265
neg rem	$\top, \bot$	119	83	80	106	87
	$\top$	119	83	80	106	87
	$\bot$	0	0	0	0	0
neg rep	$\top, \bot$	69	48	82	140	90
	$\top$	62	42	86	148	87
	$\bot$	7	100	49	71	89
polarity	$\top, \bot$	239	70	76	151	645
	$\top$	126	84	76	193	882
	$\bot$	113	55	76	105	89

invalid domains (3% vs. 91%). Missing negative effects are recovered in 83% of cases, while modified negative effects are repaired in 42% of valid domains and all invalid ones. Effect polarity errors show the opposite trend, achieving a higher recovery rate in valid than in invalid domains (84% vs. 55%). Runtime t_μ is highest for additional precondition errors due to the exponential complexity of the minimal hitting set computation. The large t_σ is largely explained by the varying complexities of the evaluated domains.

The most surprising result is the limited impact of the number of errors on recovery rate (Table 7). While domains with a single error achieve a recovery rate of 76%, domains with up to seven errors still average 71%. Only beyond that point does performance decline substantially, reaching 35% for domains with 15 errors. As expected, the average runtime t_μ increases with the number of errors, whereas the false positive rate f_{p_μ} decreases. These results suggest that the debugger can reliably reconstruct missing components as long as the domain's basic structure remains intact, but its effectiveness deteriorates once too many elements are missing.

Table 7 The results for all domains grouped by the number of errors introduced in each testcase. The testcases are run under normal execution mode with mixed error types being picked. The table shows the average recovery rate rec_μ, average false positive rate fp_μ, the average runtime t_μ as well as its standard deviation t_σ. In the first row, n is less than 1000, because only 883 distinct errors can be generated using our strategy.

Error Count	n	$rec_\mu[\%]$	$fp_\mu[\%]$	$t_\mu[ms]$	$t_\sigma[ms]$
1	883	76	71	81	343
2	1000	75	60	88	93
3	1000	75	55	147	717
4	1000	74	52	231	910
5	1000	73	51	307	1059
6	1000	72	47	555	1552
7	1000	71	44	933	2148
8	1000	68	41	1355	2604
9	1000	64	38	1931	3116
10	1000	60	34	2497	3593
11	1000	55	29	3139	3888
12	1000	51	26	3754	4111
13	1000	43	22	4645	4321
14	1000	40	18	5099	4349
15	1000	35	15	5674	4353

5 Limitations

While the quantitative evaluation demonstrated that DrPDDL can reliably repair the considered error types, it also revealed some limitations. One limitation is scalability. Algorithms such as ViolationFixer have exponential runtime and may become computationally expensive for larger domains, even though this was not an issue in our evaluation. Another limitation is that DrPDDL reports only a single repair. Although users can exclude suggested fixes and rerun the debugger, alternative solutions produced by components such as the minimal hitting set algorithm are discarded, and the fixed sequence of algorithms further restricts the set of possible repairs. To illustrate these limitations, we now examine error types that were not considered in the implementation.

Removing the `pickup` action from the delivery robot domain causes a precondition violation for `(holding ?r ?p)` during `drop` evaluation. The debugger removes this precondition, but a subsequent SIA violation and failed goal achievement lead to contradictions, yielding no solution. In contrast, removing the fluent `empty` relaxes the domain. Since no preferred-plan violations occur, the debugger cannot detect the error. Missing fluents generally exhibit this behavior. Initial-state errors covered by SIAs can be detected. For example, adding `(isat r1 lb)` triggers an SIA violation. Errors not covered by SIAs, such as removing a traversed `(edge la lb)`, cannot be identified directly; instead, the debugger removes related preconditions, indicating the affected fluent. Impossible goals, incorrect SIAs, and errors in preferred plans all lead to contradictions that prevent a consistent repair. For instance, requiring a packet to be in two places simultaneously or removing a `(pickup r1 p2 le)` step causes violations that ultimately leave the problem unsolvable.

6 Conclusion and Future Work

The goal of this work was to develop an automated debugger for PDDL domains. Given a faulty domain, a set of preferred plans, and background knowledge in the form of State Integrity Axioms (SIAs), the debugger repairs the domain such that all plans remain valid and all SIAs are satisfied.

We first introduced the necessary planning concepts, formalized planning domains, classified common domain errors, and defined a restricted language for expressing SIAs. Based on this foundation, we designed a repair framework consisting of four algorithms: ViolationFixer, which resolves plan and SIA violations; Tighten1, which removes redundant positive effects; Tighten2, which adds valid negative effects; and Tighten3, which adds maximal positive preconditions.

The implementation was evaluated on thousands of faulty instances generated from 10 IPC domains. Recovery rates were analyzed with respect to domain characteristics, error count, error type, available plans, and SIAs. The debugger repaired 75% of domains containing a single error and still achieved 69% recovery with 10 concurrent errors. Using multiple preferred plans improved recovery by about 10 percentage points, while SIAs provided a smaller but measurable benefit. Positive-effect errors were the easiest to repair, whereas negative-effect errors proved the most challenging.

Overall, the results demonstrate that preferred plans and SIAs can be effectively combined to automatically repair planning domains containing multiple concurrent errors.

We have laid the groundwork for an advanced PDDL domain debugger, leaving several avenues for future extension. Adding parameters expands the search space and requires updating preferred plans with new arguments. Supporting full propositional logic in SIA improves expressiveness but increases validation complexity. Incorporating features like quantifiers, fluents, and durative actions scales complexity beyond lifted STRIPS. Finding violations using only the domain, problem, and SIAs would eliminate the need for preferred plans.

References

1 J. Benton, Amanda Coles, and Andrew Coles. Temporal planning with preferences and time-dependent continuous costs. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, pages 2–10. AAAI Press, 2012.

2 Giuseppe De Giacomo and Toni Mancini. Scaling up reasoning about actions using relational database technology. In *Proceedings of the 19th National Conference on Artifical Intelligence*, pages 245–250. AAAI Press, 2004.

3 Jan Dolejsi. Vs code planning domain description language support, 2022. URL: `https://marketplace.visualstudio.com/items?itemName=jan-dolejsi.pddl`.

4 Thomas Eckstein, Marco De Bortoli, and Gerald Steinbauer-Wagner. Fixing pddl domains using delta debugging. In *34th International Workshop on Principles of Diagnosis*. AAAI Press, 2023.

5 Malik Ghallab, Adele Howe, Craig Knockblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl the planning domain definition language. Technical report, Yale Center for Computational Vision and Control, 1998.

6 Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning. Theory & practice*. Elsevier, 2004.

7 Alba Gragera, Raquel Fuentetaja, Angel Garcia-Olaya, and Fernando Fernandez. Pddl domain repair: Fixing domains with incomplete action effects. In *International Conference on Automated Planning and Scheduling*. AAAI Press, 2023.

8 Alba Gragera, Raquel Fuentetaja, Angel Garcia-Olaya, and Fernando Fernandez. A planning approach to repair domains with incomplete action effects. In *International Conference on Automated Planning and Scheduling*. AAAI Press, 2023.

9 Richard Howey and Derek Long. Val's progress: the automatic validation tool for pddl2.1 used in the international planning competition. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, pages 28–37. AAAI Press, 2003.

10 Songtuan Lin, Alban Grastien, and Pascal Bercher. Planning domain repair as a diagnosis problem. *DX 2022*, 2022.

11 Songtuan Lin, Alban Grastien, and Pascal Bercher. Towards automated modeling assistance: An efficient approach for repairing flawed planning domains. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37:12022–12031, 2023. doi:10.1609/AAAI.V37I10.26418.

12 Patrick Lühne. pddl-instances, 2017. URL: https://github.com/potassco/pddl-instances.

13 Erik Mueller. *Commonsense Reasoning - An Event Calculus Based Approach*. Elsevier, 2014.

14 Christian Muise. planutils, 2020. URL: https://github.com/AI-Planning/planutils.

15 Tomas Plch, Miroslav Chomut, Cyril Brom, and Roman Barták. Inspect, edit and debug pddl documents: Simply and efficiently with pddl studio. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*. AAAI Press, 2012.

16 Raymond Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64(2):337–351, 1993. doi:10.1016/0004-3702(93)90109-O.

17 Kayleigh Sleath and Pascal Bercher. Detecting ai planning modelling mistakes - potential errors and benchmark domains. In *Proceedings of the 20th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2023)*. Springer, 2023. doi:10.1007/978-981-99-7022-3_41.