Toward a General Framework for Evaluating Per-Domain Generalization Using LLMs, Theorem Provers, and Statistical Model Checking

Nicola J. Müller[1,2,3], **Naya Rudolph**[1,2], **Ayal Taitler**[4], **Timo P. Gros**[1,2,3]
[1]German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
[2]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[3]Center for European Research in Trusted Artificial Intelligence (CERTAIN)
[4]Ben-Gurion University of the Negev
{Nicola.Mueller, Naya.Rudolph, Timo_Phillip.Gros}@dfki.de, ataitler@bgu.ac.il

Abstract

Generalized planning aims to synthesize generalized plans that can solve any instance of a fixed domain. Evaluating these methods is difficult because planning domains usually describe an infinite family of instances. Recent work proposed evaluating generalized planning methods on randomly generated instances of increasing size until statistical guarantees on the performance can be provided. However, this approach does not support generating instances from specific distributions, meaning it can only evaluate the average generalization across the domain, potentially hiding failures on rare but critical instances. In this work, we propose an efficient and flexible approach to targeted instance generation, which is based on the idea that we can modify existing instance generators using large language models (LLMs). Concretely, we specify instance distributions using first-order logic constraints and prompt an LLM to modify an existing instance generator such that it only returns instances fulfilling the given constraints. We then use a theorem prover to verify that the instances returned by the resulting generator fulfill the given constraints. Any incorrectly generated instances are used in an iterative feedback loop with the LLM. We then plug the instance generators into an evaluation framework that specifies a generic agent interface and is compatible with off-the-shelf statistical model checking tools. The result is a first step toward a general evaluation framework for per-domain generalization, which can evaluate any generalized planning approach according to any property on any distribution of instances from a given domain.

1 Introduction

Generalized planning is a well-established setting in the classical planning community. Given a fixed domain and a set of training instances, the goal is to synthesize a generalized plan that can solve any instance from the domain. There is a wide variety of generalized planning methods, ranging from feature-based abstractions for computing general policies (Bonet, Frances, and Geffner 2019), to neural approaches that learn policies with graph neural networks (Ståhlberg, Bonet, and Geffner 2022a,b, 2025), and approaches that synthesize Python programs using large language models (Silver et al. 2024; Stein et al. 2026).

Evaluating the generalization capabilities of these methods is uniquely difficult because planning domains usually describe an infinite family of possible problem instances. Although some generalized planning approaches can provide strong formal guarantees on their generalization capabilities (Abdulaziz, Pommerening, and Corrêa 2022), such guarantees are not available in general, especially for approaches based on machine learning. Hence, we require methods that automatically evaluate generalized planning methods on representative test instances according to relevant properties. A first step toward this goal is Gros et al.'s scaling behavior evaluation for per-domain generalizing policies in the classical planning setting (2025b). For a given instance size, scaling behavior evaluation runs a policy on uniformly generated instances until its estimate of the policy's performance is within a predefined confidence interval. This process repeats for increasing instance sizes until the evaluation is stopped. While scaling behavior evaluation shows the average performance across the full distribution of instances induced by the given generator, it does not evaluate the performance on specific instance distributions. For example, a policy trained on the Transport domain might perform well on average, but it could fail on instances with specific road patterns, such as particularly long roads. Finding or showing the absence of such rare but critical failures is essential to understanding the generalization capabilities of generalized planning methods. This is particularly true for machine-learning-based approaches, which often lack interpretability. We aim to solve this issue by introducing a method that allows us to restrict instance distributions and efficiently generate corresponding instances.

Automating instance generation and selection is an established topic in the planning community. For example, Torralba, Seipp, and Sievers treat the problem of selecting a set of suitably difficult instances as an optimization problem, where state-of-the-art planners are used to assess the difficulty of instances (2021). Further, Núñez-Molina, Mesejo, and Fernández-Olivares formulate the problem of generating diverse and difficult instances as a Markov Decision Process and use deep reinforcement learning to train policies that generate initial states and goals (2024; 2025). Lastly, Grundke, Helmert, and Röger use answer set programming to generate instances from formal specifications of planning domains (2025).

We observe that instance generators are already available for most of the popular classical planning domains, typically in the form of simple C or Python code, and thus it should be

possible to modify them to generate only certain instances. In this work, we propose a pipeline that automatically modifies given generators using a feedback loop between a large language model (LLM) and a theorem prover. Concretely, we specify an instance distribution using constraints in first-order logic (FOL) and then prompt an LLM to adapt the existing instance generator so that it only returns instances fulfilling the given constraints. For example, we specify the distribution of Blocksworld instances where, in the initial state, all blocks are on the table and, in the goal, all blocks are in a single tower, as follows:

```
(and (forall (?b - object) (on-table_I ?b))
     (exists (?bg - object)
        (and (clear_G ?bg)
             (forall (?bgp - object)
                (implies (clear_G ?bgp)
                         (= ?bgp ?bg)))))).
```

We then test the resulting instance generator by generating a large number of instances across multiple sizes and translating them such that we can use the Z3 theorem prover (De Moura and Bjørner 2008) to check for any constraint violations. Additionally, we test for runtime errors and obvious mistakes in the generated instances, such as unreachable goals or incorrect instance sizes. The results of these tests are then returned to the LLM as part of a debug prompt, closing the feedback loop. After a predefined number of iterations, we return the generator that generated the largest number of correct instances. During evaluation, we apply these tests to every generated instance and return only those that pass all of them, which guarantees their correctness.

In our evaluation framework, we wrap the LLM-generated instance generators in Gymnasium (Towers et al. 2024) environments and specify a generic agent interface for environment interactions. This way, we can treat any generalized planning approach as a simple reactive agent and evaluate it using off-the-shelf statistical model checking tools, such as PyDSMC (Gros et al. 2025a). The result is a first step toward a general evaluation framework for per-domain generalization, which can evaluate any generalized planning approach according to any property on any distribution of instances from a given domain.

In our preliminary experiments, we demonstrate that our pipeline can efficiently generate constrained generators and that our evaluation framework can use these generators to evaluate various generalized planning approaches.

2 A Pipeline for Modifying Generators

In this section, we introduce our pipeline for modifying instance generators using LLMs and theorem provers. We first specify which types of bugs we expect from LLM-generated instance generators and how we test for them. Then, we show how we can check constraints on PDDL instances using theorem provers. Lastly, we introduce our prompting strategy for attaining and iteratively improving modified generators.

2.1 Testing Generators

Our test suite assumes that every given instance generator is implemented in Python and follows a simple API: The generator consists of a single class with a predefined name, and a method `generateInstanceForSize`, which inputs an instance size and an optional random seed, and returns either a string, corresponding to an instance file, or `None`. Consider Table 1, which shows an overview of the bugs that we check for, how we test them, and what error messages we provide. We group the bugs into four categories: code bugs, which correspond to the generator throwing runtime errors or running too inefficiently, instance file bugs, which correspond to a returned instance file being fundamentally incorrect, quality bugs, which correspond to a returned instance being trivial or impossible to solve, and constraint bugs, which correspond to a returned instance violating the user-defined constraints.

Given an instance generator, our testing pipeline attempts to generate M instances for each instance size n from a given range. For each generation attempt, we run the tests shown in Table 1 in order from top to bottom, skipping any remaining tests if a code or instance file bug occurs. If a test fails, we record the bug type, the inputs to the `generateInstanceForSize` method, the output (if one was generated), and the error message. Generated instance files that pass all tests are stored for later use.

2.2 Checking Constraints

We use constraints in first-order logic (FOL) to specify the distribution of instances that an LLM-generated instance generator should return. To test whether any constraint violations, i.e., constraint bugs, occur, we translate each generated PDDL instance and the FOL constraints into a finite Z3 model, using Microsoft's SMT solver Z3 (De Moura and Bjørner 2008).

We encode each PDDL object type as a custom `Enum` sort and each predicate as an uninterpreted Boolean function. To distinguish facts about the initial state or the goal of the instance, we create for each predicate two versions P_I and P_G, respectively. Then, for every predicate, we enumerate all ground tuples matching its signature and add axioms that assign truth values according to the facts specified in the instance. We then add the user constraint to the model and let Z3 decide satisfiability. If the model is unsatisfiable, the generated instance violates the constraint.

A complication arises because PDDL treats initial states and goals differently. Initial states are given as lists of facts under a closed-world assumption, i.e., facts not listed are false. Goals, however, are formulas that describe what must hold in any goal state. Consequently, facts not mentioned in the goal are not false but unconstrained. This open-world interpretation can be problematic when we specify constraints over the goals. For example, the constraint "At least one package must be inside a truck in the goal state" is satisfied under the open-world assumption even if the goal does not specify any package that should be inside a truck. To avoid this ambiguity, we introduce a third predicate version P_{GS}, where GS stands for "goal specified". For a fact

Category	Bug	Test	Error Message
Code bugs	Class loading bug	Try loading the generator class.	Python error message
	Instance Generation bug	Check whether instance generation terminates without a runtime error.	Python error message
	Timeout bug	Check whether instance generation exceeds a time limit.	"Generation exceeded timeout of N seconds"
Instance file bugs	Parsing bug	Check whether the instance file can be parsed.	Error message of PDDL parser
	Instance size bug	Check whether the number of objects in the instance matches the requested size.	"Expected N objects, but got M instead"
Quality bugs	Goal tautology bug	Check whether the goal is always true.	"The goal is a tautology"
	Goal fulfilled bug	Check whether the initial state already satisfies the goal.	"The initial state already fulfills the goal"
	Goal contradiction bug	Check whether the goal is contradictory.	"The goal is a contradiction"
	Unsolvability bug	Check whether the initial state's h^{FF} value is infinite.[1]	"The initial state has the heuristic value $h^{\text{FF}}(s) = \infty$."
Constraint bugs	Constraint bug	Check whether the facts in the initial state and goal fulfill the constraints using Z3.	Unsatisfiability core provided by Z3 [2]

Table 1: Overview of bug categories, bug types, tests, and corresponding error messages for LLM-generated instance generators. [1]An instance with a finite h^{FF} value may still be unsolvable. It is in general not possible to efficiently determine solvability for all instances of a domain. [2] Here, an unsatisfiability core is the not necessarily minimal subset of state predicates and constraint axioms that are used in the unsatisfiability proof.

$P(x_1, \ldots, x_n)$, the predicate $P_{GS}(x_1, \ldots, x_n)$ is true iff that fact, or its negation, explicitly occurs in the goal. This way we can specify constraints that distinguish between facts that are required by the goal, forbidden by the goal, or left unspecified.

Our pipeline inputs constraints in a PDDL-like syntax supporting standard FOL operators, including conjunction, disjunction, negation, implication, XOR, universal and existential quantification, and equality. Variables are prefixed with "?" and may be typed. We disallow free variables, since our use case only requires satisfiability checks over the finite object universe of a generated instance. Although Z3 is more expressive than strictly necessary, our constraints range over a finite universe with finitely many non-recursive predicates and no function symbols beyond uninterpreted Boolean functions. Thus, all checks are decidable and terminate. The checker is exposed through a simple interface: given a parsed PDDL instance and a constraint string, it returns whether the constraint is satisfiable.

When a constraint bug occurs, we store Z3's unsatisfiability core as an error message to provide feedback to the LLM. This core is a subset of the axioms used in the unsatisfiability proof and gives the LLM concrete feedback about which facts cannot be satisfied together.

2.3 Prompting

In our initial prompt, we provide the LLM with the domain file, the original generator (which can be implemented in any programming language), an example instance file generated using the original generator, and the FOL constraints that instances must fulfill. Given these inputs, the initial prompt asks the LLM to generate a single, self-contained Python script that modifies the existing generator as little as possible while fulfilling the API defined in Section 2.1. Further, we specify the bugs shown in Table 1 and state that none of them should occur. Additionally, we ask that the generator returns diverse instances, i.e., it covers the full distribution of instances that fulfill the given constraints. The prompt is then given to the LLM, and the resulting instance generator is passed to the test suite defined in Section 2.1.

If any bugs are found during testing, we begin the debugging process. To do so, we create a concise debugging prompt that states the number of test cases, how many of them showed bugs, how many correct instances were generated, and how often the generator returned `None`. Further, we provide a concise set of failed test cases, where instead of listing every single case, we select the smallest subset that covers all unique bug types that occurred during testing. Additionally, we provide positive feedback in the form of a randomly sampled set of successful test cases. Lastly, we instruct the LLM to fix the bugs in its generator while keeping the code simple. The debug prompt is then given to the LLM, and the updated instance generator is again passed to the testing pipeline. This debugging process repeats up to M times and stops early if a generator passes all test cases. Finally, the generator with the fewest bugs is returned.

3 Toward a General Evaluation Framework

This section describes our evaluation framework for per-domain generalization in classical planning. We build upon Gros et al.'s scaling behavior evaluation, which uses statistical model checking to evaluate a given policy's coverage or plan length on automatically generated instances of increasing size (2025b). Specifically, for a given instance size n, the scaling behavior evaluation runs a policy on uniformly generated instances until the estimate of the mean coverage $\hat{C}_n$ or the mean plan length $\hat{L}_n$ for size n is within a confidence interval of width $2 \cdot \epsilon$ with a probability of $1 - \kappa$. This process repeats for increasing instance sizes n until either a maximum size is reached or the mean coverage falls below a threshold.

We retain the central ideas of scaling behavior evaluation but generalize them along three axes:

(i) We move from only evaluating instances from the distribution induced by the generator to allowing the specification of arbitrary distributions.

(ii) We move from evaluating coverage and plan length to supporting the evaluation of arbitrary trajectory-based properties.

(iii) We move from evaluating GNN-based policies to supporting the evaluation of arbitrary generalized planning methods using a generic agent interface.

We note that, while (i) is achieved through our LLM-generated instance generators introduced in Section 2, (ii) and (iii) are theoretically supported by scaling behavior evaluation but have never been practically realized.

To support the evaluation of arbitrary trajectory-based properties (ii), we make our framework compatible with the PyDSMC tool (Gros et al. 2025a), which was developed for statistical model checking of neural agents implementing the Gymnasium interface. Given an agent, an environment, a list of properties, and statistical parameters, PyDSMC automatically generates trajectories and computes confidence bounds using the correct statistical methods. We note that PyDSMC does not actually require agents to be neural, but only requires that agents and environments implement the Gymnasium interface. Accordingly, we wrap our instance generators inside Gymnasium environments. A Gymnasium environment must implement a `reset` method, which resets the environment to an initial state, and a `step` method, which executes state transitions. For a given instance size, the `reset` method of our environments generates a new instance file and then parses it using the Mimir library (Ståhlberg and contributors 2026), which serves as the planning backend. The method then returns the initial state, applicable actions, and additionally, the domain and instance file. For a given action, the `step` method computes the state transition using Mimir and returns the successor state, applicable actions, the cost of the executed action, and flags indicating whether the episode was terminated due to reaching a goal or a dead end, or whether the episode was truncated due to exceeding a step limit.

We note that the PDDLGym library (Silver and Chitnis 2020) offers a similar interface. However, PDDLGym differs from our approach in several key aspects: Our environments are built around instance generators, whereas PDDLGym is designed to use a fixed set of instance files. PDDLGym implements its own planning backend for parsing, inference, and state transitions, whereas our environments are designed to use any off-the-shelf planning backend, such as Mimir. PDDLGym also follows the outdated OpenAI Gym API (Brockman et al. 2016), whereas we use the latest Farama Foundation Gymnasium API (Towers et al. 2024). Finally, PDDLGym makes an explicit distinction between agent actions and PDDL operators and does not support action costs, whereas our environments use ground actions directly and support action costs.

To support the evaluation of arbitrary generalized planning methods (iii), we introduce a generic agent interface. The agent interface requires the specification of an `initialize` method and an `act` method. The `initialize` method initializes the underlying generalized planning method. For example, for a GNN policy, `initialize` would load the network parameters from a given checkpoint file, whereas for an LLM-generated generalized plan, `initialize` would execute the generalized plan on a given instance and store the resulting plan. The `act` method queries the agent for an action. For example, for a GNN policy, `act` would pass the current state to the GNN and return the predicted action, whereas for an LLM-generated generalized plan, `act` would return the corresponding action from the plan that was computed in the call to `initialize`, or return `None` if no plan was found, which terminates the current episode.

4 Preliminary Experiments

In our preliminary experiments, we show that our pipeline for modifying instance generators can efficiently compute generators that return instances fulfilling given FOL constraints. We consider the three domains Transport, Blocksworld, and Satellite from the learning track of the International Planning Competition (IPC) 2023 (Taitler et al. 2024). Our pipeline for modifying instance generators uses OpenAI's GPT-5.4 nano model with reasoning disabled. The pipeline runs for 10 iterations or until no more bugs are found. In each iteration, we test the LLM's generator by generating 50 instances for every instance size between 10 and 60, totaling 2500 test instances. For every generation attempt, the generator may take at most 60 seconds before we consider the attempt a timeout bug.

Further, we demonstrate that our generators can be used in our evaluation framework to compare different generalized planning approaches. We consider two generalized planning approaches: The approach by Müller et al. learns robust and efficient per-domain generalizing policies by training graph neural networks to imitate optimal planners (2026). We refer to this approach as "GNN". The approach by Stein et al. uses LLMs to compute generalized plans in the form of Python programs (2026). We refer to this approach as "GenPlan". In our experiments, we give the Python programs a runtime limit of 45 seconds per instance. As a baseline, we consider the satisficing planner LAMA-first (Richter and Westphal 2010) with time and memory limits of 60 seconds and 8GB,

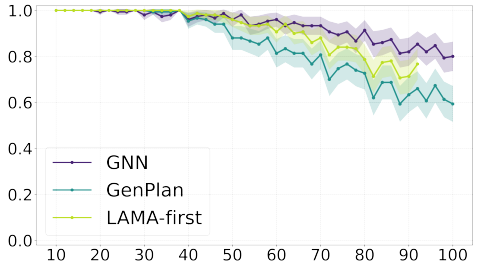
respectively. We evaluate each approach on instances ranging from sizes 10 to 100, where we increase the size in steps of 2. For each size, we use PyDSMC to sample 150 trajectories with a step limit of 120 and compute confidence bounds for both mean coverage and plan length. The evaluation terminates early if the mean coverage drops below 30% for two consecutive instance sizes.

4.1 Transport

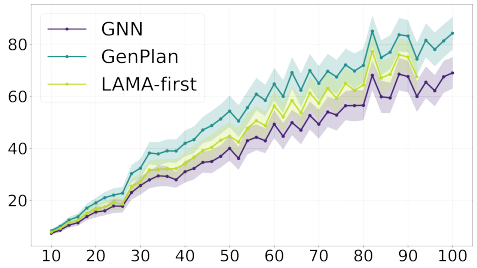
A Transport instance consists of trucks, packages, locations, and size objects used to represent the capacity of trucks. The IPC generator constructs initial states by assigning trucks and packages to random locations that are all connected via randomly drawn roads. The goals only specify the final locations of packages, which must be different from their initial locations but are otherwise sampled randomly. We consider the distribution of Transport instances where only a single package must be delivered, and the goal location is at most two roads away from the initial location. There must also be a truck at the initial location of this package. Hence, any further trucks, packages, and locations are irrelevant to any optimal plan. Formally, we define the constraint as:

```
(exists (?lg - location ?pg - package
         ?li - location ?lm - location
         ?vi - vehicle)
    (and
        (at_I ?pg ?li)
        (at_I ?vi ?li)
        (at_G ?pg ?lg)
        (or
            (and
                (not (= ?li ?lm))
                (not (= ?lm ?lg))
                (not (= ?lg ?li))
                (road_I ?li ?lm)
                (road_I ?lm ?lg))
            (and
                (not (= ?lg ?li))
                (road_I ?li ?lg)))
        (forall (?lgp - location
                 ?pgp - package)
                (implies
                    (at_G ?pgp ?lgp)
                    (and (= ?pg ?pgp)
                         (= ?lg ?lgp)))))).
```

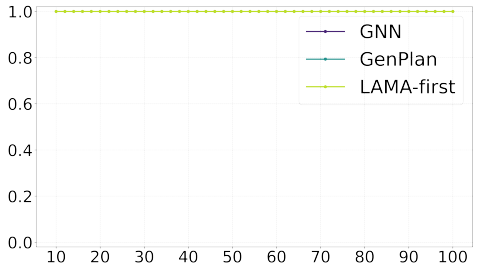
Our pipeline produced a corresponding instance generator that showed no bugs after 1 iteration, which required 12 minutes. In Figure 1, we see that, on the original generator, all approaches scale well, with GNN achieving the highest coverages and shortest plan lengths. On the constrained generator, all approaches achieve 100% coverage, meaning they do not get distracted by objects that are irrelevant to the goal.



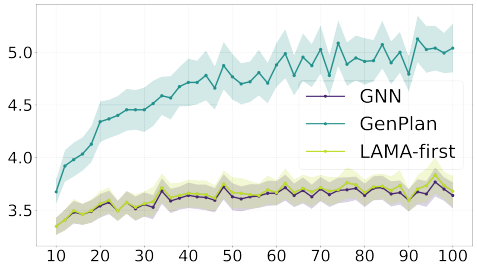
(a) Coverage



(b) Plan length



(c) Coverage on constrained instances



(d) Plan length on constrained instances

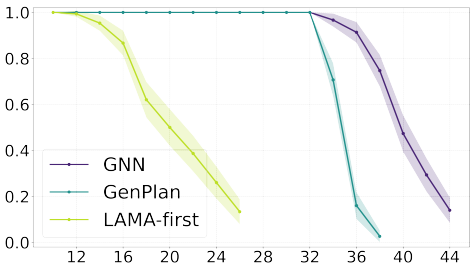
Figure 1: Evaluation on Transport

4.2 Blocksworld

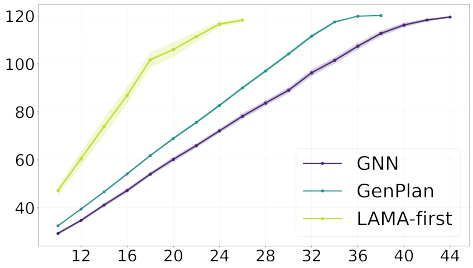
A Blocksworld instance consists of several blocks that must be moved from one configuration to another, using an arm that can pick up one block at a time. The IPC generator assembles blocks in random configurations in both the initial state and the goal, with a bias toward stacking blocks in towers. We consider the distribution of Blocksworld instances where, in the initial state, all blocks are on the table, and in the goal, all blocks are stacked in a single tower. Formally, we define the constraint as:

```
(and (forall (?b - object) (on-table_I ?b))
     (exists (?bg - object)
        (and (clear_G ?bg)
             (forall (?bgp - object)
                (implies (clear_G ?bgp)
                         (= ?bgp ?bg)))))).
```

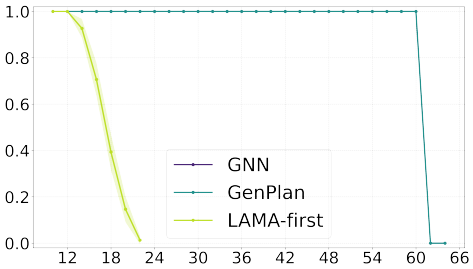
Our pipeline produced a corresponding instance generator that showed no bugs after 3 iterations, which required 21 minutes. Consider Figure 2, which shows the evaluation results for the original instance generator and for the constrained generator. We see that on the original generator,



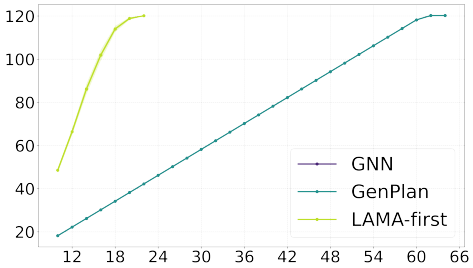
(a) Coverage



(b) Plan length



(c) Coverage on constrained instances



(d) Plan length on constrained instances

Figure 2: Evaluation on Blocksworld

GNN and GenPlan scale significantly better than LAMA-first. On the constrained generator, this difference becomes even larger, as GNN and GenPlan always find optimal plans, whereas LAMA's plan lengths quickly reach the step limit.

4.3 Satellite

A Satellite instance contains satellites, instruments, modes and directions. Satellites may point in any of these directions and have instruments on board. Each instrument supports a certain number of modes. Instruments have to be calibrated by pointing the satellite in the correct direction and are then used to take images using a mode. Satellites can allocate their power to one instrument at a time, which is needed to calibrate and take images. The IPC generator constructs initial states by assigning each satellite an arbitrary start direction as well as at least one instrument. The remaining instruments are distributed arbitrarily and receive a random calibration direction. All satellites have power available in the initial state. Lastly, the generator randomly assigns modes to instruments such that each instrument has at least one mode and each mode is supported by at least one instrument. Two instruments may support the same mode. For the goal, the generator defines both random directions for a random number of satellites as well as a random number of images that must be taken while pointing in specific directions, using certain modes. We consider the distribution of Satellite instances where, in the initial state, at least one satellite already has its power allocated to an instrument. Formally, we define the constraint as:

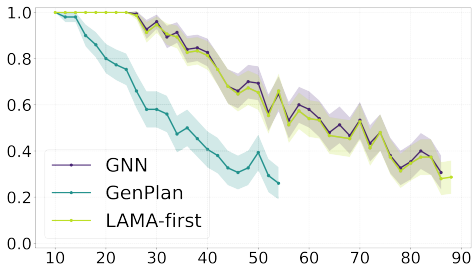
```
(exists (?s - satellite ?i - instrument)
    (and (on_board_I ?i ?s)
         (not (power_avail_I ?s))
         (power_on_I ?i) )).
```

Our pipeline produced a corresponding instance generator that showed no bugs after 3 iterations, which required 6 minutes. In Figure 3, we see that, on the original generator, GNN and LAMA-first scale well, whereas GenPlan struggles on instances larger than size 50. On the constrained generator, we see that the resulting instances are significantly easier to solve, as GNN and LAMA-first achieve nearly constant 100% coverage, yet GenPlan only reaches between 40% and 80% coverage. This is because GenPlan's program can fail if an instrument of any satellite is already powered on in the initial state. The constrained generator yields instances that are easier to solve because, for every direction in which an image must be taken with a certain mode, the generator adds a satellite that must be calibrated in that direction while also having an instrument that supports the required mode. Hence, the LLM restricted the instance distribution more than was necessary. This could be avoided by providing more explicit constraints or by selecting generators that not only have few bugs but also yield diverse instances.

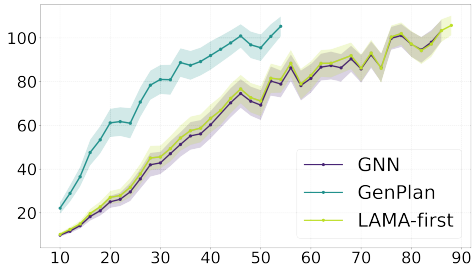
5 Conclusion and Future Work

We present a first step toward a general evaluation framework for per-domain generalization in classical planning. The central idea is to evaluate generalized planning methods not only on the distribution induced by an instance generator but also on specific instance distributions that expose more strengths and weaknesses. To this end, we introduced an efficient and flexible LLM-based pipeline for modifying existing instance generators and validating their outputs using theorem provers. By wrapping the resulting generators in Gymnasium environments and using a generic agent interface, the framework can be combined with off-the-shelf statistical model checking tools to evaluate arbitrary generalized planning methods according to arbitrary trajectory-based properties.

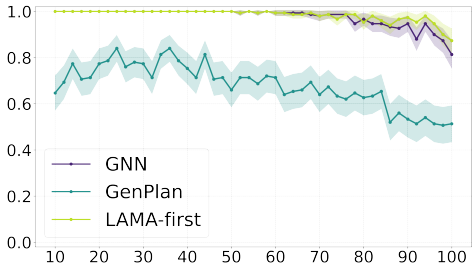
We plan to further improve our general evaluation framework in the near future, especially with respect to the modification of instance generators using LLMs and theorem



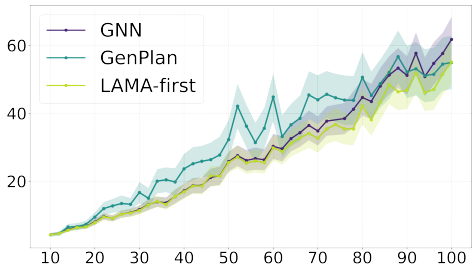
(a) Coverage



(b) Plan length



(c) Coverage on constrained instances



(d) Plan length on constrained instances

Figure 3: Evaluation on Satellite

provers. At the moment, we do not explicitly test whether the instances returned by the LLM-generated generators are legal instances of the domain. For example, a Blocksworld generator could pass the test suite with no bugs but return instances with floating blocks. However, we have not yet observed such failures, which is likely because the original generators implicitly encode legality constraints, and thus the LLM accounts for them. Nonetheless, we plan to address this by additionally checking the constraints for legal instances during testing. While this increases the manual effort for modeling constraints and the runtime for checking them, it would also provide our approach with a formal guarantee that only legal instances are returned.

Once legality constraints are incorporated into our pipeline, it would be interesting to see whether we can generate instance generators from scratch by only providing a domain file and legality constraints. This would be similar to the domain-independent instance generator of Grundke, Helmert, and Röger, but it would rely on LLMs and theorem provers instead of answer set programming, which could be more efficient (2025). A challenge for these experiments would be to design novel domains that were not part of the LLM's training data.

Further, we plan to provide the LLM with concrete feedback on the diversity of the instances that its generator returns. During each iteration of the modification pipeline, we could compute the diversity of all instances without any bugs and provide a diversity score as feedback to the LLM. The final generator could then be selected as the one that balances the number of bugs and diversity.

Lastly, we plan to conduct more experiments, which will include testing different LLMs in our modification pipeline, testing more complex constraints for instance distributions across more domains, and evaluating more generalized planning approaches according to more properties.

Acknowledgements

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) as part of the project MAC-MERLin (Grant Agreement No. 01IW24007), by the German Research Foundation (DFG) - GRK 2853/1 "Neuroexplicit Models of Language, Vision, and Action" - project number 471607914, by the European Regional Development Fund (ERDF) and the Saarland within the scope of (To)CERTAIN, and by Israel's Ministry of Innovation, Science and Technology (MOST) grant #0008683/1001948181.

References

Abdulaziz, M.; Pommerening, F.; and Corrêa, A. B. 2022. Mechanically Proving Guarantees of Generalized Heuristics: First Results and Ongoing Work. In *ICAPS 2022 Workshop on Heuristics and Search for Domain-independent Planning*.

Bonet, B.; Frances, G.; and Geffner, H. 2019. Learning Features and Abstract Actions for Computing Generalized Plans. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 2703–2710.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI Gym. *arXiv preprint arXiv:1606.01540*.

De Moura, L.; and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 337–340. Springer.

Gros, T. P.; Hartmanns, A.; Hoese, I.; Meyer, J.; Müller, N. J.; and Wolf, V. 2025a. PyDSMC: Statistical Model Checking for Neural Agents Using the Gymnasium Interface. In *International Conference on Quantitative Evalua-*

tion of Systems and Formal Modeling and Analysis of Timed Systems, 134–156. Springer.

Gros, T. P.; Müller, N. J.; Fišer, D.; Valera, I.; Wolf, V.; and Hoffmann, J. 2025b. Per-Domain Generalizing Policies: On Validation Instances and Scaling Behavior. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 35, 198–203.

Grundke, C.; Helmert, M.; and Röger, G. 2025. Domain-Independent Instance Generation for Classical Planning. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 22, 805–809.

Müller, N. J.; Oster, M.; Valera, I.; Hoffmann, J.; and Gros, T. P. 2026. Per-Domain Generalizing Policies: On Learning Efficient and Robust Q-Value Functions. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 36, 703–707.

Núñez-Molina, C.; Mesejo, P.; and Fernández-Olivares, J. 2024. NeSIG: A Neuro-Symbolic Method for Learning to Generate Planning Problems. In *ECAI 2024*, 4084–4091. IOS Press.

Núñez-Molina, C.; Mesejo, P.; and Fernández-Olivares, J. 2025. Automated planning instance generation with neuro-symbolic AI. *Artificial Intelligence*, 104471.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Silver, T.; and Chitnis, R. 2020. PDDLGym: Gym Environments from PDDL Problems. In *International Conference on Automated Planning and Scheduling (ICAPS) PRL Workshop*.

Silver, T.; Dan, S.; Srinivas, K.; Tenenbaum, J. B.; Kaelbling, L.; and Katz, M. 2024. Generalized Planning in PDDL Domains with Pretrained Large Language Models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 20256–20264.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022a. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, 629–637.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022b. Learning Generalized Policies without Supervision Using GNNs. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 19, 474–483.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2025. Learning More Expressive General Policies for Classical Planning Domains. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 26697–26706.

Ståhlberg, S.; and contributors. 2026. Mimir: A Generalized Planning Library. https://github.com/simon-stahlberg/mimir. GitHub repository. Accessed: 2026-05-07.

Stein, K.; Hodel, N.; Fišer, D.; Hoffmann, J.; Katz, M.; and Koller, A. 2026. Improved Generalized Planning with LLMs through Strategy Refinement and Reflection. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 36, 677–686.

Taitler, A.; Alford, R.; Espasa, J.; Behnke, G.; Fišer, D.; Gimelfarb, M.; Pommerening, F.; Sanner, S.; Scala, E.; Schreiber, D.; et al. 2024. The 2023 International Planning Competition. *AI Magazine*, 45(2): 280–296.

Torralba, A.; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 376–384.

Towers, M.; Kwiatkowski, A.; Terry, J.; Balis, J. U.; De Cola, G.; Deleu, T.; Goulão, M.; Kallinteris, A.; Krimmel, M.; KG, A.; et al. 2024. Gymnasium: A Standard Interface for Reinforcement Learning Environments. *arXiv preprint arXiv:2407.17032*.