Incremental Planning over Lifted Abstractions

Michelle Kornherr[1], Daniel Gnad[2,3], Zeynep G. Saribatur[1] and Johannes K. Fichte[3]

[1] Institute of Logic and Computation, TU Wien
[2] Institute of Computer Science, Heidelberg University
[3] Department of Computer and Information Science (IDA), Linköping University

Abstract

Abstraction is a well-known concept in planning that lends itself to numerous approaches for solving planning tasks more efficiently. A prominent use are abstraction heuristics that effectively guide a search algorithm. In contrast to that, we propose to abstract the planning task for finding abstract solutions that are then repaired into concrete plans for the input task. Such abstractions cannot be as extreme as those typically employed in heuristics, but need to follow a scheme that induces abstract plans that can in fact be repaired with reasonable computational effort. Our abstractions are established on the lifted PDDL model by relaxing actions and grouping objects of the same type. This leads to substantial reductions in the state-space size, which simplifies finding (abstract) plans. In a second step, we repair these plans by substituting abstract actions with concrete ones. In our evaluation we showcase domains in which finding an abstract plan and repairing it is significantly faster than solving the original problem.

Introduction

Abstraction is a common approach in planning to simplify complex tasks. By reducing the level of detail in a planning model, abstraction can make reasoning about large state spaces feasible. Traditionally, abstracted tasks are primarily used to compute heuristic functions that guide search algorithms (Culberson and Schaeffer 1998; Edelkamp 2001; Helmert et al. 2014; Klößner et al. 2021; Klößner, Seipp, and Steinmetz 2023; Kreft et al. 2023). In these works, the abstraction is often extreme: many details of the original problem are ignored to accelerate heuristic computation, even if the resulting heuristics are only loosely connected to concrete plans. While effective for search guidance, e.g. when combining multiple abstractions, this limits the possibility to directly leverage the abstract solution for plan generation.

In this paper, we investigate a different use of abstraction: generating abstract plans that can be systematically repaired into valid concrete plans. This approach requires abstractions that are less extreme than those used for heuristics, as the resulting abstract plan must remain "repairable" without incurring excessive computational cost. Our system implements this idea through a multi-stage pipeline that operates on lifted PDDL representations. There are two key components: (1) we identify objects that have the same type and abstract a set of them by keeping a single representative, to then (2) perform a targeted partial relaxation of the delete effects of the action schemas involving these objects. This allows us make up for the reduced object count, for example relaxing the capacity of a vehicle when mapping multiple vehicle objects into one.

Both abstract and concrete PDDL problems are first translated into finite domain representations using Fast Downward (Helmert 2009), and then converted into answer-set programming (ASP) encodings via plasp (Dimopoulos et al. 2019). The abstract ASP program is solved to produce an abstract plan, from which all action occurrences are extracted. To bridge the gap between abstract and concrete actions, we generate a mapping that links each abstract action to compatible concrete actions. The concrete ASP program is then solved under the constraints from this mapping, producing a concrete plan that refines the abstract plan while satisfying the conditions of the original domain. By grouping objects of the same type and relaxing certain preconditions, this approach reduces the effective state space and allows planning to focus on high-level structure first, before filling in details.

Our approach is closely related to the work of Horčík, Fišer, and Torralba (2022), who study homomorphisms of lifted planning tasks to derive delete-relaxation heuristics. While their work uses abstraction for heuristic evaluation, we extend the concept to full plan synthesis. By exploiting lifted task homomorphisms, we construct abstract tasks whose solutions can be concretized efficiently. In this sense, our work complements theirs: we move from heuristic guidance to full plan generation via abstraction and repair.

The identification of similar objects is close to symmetry detection mechanisms (Pochter, Zohar, and Rosenschein 2011; Domshlak, Katz, and Shleyfman 2012; Röger, Sievers, and Katz 2018; Sievers et al. 2019). Our current approach is based on pure type-checking of the involved objects. In the future we plan to extend our abstractions by including the stronger symmetry detection check.

We evaluate our method on selected benchmark domains. Our results demonstrate that computing an abstract plan and repairing it can often be significantly faster than directly solving the original task, while still producing high-quality plans. These findings suggest that abstract-then-repair planning can provide a powerful alternative to conventional heuristic-driven search, particularly in domains where lifted abstractions are natural and the state space is large.

Preliminaries

Lifted Classical Planning We follow established definitions of lifted classical planning (Corrêa et al. 2020), which capture a common fragment of PDDL. A planning task is a tuple $\Pi = \langle \mathcal{P}, O, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{P}$ is a set of *predicate symbols*, O is a set of *objects*, $\mathcal{A}$ is a set of *action schemas*, $\mathcal{I}$ is the *initial state*, and $\mathcal{G}$ is the *goal condition*. Each predicate $P \in \mathcal{P}$ has an arity n, and for an n-ary predicate and a n-tuple $\vec{t} = \langle t_1, \dots, t_n \rangle$ consisting of variables or objects $o \in O$, $P(\vec{t})$ is called an *atom*. An atom $P(\vec{t})$ is *ground* if all of $\vec{t}$'s components are objects $o \in O$. A *state* s is a set of ground atoms, and both $\mathcal{I}$ and $\mathcal{G}$ are states. A state s that satisfies the goal condition, i.e., where $s \supseteq \mathcal{G}$, is a *goal state*.

An *action schema* $a[\Delta] \in \mathcal{A}$ is a triple $\langle \text{pre}(a[\Delta]), \text{add}(a[\Delta]), \text{del}(a[\Delta]) \rangle$, consisting of the *precondition*, *add list*, and *delete list* of $a[\Delta]$, all of which are finite sets of atoms. Here, Δ is the set of free variables that occur in any atom of any component of $a[\Delta]$. We can ground an action schema $a[\Delta]$ by substituting the free variables Δ by objects in O, which results in a *ground* action a, or simply action. An action a is applicable in a state s if $\text{pre}(a) \subseteq s$. When applying a in s, the successor state s' is defined as $a[\![s]\!] = s' := (s \setminus \text{del}(a)) \cup \text{add}(a)$. A sequence of actions $\pi = \langle a_1, \dots, a_n \rangle$ is applicable in state s_0 if each a_i is applicable in the state s_i generated by applying $a_1, \dots, a_{i-1}$ from s_0. The resulting state of this application is $\pi[\![s_0]\!] = s_n$. The solution of a planning problem is a sequence of actions, called a *plan*, applicable to $\mathcal{I}$ that ends in a goal state, $\pi[\![\mathcal{I}]\!] \supseteq \mathcal{G}$.

Answer Set Programming (ASP) We consider propositional programs. Let l, m, n be non-negative integers such that $l \leq m \leq n$, and $a_1, \dots, a_n$ distinct propositional atoms. A *disjunctive rule* r is of the form

$$a_1 \vee \dots \vee a_l \leftarrow a_{l+1}, \dots, a_m, \mathit{not}\ a_{m+1}, \dots, \mathit{not}\ a_n.$$

By $H_r \coloneqq \{a_1, \dots, a_l\}$, $B_r^+ \coloneqq \{a_{l+1}, \dots, a_m\}$, and $B_r^- \coloneqq \{a_{m+1}, \dots, a_n\}$, we denote the *head*, *positive* or *negative body* atoms of r, respectively. We say that r is *normal* if $|H_r| \leq 1$, *positive* if $B_r^- = \emptyset$, and *Horn* if r is normal and positive. A *disjunctive logic program (DLP)* P, or *program* for short, is a set of rules, where $\text{at}(P) \coloneqq \bigcup_{r \in P}(H_r \cup B_r^+ \cup B_r^-)$ denotes its atoms. For a set $\mathcal{U}$ of atoms and a logic program P, we say P is *over* $\mathcal{U}$ if $\text{at}(P) \subseteq \mathcal{U}$. A program P has a certain property if all its rules have the property.

Semantically, P induces a set of answer sets (Gelfond and Lifschitz 1991), which are Herbrand models (sets I of ground atoms) of P justified by the rules, in that I is a minimal model of $fP^I = \{r \in P \mid I \models B(r)\}$ (Faber, Leone, and Pfeifer 2004). The set of answer sets of a program P is denoted as $AS(P)$. A program P is *unsatisfiable* if $AS(P) = \emptyset$.

Common syntactic extensions are *choice rules* of the form $\{\alpha\} \leftarrow B$, which stands for the rules $\alpha \leftarrow B, \mathit{not}\ \alpha'$ and $\alpha' \leftarrow B, \mathit{not}\, \alpha$, where α' is a new atom, and cardinality constraints and conditional literals (Simons, Niemelä, and Soininen 2002); in particular, $i_\ell\{\, a(X) : b(X)\, \}i_u$ is true whenever at least i_ℓ and at most i_u instances of $a(X)$ subject to $b(X)$ are true.

Constructing Lifted Abstractions

Similar to Horčík, Fišer, and Torralba (2022), we are interested in the notion of PDDL homomorphisms as a map between two sets of objects, which are then extended to actions.

Definition 1. *Let* $\Pi = \langle \mathcal{P}, O, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ *and* $\Pi' = \langle \mathcal{P}', O', \mathcal{A}', \mathcal{I}', \mathcal{G}' \rangle$ *be planning tasks. A map* $\sigma : O \to O'$ *is called a* PDDL homomorphism*, denoted by* $\sigma : \Pi \to \Pi'$*, if the following conditions are satisfied:*

(P1) *σ is a homomorphism from* $\langle O, \mathcal{I} \rangle$ *to* $\langle O', \mathcal{I}' \rangle$,
(P2) $\sigma(\mathcal{G}) \supseteq \mathcal{G}$,
(P3) *for each reachable state s in $\mathcal{P}$, each state s' in $\mathcal{P}'$, and each ground action a applicable in s, if* $\sigma : \langle O, s \rangle \to \langle O', s' \rangle$ *and* $t = a[\![s]\!]$*, then $\sigma(a)$ is applicable in s' and* $\sigma : \langle O, t \rangle \to \langle O', t' \rangle$ *for* $t' = \sigma(a)[[s']\!]$.

We shall refer to the action a *non-abstracted* if $\sigma(a) = a$, and *abstracted* otherwise, which we denote by $\mathcal{A}$.

As in Horčík, Fišer, and Torralba (2022), given a PDDL task Π, we can use a PDDL homomorphism σ to construct another smaller task Π' that *over-approximates* the original task Π, in the sense that if π is a plan in Π, then its mapping $\sigma(\pi)$ is a plan in Π'.

To construct homomorphisms we follow a partial delete-relaxation approach similar to Horčík, Fišer, and Torralba (2022), only relaxing the delete effects of abstracted objects.

Definition 2. *Let* $\Pi = \langle \mathcal{P}, O, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ *be a PDDL We define its* delete relaxation w.r.t. σ *as* $\Pi = \langle \mathcal{P}, O, \mathcal{A}^+, \mathcal{I}, \mathcal{G} \rangle$ *where* $\mathcal{A}^+ = \{\langle \text{pre}(a), \text{add}(a), \text{del}(a) \setminus R \rangle \mid \langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle \in \mathcal{A}, R \cap \sigma(O) \neq \emptyset\}$.

Starting from Π^+ and a map $\sigma : O \to O$, we can construct a new PDDL task as $\sigma(\Pi^+) = \langle \mathcal{P}, \sigma(O), \mathcal{A}^+, \sigma(\mathcal{I}), \sigma(\mathcal{G}) \rangle$. We can show that any $\sigma : O \to O'$ is a PDDL homomorphism from Π to $\sigma(\Pi^+)$.

Theorem 3. *Let P be a PDDL task with a set of objects O and $\sigma : O \to O'$. Then σ is a PDDL homomorphism from P to $\sigma(P^+)$.*

Proof sketch. Our homomorphisms are a special case of those introduced by Horčík, Fišer, and Torralba (2022). ☐

Given a planning task Π, we construct an abstract task Π' together with a mapping $\sigma : O \to O'$ that induces a PDDL homomorphism. The mapping σ is defined by partitioning O into equivalence classes based on their type and assigning each class a single representative in O'. This merges objects, considering them interchangeable at the abstract level. The initial state and goal condition are lifted via σ by replacing each object occurrence accordingly, ensuring that conditions (P1) and (P2) are satisfied. We then modify the action schemas in $\mathcal{A}$ by applying a partial delete relaxation with respect to σ: for an action schema $a[\Delta] \in \mathcal{A}$, we construct a corresponding schema in $\mathcal{A}'$ by removing delete effects that refer to atoms whose arguments are affected by the abstraction (i.e., whose objects are mapped non-injectively by σ). Intuitively, this relaxation compensates for the loss of distinguishability between merged objects by eliminating constraints that would otherwise prohibit valid abstract transitions. The resulting abstract task Π' thus over-approximates

the transition system of Π while preserving the correspondence induced by σ, ensuring that concrete transitions are preserved and that abstract plans can be systematically refined into concrete ones.

Repairing Abstract Plans in ASP

Once we have an abstracted planning task, we compute an abstract plan $\hat{\pi}$ in the simplified task, which then needs to be converted into a concrete plan for the original problem. We translate the original task to ASP to repair the abstract plan.

Mapping abstract to concrete actions We guide the search for a concrete plan π by constraining how the actions should be in correspondence with $\hat{\pi}$. The abstract plan is used to restrict solutions, i.e., action *occurrences* in the plan, of the original planning task. For this purpose, we generate a mapping program (map.lp) that links abstract action occurrences to compatible concrete actions. For abstract actions that involve abstracted objects, the mapping introduces choice rules that enforce the selection of exactly one corresponding concrete action.[1] As example, we are using the Airbus Beluga logistics domain[2], where we abstract multiple *hangar* objects, which can store a limited number of *jigs* in them.

$$1\{\texttt{occurs}(a_{hangar1});\texttt{occurs}(a_{hangar2});\ \ldots\}1 \\ \texttt{:- occurs_abstract}(a_{hangarAbs}).$$

This ensures that each abstract action is refined by exactly one concrete action among its possible instantiations. For abstract actions that do not involve abstracted objects, the mapping can be simplified to a direct implication:

```
occurs(A) :- occurs_abstract(A).
```

Here, the abstract and concrete actions coincide, so no refinement is required. Together, these mapping rules restrict the concrete planner to consider only those plans that are consistent with the abstract plan, while still allowing flexibility in the choice of concrete refinements.

More generally, given an abstract plan $\hat{\pi}$, we guide the construction of a corresponding concrete plan π for Π by constraining the search space. We encode the refinement problem in ASP and introduce a mapping program that enforces a correspondence between abstract and concrete action occurrences. Let $\mathcal{A}'$ be the set of abstract ground actions occurring in $\hat{\pi}$, and for each $a \in \mathcal{A}'$, let $C_1, \ldots, C_n$ be the set of concrete actions whose images under σ coincide with a. Then we let the solver chose to replace a with any one of $C_1, \ldots C_n$. These constraints restrict the concrete search to plans that are consistent with $\hat{\pi}$ under σ, while still allowing non-deterministic choices where multiple refinements are possible. The mapping program is combined with facts encoding the abstraction σ and the abstract plan $\hat{\pi}$, yielding a constrained ASP encoding of the original planning task. While this approach effectively prunes the search space, it is inherently brittle: if no concrete plan consistent with $\hat{\pi}$ exists, the resulting encoding becomes unsatisfiable without providing guidance on how to relax or adapt the abstract plan.

Incremental method We address this incompleteness by introducing an incremental technique that checks for a plan by considering *switches* on the actions in $\hat{\pi}$. This can be easily represented in the answer set program, by adding switch atoms in the body of the rules that can be made true in order for the constraining rule to fire. Thus we have $0\{\texttt{switch}(T)\}1$ for each time step T in the plan $\hat{\pi}$ and the following altered rules

$$1\ \{\ \texttt{occurs}(C_1);\ \ldots;\ \texttt{occurs}(C_n)\ \}\ 1\ \texttt{:-} \\ \texttt{occurs_abstract(a,T), switch(T).}$$

This way we can navigate the search for an original plan by turning the switches for the actions "on/off". Finally, for the translation program to work properly, we need to slightly modify the encoding of the original task to allow adding actions to some plan step only if the corresponding switch if off:

```
0 {occurs(a, T) : action(a)} 1 :-
      time(T), T > 0, not switch(T).
```

so that an original action is guessed for those time steps where the switch is not on.

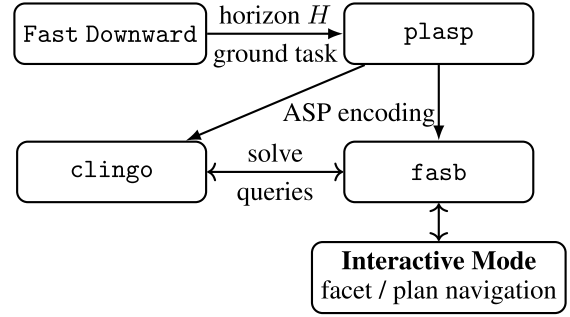

Figure 1: Illustration of PlanPilot's components.

Software Architecture

Our architecture is based on PlanPilot, which allows for interactive navigation of the solution space of a planning task (Gnad et al. 2025). Figure 1 illustrates the interaction between the components of PlanPilot. First, it uses Fast Downward (Helmert 2006) to ground the given planning task. We pass this, together with the *horizon* H, a bound on the plan length to plasp to encode the planning task in ASP. There are options that restrict the plan length either *exactly* to H or let's H be the upper *bound*. The ASP encoding is passed to fasb (and its internal solver clingo), which enables interactive reasoning with facets or plans.

Our system's pipeline begins by processing two PDDL planning tasks independently: the abstract domain/problem (Π') and the concrete domain/problem (Π). Both are passed to the Fast Downward translator to compute a finite-domain SAS representation. These SAS files are subsequently converted into ASP programs using plasp, resulting in an abstract ASP encoding and a concrete counterpart.

The system then enters a multi-stage solving cycle. First, an abstract plan is computed from the abstract encoding. A

[1]Note that here we assume that there are no multiple actions occurring in the same time step and using the abstracted objects. This assumption however can be lifted.

[2]https://tuples.ai/competition-challenge/

extraction module parses the resulting answer set to produce the action occurrences as facts which will be used for repairing the abstract plan on the concrete program. Simultaneously, a dedicated module constructs the mapping file, by identifying which actions require refinement based on the abstraction and generates the switch-guarded rules discussed previously. Our architecture supports two distinct operational modes for repairing the abstract plans. In *incremental mode (inc)* the system activates switches sequentially according to the abstract plan's execution order. At each time step, a concrete plan that matches the actions from the abstract plan that are "on" is queried. If a switch activation leads to unsatisfiability, the specific abstract action is identified as the point of failure. Then a new abstract plan is computed, by forbidding the previously computed sequence of abstract actions until the failed time step. In *decremental mode (dec)* the system attempts to solve the restricted concrete task with all switches initially active. If that is unsatisfiable, switches are disabled in reverse order until a satisfiable sub-plan is found which helps identifying the earliest failing action. Then again a new abstract is queried by forbidding the failed sequence.

Empirical Evaluation

We evaluate our approach on the IPC nomystery domain and the Airbus Beluga logistics problems. All experiments were conducted with a 1200s (20-minute) timeout.

Nomystery with fuel abstraction We evaluate our approach on the optimal-track nomystery domain consisting of 20 instances (p01–p20), where the abstraction is done over the fuel consumption. These instances vary in difficulty, with increasing complexity in terms of state space size and planning horizon. For a subset of instances (p07–p10 and p16–p20), the concrete formulation exceeded the time limit.

Figure 2 shows the runtime comparison between concrete and abstract formulations with the incremental and decremental mode for abstract plan repairs. For clarity, timeout instances are not displayed in the plot, but are accounted for using a 20-minute cutoff. The concrete formulation exhibits a largely monotonic increase in runtime with problem difficulty, while the abstract formulation shows a more irregular pattern with noticeable spikes on certain instances such as p02 and p04. While the incremental mode follows the increase in runtime similar to the concrete planning, the decremental mode shows advantage in finding a plan via abstract planning and repair process.

Beluga with hangar abstraction Table 1 summarizes runtime behavior across representative instances from the Beluga benchmark suite under varying configurations by also increasing the number of hangars to observe the effect of abstraction. Benchmarks are taken from the explainability track of the Beluga Challenge[3]. On smaller instances, such as Problems 3 and 14, all variants solve efficiently with negligible differences, indicating that the abstraction overhead is minimal in these regimes. In contrast, larger instances (e.g., Problems 1, 12, and 26) show high divergence: the

[3]https://tuples.ai/competition-challenge/

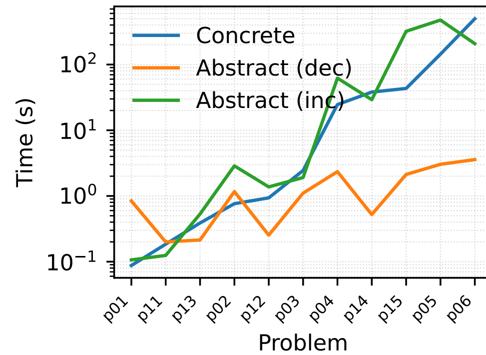

Figure 2: Runtime comparison between concrete and abstract formulations on the Nomystery benchmark.

Prob.	H	Hangars	Solving Time (s) Concrete	Abstract (inc)	Abstract (dec)
1	30	3	410.56	806.23	TO
3	17	3	0.20	0.39	0.31
3	17	7	0.24	0.42	0.35
12	26	3	TO	TO	TO
14	15	3	8.48	7.40	6.53
14	15	5	7.45	9.23	12.70
26	28	2	485.84	TO	TO
26	28	4	TO	TO	TO
39	20	5	6.72	7.40	6.82

Table 1: Runtime comparison between concrete and abstract formulations across Beluga instances with varying number of hangars.

concrete formulation remains comparatively stable in some cases, while abstract variants frequently degrade or exceed the timeout threshold, particularly as the number of hangars increases. We plan to further investigate this behavior, as we expected to observe similar reductions as in nomystery. One hypothesis is that we find many abstract plans that cannot be repaired easily. Across all formulations, increasing hangar counts consistently increases runtime, reflecting higher combinatorial branching. Incremental and decremental modes exhibit no consistent advantage, instead showing instance-dependent variability without systematic gains.

Conclusion

We presented a framework for planning via lifted PDDL abstractions that generates repairable abstract plans rather than just heuristic guidance. By grouping objects of the same type and applying partial delete relaxation, our system reduces state-space complexity while maintaining a path to concrete refinement. The integration to an ASP-based reasoning enables efficient incremental and decremental solving strategies. Our results indicate that this pipeline has the potential to significantly outperform direct solving in structured domains. Future work will focus on enhancing the abstraction process by incorporating symmetry detection to identify candidate objects more precisely, and automating the creation of PDDL homomorphisms.

References

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation using Query Optimization Techniques. In Beck, J. C.; Karpas, E.; and Sohrabi, S., eds., *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*, 80–89. AAAI Press.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence*, 14(3): 318–334.

Dimopoulos, Y.; Gebser, M.; Lühne, P.; Romero, J.; and Schaub, T. 2019. plasp 3: Towards Effective ASP Planning. *Theory and Practice of Logic Programming*, 19(3): 477–504.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2012. Enhanced Symmetry Breaking in Cost-Optimal Planning as Forward Search. In McCluskey, L.; Williams, B. C.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*. AAAI.

Edelkamp, S. 2001. Planning with Pattern Databases. In Cesta, A.; and Borrajo, D., eds., *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, 84–90. AAAI Press.

Faber, W.; Leone, N.; and Pfeifer, G. 2004. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, 200–212. Springer.

Gelfond, M.; and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9(3): 365–385.

Gnad, D.; Hecher, M.; Gaggl, S.; Rusovac, D.; Speck, D.; and Fichte, J. K. 2025. Interactive Exploration of Plan Spaces. In Ortiz, M.; Wassermann, R.; and Schaub, T., eds., *Proceedings of the Twenty-Second International Conference on Principles of Knowledge Representation and Reasoning (KR 2025)*. IJCAI Organization.

Helmert, M. 2006. The Fast Downward Planning System. 26: 191–246.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence*, 173: 503–535.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM*, 61(3): 16:1–63.

Horčík, R.; Fišer, D.; and Torralba, Á. 2022. Homomorphisms of Lifted Planning Tasks: The Case for Delete-free Relaxation Heuristics. In Honavar, V.; and Spaan, M., eds., *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022)*, 9767–9775. AAAI Press.

Klößner, T.; Seipp, J.; and Steinmetz, M. 2023. Cartesian Abstractions and Saturated Cost Partitioning in Probabilistic Planning. In Gal, K.; Nowé, A.; Nalepa, G. J.; Fairstein, R.; and Rădulescu, R., eds., *Proceedings of the 26th European Conference on Artificial Intelligence (ECAI 2023)*, 1272–1279. IOS Press.

Klößner, T.; Torralba, Á.; Steinmetz, M.; and Hoffmann, J. 2021. Pattern Databases for Goal-Probability Maximization in Probabilistic Planning. In *ICAPS 2021 Workshop on Knowledge Engineering for Planning and Scheduling*, 80–89.

Kreft, R.; Büchner, C.; Sievers, S.; and Helmert, M. 2023. Computing Domain Abstractions for Optimal Classical Planning with Counterexample-Guided Abstraction Refinement. In Koenig, S.; Stern, R.; and Vallati, M., eds., *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS 2023)*, 221–226. AAAI Press.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2011. Exploiting Problem Symmetries in State-Based Planners. In Burgard, W.; and Roth, D., eds., *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, 1004–1009. AAAI Press.

Röger, G.; Sievers, S.; and Katz, M. 2018. Symmetry-Based Task Reduction for Relaxed Reachability Analysis. In de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M. T. J., eds., *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, 208–217. AAAI Press.

Sievers, S.; Röger, G.; Wehrle, M.; and Katz, M. 2019. Theoretical Foundations for Structural Symmetries of Lifted PDDL Tasks. In Benton, J.; Lipovetzky, N.; Onaindia, E.; Smith, D. E.; and Srivastava, S., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2019, Berkeley, CA, USA, July 11-15, 2019*, 446–454. AAAI Press.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2): 181–234.