Integrating the Unified Planning Framework via MCP with Large Language Models for Reliable Automated Planning

João Areias Saraiva[1], **Thomas Kirste**[1],

[1]Hybrid Methods in Artificial Intelligence and Machine Learning
Department of Computer Science
University of Rostock, Germany

Abstract

Large Language Models (LLMs) have been increasingly used for planning, from everyday tasks to complex jobs that require formal guarantees and provenance. For the latter case, multiple studies argue LLMs should not replace formal planners, but rather make them substantially easier to access. Recent approaches therefore use LLMs to formalize planning problems, such as in PDDL files, and then delegate plan search to mature engines that are far more reliable than direct LLM plan generation. Yet generating full symbolic specifications as free text in one go has proven brittle. We propose a procedural formalization interface, where an LLM can formalize planning problems step-wise via tool calls to a specialized modeling and planning backend (such as Unified Planning). Across 2,550 Blocksworld, Logistics, and Rovers planning problems, using small open-weight models (Qwen 3.5 and GPT-OSS), problem formalization accuracy and plan validity were 98-100% and 99-100%, respectively, outperforming text-to-PDDL synthesis and prior direct LLM planning studies.

1 Introduction

Large language models (LLMs) have rapidly become a user interface for sequential decision-making systems. This has driven a fast-growing body of work on planning with LLMs across informal planning agents for common everyday use, as well as formal planning agents for scientific use, robotics, and other applications (Huang et al. 2022; Yao et al. 2023; Boiko et al. 2023; Lu et al. 2026; Bran et al. 2024; Ahn et al. 2023; Agarwal et al. 2025; Zhang and Luo 2025; Huang et al. 2024). At a high level, this literature is motivated by a compelling promise: users should be able to describe goals and constraints in natural language, while an intelligent system handles the difficult process of turning those into executable plans.

A first family of approaches uses the LLM itself as the *planner*, asking it to produce a plan exploiting its encoded linguistic fluency and commonsense knowledge (Huang et al. 2022; Yao et al. 2023; Wang et al. 2023, 2024; Zhao, Lee, and Hsu 2023; Singh et al. 2023). However, while LLMs are often good at producing plausible action sequences, they are unreliable at enforcing formal constraints, maintaining long-horizon consistency, and guaranteeing formal validity (Valmeekam et al. 2023b,a; Liu et al. 2023; Huang and Zhang 2025; Zhang et al. 2026; Monti et al. 2026; Jiang et al. 2026). As a consequence, direct LLM planning tends to blur the distinction between generating a convincing explanation of a plan and actually solving a planning problem. To prevent such unconstrained language planning, some works link the LLM with external grounding signals, affordance models, or execution feedback (Ahn et al. 2023; Huang et al. 2023; Yao et al. 2023; Singh et al. 2023; Liang et al. 2023; Wu et al. 2023; Wang et al. 2024; Chen et al. 2024). But even these grounded systems do not fundamentally resolve the mismatch between language generation and formal combinatorial search (Liu et al. 2023; Agarwal and Sreepathy 2024; Chen et al. 2024; Gestrin, Kuhlmann, and Seipp 2024; Gundawar et al. 2024; Huang and Zhang 2025; Jiang et al. 2026; Hua et al. 2026). Hence, additional mechanisms are required precisely because an LLM by itself cannot guarantee soundness or consistency.

A second line of work uses the LLM not as the *planner* itself, but as a *formalizer* (Tantakoun, Muise, and Zhu 2025), so that it produces a formal problem specification, with which a formal planning engine can perform actual search and planning. Most studies evaluate whether LLMs can write formal problem specifications in PDDL within fixed benchmark domains (Liu et al. 2023; Agarwal and Sreepathy 2024; Zuo et al. 2025; Guo, Kingston, and Kavraki 2025; Bai et al. 2024; Zeng and Li 2026; Shi et al. 2025), whereas others attempt also to generate domain-level structure (Gestrin, Kuhlmann, and Seipp 2024; Zhang et al. 2024; Yu et al. 2025; Huang et al. 2025; La Malfa et al. 2025; Huang and Zhang 2025). Yet this symbolic turn exposes a second bottleneck. Recent work shows that directly producing raw PDDL text with current LLMs remains challenging due to frequent syntactic errors, wrong predicate signatures, structural inconsistency, and semantic inaccuracy (Zuo et al. 2025; Zhang et al. 2024; Vyas et al. 2025; Yu et al. 2025; Chen et al. 2024; Huang and Zhang 2025). This suggests that the key issue is not merely whether an LLM can output a formal language string, but whether the interface between the LLM and the planner is the right one.

In this line, it is clear that formalization and combinatorial search must be separate responsibilities. However, most current literature still entangles different processes at the

This work was funded by the European Union's 2021 HORIZON program, under the grant 101071485 of the Marie Skłodowska-Curie Actions (MSCA) Doctoral Network entitled CombiDiag.

formalization stage. Here we argue for a stronger architectural separation: natural language interpretation and formal problem construction must also be separate responsibilities. This means that the problem symbolic representation is not maintained nor validated by the LLM, but it is constructed via LLM suggestions based on its natural language interpretation. As a result, this requires an interface between such a backend that maintains the problem representation and an LLM, as recently pointed out by Kang (2025), arguing for intermediate representations. Once the problem generation is constrained by types, schemas, or stepwise structured decisions, LLM formalization becomes more accurate and easier to validate, repair, and integrate with downstream reasoning modules (Sumers et al. 2024; Huang et al. 2024). Thus, in order to improve formalization accuracy, we argue that such an interface must allow (Figure 1, bottom panel):

- **Incremental Construction:** Each planning problem should be constructed in steps. This might be one declaration at a time or multiple ones in batches, depending on the LLM used. Figure 1 (bottom panel) illustrates this concept in blue: if we think of the backend-maintained problem description as a *node*-based abstract data structure, *nodes* can be added in arbitrary order, increasing the degrees of freedom in which an LLM may formalize a problem.
- **Local Revision:** When the problem needs to be repaired (adding, removing or modifying declarations), the LLM may do so by referring only to the relevant declarations. Figure 1 (bottom panel) illustrates this concept in yellow, where any existing *node* of a formalized problem can be edited in an isolated manner.
- **Step-wise Semantic Validation:** Both construction and revision must go through validation by the backend that guarantees the problem representation is kept semantically valid until it reaches a planner (Figure 1, in green: each *node* creation or modification must be valid).

The existence of such an interface, would allow the LLM to formalize problems in a procedural manner. In contrast, text-to-PDDL approaches generate declarative formalizations, without incremental construction, and where revision implies re-generating the full PDDL text (Figure 1, top panel).

To realize this procedural paradigm, we propose a tool-based architecture with the Unified Planning (UP) framework (Micheli et al. 2025), because UP (i) is LLM-independent and planner-independent, (ii) provides a suitable abstraction layer for different planning paradigms (classical, numeric, temporal, hierarchical, etc.), and (iii) already integrates the connection to formal engines. The goals of this study are to:

- Implement a procedural formalization interface, in the style of ReAct (Yao et al. 2023), in which an LLM builds planning problems via tool calls, not maintaining the symbolic state of its own formalizations, and delegating plan search to formal solvers;
- Test the proposed approach on the largest corpus across 3 toy domains, which we make available for benchmarking, and evaluate whether it generalizes across these domains;

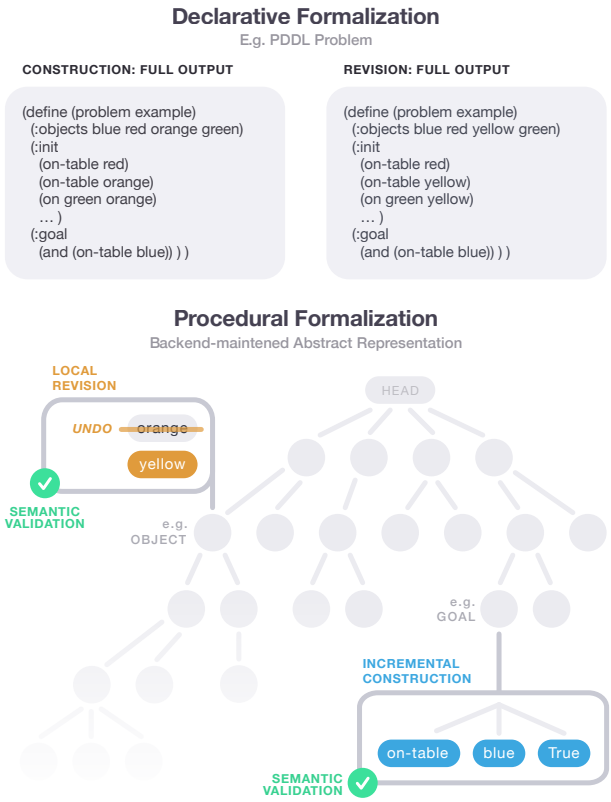

Figure 1: Formalization Paradigms Differences. Top panel: Declarative formalization entangling natural interpretation with formal problem construction, and generating planner-ready text in one go, including full re-generation when revising. Bottom panel: Procedural formalization (proposed method), here visualized as a hierarchy of nodes, that can be constructed / traversed in any order and revised locally.

- Show that tool-mediated symbolic state maintenance improves problem formalization, while performance stays robust as problem complexities increase;
- Evaluate whether the proposed approach outperforms direct PDDL generation and direct plan generation.

2 Proposed Method

2.1 Problem Setup

Let x denote a natural-language planning problem, where the user specifies the initial state and the desired goal state. Consider the trivial example where x is:

> "*I have four blocks piled up in a stack: the red block (sitting on the table), the blue block, the green block, and the yellow block. I want to restack them, from the bottom to the top: red, green, yellow, and blue.*"

The goal of the system is to produce an answer, y, in natural-language as well, describing a plan that solves problem x. In order to do that, following the aforementioned *LLM-as-formalizer* approach, the LLM must first model a formal planning problem, $\Pi = (O, I, G, M)$, where O is the

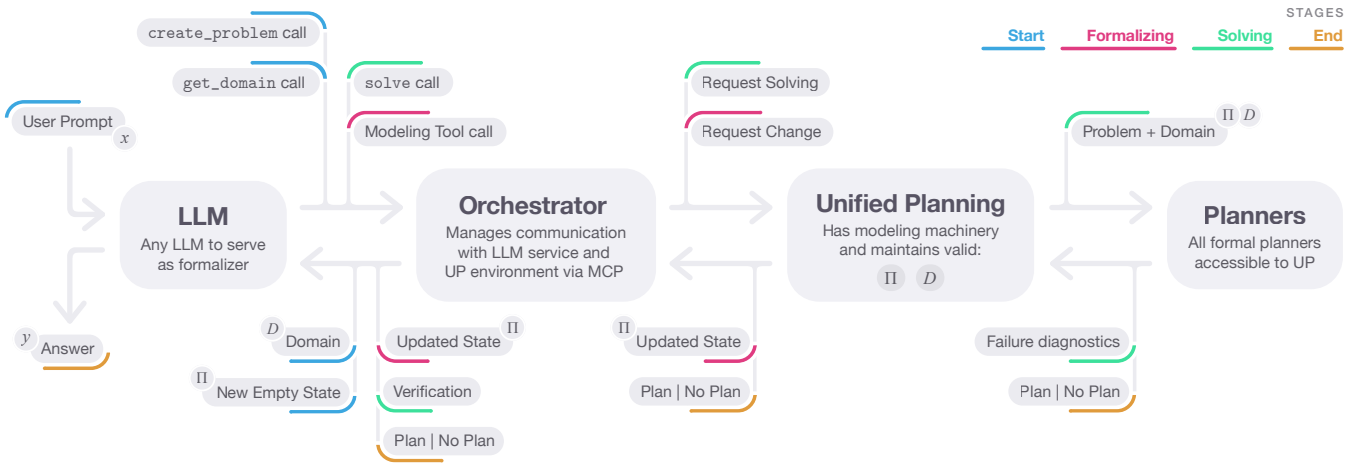

Figure 2: High-level architecture of the proposed method. Stage Blue: User starts by describing a planning problem in natural language, x. Stage Pink: In a loop of modeling tool calls, an LLM controls a Unified Planning environment, where the current state of the planning formalization, Π, and the domain, $\mathcal{D}$, live. Stage Green: When the problem is fully modeled, the LLM can trigger the solving of Π in the UP environment, and, in turn, provide an answer, y, to the user – Stage Yellow.

object set, I the initial conditions set, G the goals set, and M the metrics set. To do this, the LLM has access to modeling tools and a domain description, $\mathcal{D}$. The model has also access to formal engines to solve Π, and from their output then produce y. Hence, at a high level, y is produced from the output of $\text{PLANNER}(\mathcal{D}, \Pi \leftarrow \text{LLM}(x, \mathcal{D}))$, where the LLM true responsibility is modeling Π, while plan search is delegated to a formal planner.

2.2 High-level Architecture

The proposed architecture consists primarily of four components, as summarized in Figure 2. First is any LLM with sufficient inference power to interpret x and effectively place tool calls to formalize it. Second is an orchestrator which makes those tools available to the LLM via Model Context Protocol (MCP). These tools give the LLM access to the Unified Planning (UP) programmatic interface. Third is a UP environment where problem representations and domain signatures live. Fourth is a collection of planners. Conveniently, UP offers multiple planning engines and can select the appropriate engine to solve any problem, based on the problem characteristics.

A fresh UP environment is created on every session, containing only the programmatic interface and one specific domain information, $\mathcal{D}$. The UP framework allows to formalize and solve planning problems by building UP `Problem` instances (Micheli et al. 2025), which represent planning problems in a first-order logic manner, that is equivalent to PDDL expressiveness, but with far more formalization capabilities. The following sub-sections describe how components interact to achieve formalization and solving.

2.3 Tools Interface

We created a compact collection of tools that expose UP functionality for problem modeling and solving, without exhaustively exposing all of the UP library API, and made them available to the LLM via MCP. We can group these tools into **modeling** and **non-modeling** tools.

The **modeling** tools include `declare_object`, `declare_initial_condition`, `declare_goal`, `declare_metric`, and `undo`. These tools use the necessary UP API to modify UP `Problem` instances. In an abstract way, one can think that each of these tools creates or modifies a *node* of a `Problem` instance, as illustrated in Figure 1. Additionally, because different LLM families behave differently, we also make available batch-versions of those tools. For evaluation purposes that are subsequently explained, we also make available `declare_problem`, which allows to declare the complete problem with a single tool call, presenting a trade-off between incremental construction and time efficiency.

The **non-modeling** tools include `create_problem` (creates an empty UP `Problem` instance in the UP environment, Π), `get_domain` (returns a structured summary of the domain, $\mathcal{D}$, including its set of types, set of predicates/fluents, set of actions, and annotations), and `solve` (invokes via UP a planner to solve the specified `Problem` instance and returns its output).

Further details about these tools can be found in the Appendix A.1. Moreover, given that the majority of tool-calling standards rely on JSON, and that the LLMs we would be using for evaluation behave well with JSON, we defined JSON contracts to communicate Π and $\mathcal{D}$ between the LLM and the UP environment, which can be found in Appendix A.4.

2.4 Formalization Strategy and Policy

A session to process a user's planning problem is made of multiple interaction turns (or time points). The entry point at $t = 1$ is the user prompt, x, which is preceded by a system message briefing the model on the tools available and instructing it to first inspect the domain, then create an empty problem instance, and only then populate it. This control flow is il-

lustrated as the (blue) 'Start' stage in Figure 2, where the first tool calls are usually `get_domain` and `create_problem`. In the former, the model receives $\mathcal{D}$ in the defined JSON contract. In our previous classic Blocksworld example, this would mean the LLM now knows that it can use `clear`, `on-table`, `arm-empty`, `holding`, and `on` predicates to formalize the problem. The latter creates an empty UP `Problem` instance, $\Pi_{t=1} = (\varnothing, \varnothing, \varnothing, \varnothing)$, associated with a unique identifier. Multiple problem instances can be created.

With a problem identifier, the LLM can start modifying the corresponding Π to match the user's description, using the available modeling tools (Figure 2, stage 'Formalizing'). The state-construction process is incremental, starting from an empty state (Π_1), and at any turn t the model generates a tool call producing either a new formal state Π_{t+1} or an error message if the proposed update is invalid, producing $\Pi_{t+1} = \Pi_t$. A tool call is successful when UP accepts the modeling delta, in which case the state update is deterministic because it is implemented by the planning backend rather than by the model. The UP environment might send corrective feedback when the request is wrong or inadequate (e.g. declaring an object with a type inexistent in the domain, declaring an initial condition with an object that has not been declared before, declaring a goal with a 2-argument literal when the corresponding domain fluent takes 1 argument only, undoing a declaration not in Π, etc.).

Every time a problem is modified, the respective updated Π is refreshed in the context window, so that the next tool call is always based on the current state of the active problem (Figure 2, bottom pink message tagged with Π). This externalization of symbolic memory makes formalization depend on an explicit, machine-checked state rather than on the model's latent recollection of previous tokens.

2.5 Verification Mechanism and Solving

Once the LLM *believes* the current state of Π matches the user's description, it should call `solve`. Whenever `solve` is called, the orchestrator intercepts with an extra self-verification turn. On this verification turn, the LLM is instructed to verify if the formalized problem is complete and faithful to x and $\mathcal{D}$ – full prompt is available in Appendix B. This mechanism forces the LLM to check the adequacy of its own formalization, which is a common good practice, given the generative nature of these models. Depending on the verification feedback, the LLM may proceed with calling `solve` again, at which time the orchestrator actually requests UP to invoke a suitable planner on Π and $\mathcal{D}$ (Figure 2, stage 'Solving'). Or the LLM can amend Π with modeling tools or, in the limit, start over by creating a fresh problem instance. If it decides to revise, the verification gate is closed, requiring again two consecutive `solve` calls to access the planner. Appendix A.7 further details this mechanism.

When a planner is called to solve a problem, UP validates its output. A terminal output is either when a (satisfying or optimal) plan has been found or whether the problem is proven unsolvable; whereas a non-terminal output is everything else. Common **non-terminal outputs** occur because of planner pre-processing issues or when grounding fails before search for a solution has started. Non-terminal outputs and diagnostics are sent back to the LLM so it can amend its formalization and try `solve` again. **Terminal outputs** are also fed back to the LLM in the following turn (Figure 2, stage 'End'), at which point the LLM can reply in natural language to the user with the planner's solution.

3 Evaluation

3.1 Dataset Generated

We use three planning domains to evaluate the proposed method: the Blocksworld, the Logistics, and the Rovers domains – a selection similar to previous works on planning with LLMs. All of these are standard benchmark domains introduced in International Planning Competition (IPC) suites (McDermott 2000; Bacchus 2001; Long and Fox 2003).

A large corpus of 2550 different problems was created in PDDL with standard generators. This corpus is divided in 850 instances per domain, and 17 complexity levels per domain: 50 instances per complexity per domain. Given that our task is formalization rather than plan search, problem complexity is defined as proportional to description size, i.e., total number of declarations (sum of objects, initial conditions, goals, and metrics in the target formalization). Table 1 shows the decomposition of 850 instances per domain by problem complexity, where complexity increases linearly with the total number of declarations. Given the long-standing use of Blocksworld as a complexity-controlled symbolic reasoning domain, the complexity of a problem in this domain can also be seen as the number of blocks. Appendix D provides more details about the dataset.

For the natural language prompts (serving as the x prompts), we used the generators made available in PlanBench (Valmeekam et al. 2023a) for the Blocksworld and Logistics problems, which give us heavy-templated descriptions in the style "I have" – "I want", as described in the previous example of Section 2.1. For Rovers problems, we adapted the PlanBench method to generate templated, but more natural prompts – available in the Supplementary Material (SM). All prompts are zero-shot style, therefore the LLM makes no in-context learning (ICL). The complete corpus is made available in the SM.

3.2 Metrics

After a terminal output, UP serializes the plan outputted by the planner, if any, as well as the state of Π in an equivalent PDDL artifact. Both of these are used to evaluate the accuracy of the proposed method, in two different ways. First we are interested in the **formalization accuracy**, since the model is tasked with formalizing the natural language description of a problem, rather than solving it. We compare the serialized Π with the PDDL ground-truth via graph isomorphism with normalized object names – see Appendix E.1. From the optimal alignment between the two, we get the matching, extra, and missing declarations. An instance is considered an *exact match* (marked as accurate) iff, under the optimal alignment, there are zero missing and zero extra declarations. Otherwise, it is inaccurate.

Table 1: Performance of proposed method across problem complexities, measured in number of accurate problem formalizations and number of valid plans per total number of instances, grouped by Qwen 3.5 122B ('Q') and GPT-OSS 120B ('G') models.

Complexity	Blocksworld			Logistics			Rovers			Instances
	Total Declarations ($\mathcal{B}$ is number of blocks)	**Accurate Problems** Q | G	**Valid Plans** Q | G	Total Declarations	**Accurate Problems** Q | G	**Valid Plans** Q | G	Total Declarations	**Accurate Problems** Q | G	**Valid Plans** Q | G	
1	11 – 15 ($\mathcal{B} = 4$)	**50** | **50**	**50** | **50**	30	**50** | **50**	**50** | **50**	35 – 54	**50** | **50**	**50** | **50**	50
2	14 – 18 ($\mathcal{B} = 5$)	**50** | **50**	**50** | **50**	49	**50** | **49**	**50** | **50**	55 – 74	**50** | **50**	**50** | **50**	50
3	17 – 21 ($\mathcal{B} = 6$)	**50** | **49**	**50** | **50**	59	**50** | **50**	**50** | **50**	75 – 94	**50** | **49**	**50** | **49**	50
4	20 – 24 ($\mathcal{B} = 7$)	**50** | **50**	**50** | **50**	65	**50** | **50**	**50** | **50**	95 – 114	**50** | **50**	**50** | **50**	50
5	23 – 27 ($\mathcal{B} = 8$)	**50** | **50**	**50** | **50**	74	**50** | **50**	**50** | **50**	115 – 134	**50** | **50**	**50** | **50**	50
6	26 – 30 ($\mathcal{B} = 9$)	**50** | **49**	**50** | **49**	82	**50** | **50**	**50** | **50**	135 – 154	**50** | **50**	**50** | **50**	50
7	29 – 33 ($\mathcal{B} = 10$)	**50** | **50**	**50** | **50**	89	**50** | **50**	**50** | **50**	155 – 174	**48** | **50**	**50** | **50**	50
8	32 – 36 ($\mathcal{B} = 11$)	**50** | **50**	**50** | **50**	97	**50** | **50**	**50** | **50**	175 – 194	**50** | **49**	**50** | **49**	50
9	35 – 39 ($\mathcal{B} = 12$)	**50** | **50**	**50** | **50**	104	**50** | **50**	**50** | **50**	195 – 214	**50** | **49**	**50** | **49**	50
10	38 – 42 ($\mathcal{B} = 13$)	**50** | **49**	**50** | **49**	111	**50** | **50**	**50** | **50**	215 – 234	**50** | **50**	**50** | **50**	50
11	41 – 45 ($\mathcal{B} = 14$)	**50** | **50**	**50** | **50**	120	**50** | **50**	**50** | **50**	235 – 254	**50** | **50**	**50** | **50**	50
12	44 – 48 ($\mathcal{B} = 15$)	**50** | **50**	**50** | **50**	128	**50** | **50**	**50** | **50**	255 – 274	**49** | **50**	**50** | **50**	50
13	47 – 51 ($\mathcal{B} = 16$)	**50** | **50**	**50** | **50**	136	**50** | **50**	**50** | **50**	275 – 294	**50** | **49**	**50** | **49**	50
14	50 – 54 ($\mathcal{B} = 17$)	**50** | **50**	**50** | **50**	147	**50** | **50**	**50** | **50**	295 – 314	**50** | **48**	**50** | **49**	50
15	53 – 57 ($\mathcal{B} = 18$)	**50** | **50**	**50** | **50**	157	**50** | **50**	**50** | **50**	315 – 334	**50** | **50**	**50** | **50**	50
16	56 – 60 ($\mathcal{B} = 19$)	**50** | **50**	**50** | **50**	173	**50** | **50**	**50** | **50**	335 – 354	**50** | **48**	**50** | **48**	50
17	59 – 63 ($\mathcal{B} = 20$)	**50** | **50**	**50** | **50**	204	**49** | **48**	**49** | **48**	355 – 374	**48** | **49**	**49** | **49**	50

Secondarily, we are interested in the **plan validity**. We validate the plan against the domain PDDL and the ground-truth problem PDDL using `VAL`, which checks whether the action sequence is executable from the initial state and whether the resulting state satisfies the goal. We mark a plan as valid iff `VAL` accepts the full sequence; otherwise, it is marked as invalid. In this study, we are not concerned with plan optimality, because that depends on the search algorithm. To expedite evaluation, the planners UP selected (FastDownward and ENHSP) were configured to search for a satisfying plan.

3.3 Comparison with Baseline Method

We compared the proposed method with direct PDDL generation, following the declarative formalization paradigm illustrated in Figure 1. Because this approach has been evaluated in multiple prior studies, we will call it **baseline** approach.

In this approach, the model is asked to generate a full problem PDDL. The model receives the same instructions, the same user natural language requests, and the domain description, in the original PDDL format. Model formalizations go through the same self-verification mechanism as in the proposed method, but repair requires to generate a new full PDDL problem (no incremental construction nor local revision). Moreover, given the absence of the UP environment, or any other backend environment to validate the model's formalizations, the only feedback it receives is from the verification mechanism and, eventually, planner diagnostics.

3.4 Models

We evaluated the proposed and baseline methods using two public LLMs: Qwen 3.5 with 122 billion parameters, `Q4_K_M` quantized (herein referred as Qwen), and GPT-OSS with 120 billion parameters, `MXFP4` quantized (herein referred as OSS). Configuration of experiments and models is detailed in Appendix E.

4 Results

The next subsections present the performance of the proposed method on the above datasets, while comparing it to the baseline approach of direct PDDL generation.

4.1 Models faithfully compile natural-language requests into accurate symbolic specifications

The formalization accuracy for all instances across the benchmark domains, grouped by model is shown in Table 1. Out of 2550 instances, only 6 were not an exact match when formalized by the Qwen model, and 15 were not an exact match when formalized by the OSS model. These figures show approximate 0.24% and 0.59% global formalization error rates, respectively, across the benchmark domains. Most inaccurate formalizations were due to missing (in 40.9% of inaccuracies) and extra or spurious (in 81.8% of inaccuracies) initial conditions. Appendix F.1 lists all inaccurate formalizations.

In comparison, the baseline method – direct PDDL generation – on the same dataset (N=2550) showed higher formalization error rates of 7.06% when using the Qwen model, and of 5.53% when using the OSS model. Since both methods were evaluated on the same dataset, a paired two-sided McNemar test indicates that the proposed method had a substantially lower error rate than baseline and this difference was highly significant when using Qwen ($p = 7.16 \times 10^{-37}$) and OSS ($p = 9.14 \times 10^{-25}$) (Appendix F.2).

4.2 Some inaccurate formalizations translate downstream into valid plans

We also wanted to understand if accurate formalizations translate into valid downstream plans, and if some inaccurate formalizations do too. Every formalization that is an exact match, by definition, will lead to a valid plan, because `VAL` will deem the output plan as valid for the PDDL domain if the formalization is an exact match to the PDDL problem

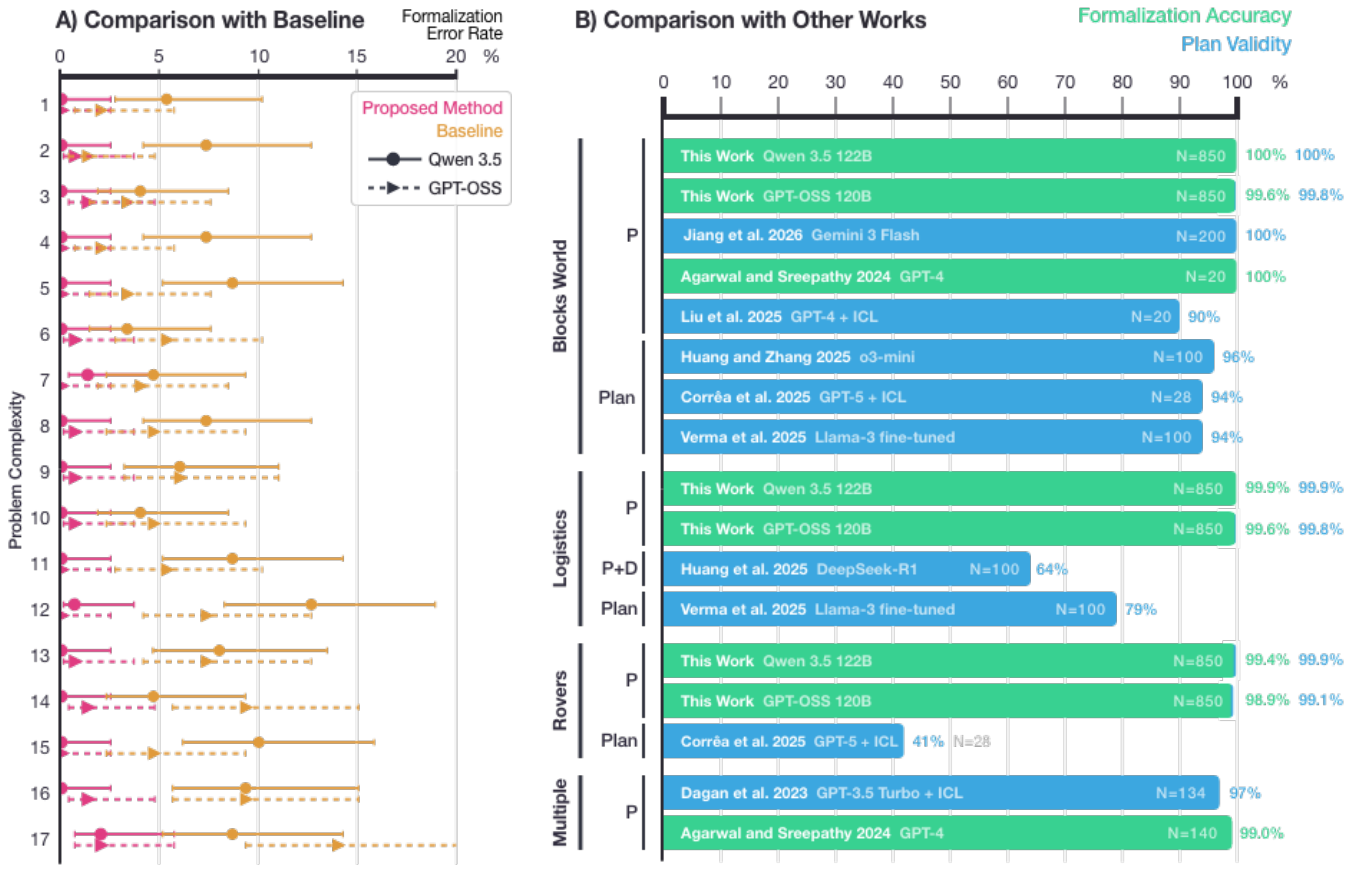

Figure 3: A) Formalization average error rate of the proposed method vs. text-to-PDDL (baseline), with 95% CI error bars. B) Performance comparison with previous approaches that formalize only the problem (P), problem and domain (P+D), or ask the model to directly generate a plan (Plan).

ground-truth. Thus, every accurate formalization always led to a valid plan. Conversely, inaccurate formalizations may or may not lead to valid plans. That will depend on the type of errors present in the formalization, and if those lead to invalid plans or unsolvable outcomes.

Out of 2550 instances, an invalid plan was outputted in 2 of them when using the Qwen model, and in 12 of them when using the OSS model. Therefore, across these benchmark domains, the global plan error rates are approximately 0.08% and 0.47%, respectively. All invalid plans were the consequence of inaccurate formalizations, and happened on different instances for each model.

4.3 Proposed method is robust to problem complexity and generalizes across domains

To assess whether accuracy significantly changes as problem size and declaration burden increase, Table 1 shows the accuracy of the proposed method decomposed in 17 formalization complexities. We fitted binomial logistic regression models for each domain, model, and accuracy metric, using the number of accurate outputs out of 50 trials as the response and complexity level as an ordinal predictor (Appendix F.4). Across all estimable models, the complexity coefficient was not statistically significant ($p > .05$), indicating no evidence that formalization accuracy or plan validity systematically changed with complexity, which is a strong indicator of robustness. Conversely, the formalization accuracy of the baseline method was significantly dependent on problem complexity ($p < .05$).

Figure 3A compares the formalization error rate of the proposed method (in pink) with baseline (in yellow) across complexity levels. Using both models, the proposed method yields on average lower formalization error rates at all 17 complexity levels. The Qwen model results show a stronger and more uniform evidence that the proposed method also improves error across all complexity levels ($q < 0.05$, McNemar tests, after Benjamini-Hochberg FDR correction), except levels 6 ($q = 0.079$) and 7 ($q = 0.204$). At many levels, the proposed method is near-zero error, while baseline remains several points higher, producing large significant gaps. The OSS model results shows a small effect at low complexity (levels 1–4 not significant), but grows in higher-complexity regimes, where several levels show large and significant reductions (Appendix F.2).

Additionally, we tested the association between domain and error occurrence with a Fisher–Freeman–Halton exact test for each accuracy measure. For the formalization accuracy, the association between domain and error occurrence was not statistically significant for Qwen ($p = .053$) and OSS

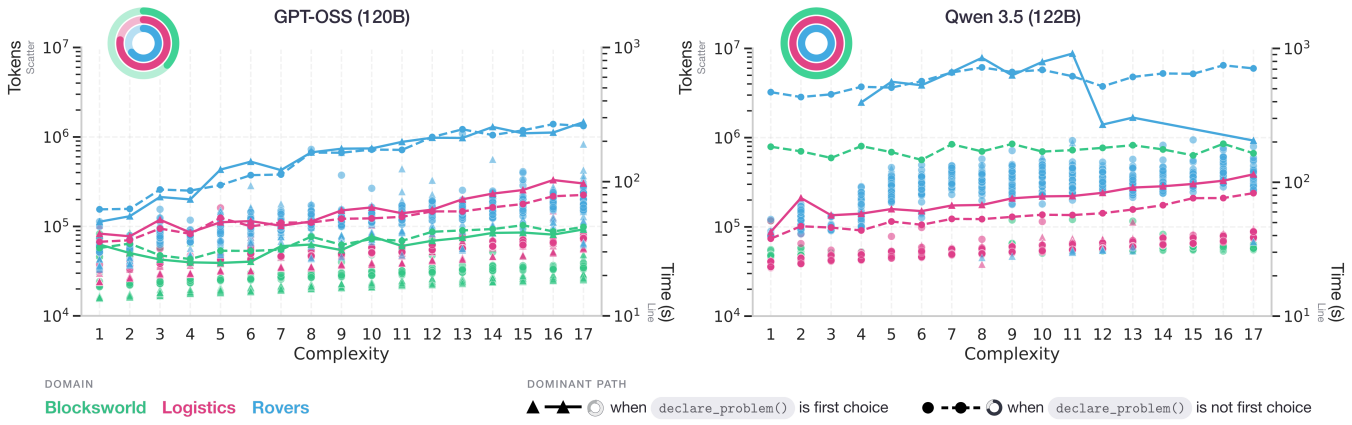

Figure 4: Output tokens and session total time across complexity levels, grouped by model (left/right) and domain (color), where two different pathways are is highlighted (triangle/circle markers). Top left insets: Proportion of instances when `declare_problem` is first choice (lighter shades), and when it is not (darker shades).

($p = .123$). Conversely, the association was statistically significant for the baseline results in both models ($p < 0.0001$) – see Appendix F.3. For the plan validity, the association was also not statistically significant for Qwen ($p = 1.0$) and OSS ($p = .070$). This indicates that the proposed method generalizes well across these three domains, which vary in structural properties, types of declarations required, and multiple ways of symbolically representing the same logical concepts. Nevertheless, these domains are scoped to classical and numerical planning toy problems, distant from real world problems and user demands. Further testing is needed to evaluate this approach on temporal, hierarchical, and graph planning, among other planning paradigms.

4.4 Unified Planning feedback prevents invalid and inaccurate formalizations

Backend maintenance of the formal problem representations and their continuous validation on construction and revision is the primary contributor for improved accuracy. We measured when UP validation errors prevented proposed symbolic updates from entering the maintained problem state. In all experiments run, 1,612 (out of 92,845) tool calls were rejected by UP (Appendix F.7). Structurally invalid updates, which are those UP can reject based on physical inconsistency or incoherence with the domain, were prevented in 30.3% of these rejections. Note that UP has access to the domain information when validating LLM-proposed changes to the problem instances.

Additionally, semantically invalid updates were prevented in 43.0% of these rejections. These latter class are updates that would have made the formalization inaccurate (when we compare with ground-truth problem), but not that UP does not have access to the ground-truth problem. These updates are *naively* prevented due to inconsistencies UP does not allow, most of them overlapping with the same as the structurally invalid updates. Appendix F.7 highlights some examples of this mechanism.

These results provide direct evidence that backend validation blocked changes that would have made the formalization inaccurate. This supports the hypothesis that backend validation is an important contributor, though this mechanistic audit does not fully isolate its causal effect.

4.5 How much tool interaction is needed?

Figure 4 shows the total outputted tokens and total time (including LLM inference and planner inference) of the proposed method. On average, Qwen calls 24.5 modeling tools to formalize a problem, whereas OSS calls 3.2 modeling tools. At the same time, sessions using Qwen call `undo` less times and go through less verification turns, than when using OSS (see Appendix F.6), indicating the need to revise more when using OSS. Regarding the usage of `undo`, OSS used it heavily in Rovers instances, signaling more difficult formalization trajectories. Regarding the higher number of verification turns, this mechanism was responsible for only 2.7% of runs transitioning from an inaccurate formalization to an accurate one, hence it was not a frequent repair mechanism (Appendix F.8). It is primarily the combination of tool usage and using a *chattier* language like JSON that constitutes a more accurate and reliable interface to formalize planning problems.

To understand if the faster formalization of OSS (Figure 4) explains its associated higher error rate, we investigated the possible pathways and shortcuts that the models might take. The models were instructed with *full liberty* regarding which modeling tools to call at every moment. Modeling tools ranged from allowing atomic declarations (in Figure 1 this would be modifying one *node* at a time), to allowing declarations of multiple pieces of problem information (multiple *nodes* at a time). There is even the `declare_problem` tool, which allows to formalize the complete problem in one go – see Subsection 2.3. This range of fine and coarse offers led to different formalization pathways naturally occurring.

The OSS model called `declare_problem` as the very first modeling tool for 63% of Blocksworld problems, 23% of Logistics problems, and 35% of Rovers problems (Figure 4, top left inset). One could suspect that the simplicity

of the Blocksworld domain compared to the others leads OSS to choose a faster pathway and formalize the problem in a single tool call. Conversely, the Qwen model showed higher formalization accuracy while rarely ($< 1\%$) calling `declare_problem`, indicating an added benefit of adhering to the incremental construction philosophy.

Overall, trajectories using `declare_problem` at least once achieved 1590/1598 accurate formalizations (99.50%), and coarse construction was not significantly associated with lower accuracy, compared to the trajectories that never used `declare_problem` (Appendix F.6). However, human inspection on pathways with individual declaration tools revealed more organized and structured thinking patterns, which leads to higher explainability and provenance (see chat histories in the Supplementary Material).

5 Discussion

Evaluations most closely related to our setting treat the **LLM as a *formalizer***, rather than as the *planner* itself, and have fixed-domains. Zuo et al. (2025) showed GPT-4o (zero-shot) was able to generate PDDL problem descriptions that are 96.1% parseable and 24.8% semantically correct, on Blocksworld, Gripper, and Floor Tile problems. The Blocksworld problems alone attain 65.51% valid plans (Figure 3), displaying a strong domain-dependent instability across the three used domains.

Most recently, Jiang et al. (2026) showed Gemini 3 Flash (zero-shot) can formalize 100% (N=200) of Blocksworld's PDDL problems also without prior examples (Figure 3). Conversely, Liu et al. (2023) showed GPT-4 with an in-context learning (ICL) approach can write Blocksworld PDDL problems that lead to 90% valid plans (Figure 3). Same problem formalization strategy in Grippers and Storage domains lead to 95% and 85% valid plans, respectively. Also using an ICL approach, Dagan, Keller, and Lascarides (2023) showed GPT 3.5 Turbo can translate natural language tasks into executable PDDL goals in 97% of ALFWorld problems (Figure 3).

Another closely related study is that of Agarwal and Sreepathy (2024), which also uses LLMs to formalize problems, the domain is fixed and available, and to avoid direct text-to-PDDL generation, it asks LLMs to produce logically interpretable intermediate representations, which are then compiled to PDDL problem files. With ICL, GPT-4 is able to formalize accurately 100% (N=140) of Blocksworld, Barman, Floortile, Grippers, Storage, Termes, Tyreworld problems. In zero-shot manner, GPT-4 attains 95% – 100% formalization accuracies across these seven domains (100% in Blocksworld – Figure 3). This study showed that using an intermediate representation that is much closer to natural language than to raw PDDL, dividing the formalization task into multiple steps, and using a backend semantic validation and repair mechanism improves accuracy.

Regarding the studies on both **problem and domain formalization** in raw PDDL, Huang and Zhang (2025) showed that o3-mini (zero-shot) attains 94% valid plans on the Blocksworld-100 dataset, and DeepSeek-R1 (zero-shot) attains 64% valid plans on the Logistics-100 dataset (Figure 3), indicating a domain-dependent instability. Although this work is also concerned with domain formalization, the domain is explained in natural language to the model. The same phenomenon happens with GPT-4 and ICL in Gestrin, Kuhlmann, and Seipp (2024), where 100% of Blocksworld problems (N=6) attain valid plans (only up to 12 actions), but performance decreases on other domains.

Huang and Zhang (2025) also compare their approach with **LLM direct planning**, where performance is kept for Blocksworld problems but significantly decreases for Logistics problems, showing that using the LLM as a *planner* leads to worse performance. Corrêa, Pereira, and Seipp (2025) also shows LLM direct planning attains poor plan validity on Blocksworld (94%) and Rovers (41%) problems (Figure 3) when using GPT-5 with ICL. When fine-tuned, a smaller model such as Llama 3 was showed by Verma *et al.* (2025) to be as capable as GPT-5, attaining 94% plan validity on Blocksworld problems, and 79% plan validity on Logistics problems. Benyamin et al. (2025) also show that GPT-OSS (20B), in a zero-shot manner, attains moderate performance (78%) in direct LLM planning in multiple domains, including Blocksworld, Rovers, Depots, and other classical and numerical domains. In this case, their approach outperforms GPT-5 direct planning, because it makes tools available to the model, including calling a planner, calling VAL and simulating plans, therefore making answers more reliable – a similar philosophy to our proposed method, but for finding a plan rather than formalizing the planning problem. Similarly, Göbel et al. (2026) makes simulation tools available to Claude Haiku 4.5, however this approach attains only 78% valid plans o Blocksworld problems (N=102). This might indicate that tool use might work better in formalization scenarios rather than in planning scenarios, whereas plan search is delegated to mature engines.

6 Conclusion

In this study, we show how LLM-based systems can excel at solving planning problems ($> 99.1\%$ plan validity), when tasked only with formalizing these problems via tool-calling to a specialized symbolic backend, whereas plan search is delegated to a formal planner. The proposed method allows LLMs to incrementally construct and locally revise their problem formalizations, while the symbolic backend semantically validates their every suggestion. This study shows the feasibility of using Unified Planning as a symbolic backend for LLMs, which is a necessary property for applications needing formal guarantees of correctness.

Compared to text-to-PDDL formalization and prior studies following multiple other approaches, we show that the proposed method is more accurate and highly robust to domain variance and problem complexity, while using small open-weight models, not requiring in-context learning (ICL) nor fine-tuning. To the best of our knowledge, we evaluated a novel natural language planning approach on the largest dataset reported in the literature (N=850 per domain; N=2550 total). However, further evaluation is needed on other benchmark domains, as well as with real-world problems, handling real user language, ambiguity, underspecification, and noisy instructions.

Data — doi.org/10.5281/zenodo.20094411
Results — doi.org/10.5281/zenodo.20094503

References

Agarwal, S.; and Sreepathy, A. 2024. TIC: Translate-infer-compile for Accurate "Text to Plan" Using LLMs and Logical Representations. In *Neural-Symbolic Learning and Reasoning*, volume 14980 of *Lecture Notes in Computer Science*, 222–244. Springer.

Agarwal, S.; Sreepathy, A.; Alonso, D. H.; and Lamba, P. 2025. LLM+reasoning+planning for Supporting Incomplete User Queries in Presence of APIs. In *Proceedings of the 40th International Conference on Logic Programming*, volume 416 of *Electronic Proceedings in Theoretical Computer Science*, 29–58.

Ahn, M.; Brohan, A.; Brown, N.; Chebotar, Y.; Cortes, O.; David, B.; Finn, C.; Fu, C.; Gopalakrishnan, K.; Hausman, K.; Herzog, A.; Ho, D.; Hsu, J.; Ibarz, J.; Ichter, B.; Irpan, A.; Jang, E.; Jauregui Ruano, R.; Jeffrey, K.; Jesmonth, S.; Joshi, N. J.; Julian, R.; Kalashnikov, D.; Kuang, Y.; Lee, K.-H.; Levine, S.; Lu, Y.; Luu, L.; Parada, C.; Pastor, P.; Quiambao, J.; Rao, K.; Rettinghouse, J.; Reyes, D.; Sermanet, P.; Sievers, N.; Tan, C.; Toshev, A.; Vanhoucke, V.; Xia, F.; Xiao, T.; Xu, P.; Xu, S.; Yan, M.; and Zeng, A. 2023. Do as I Can, Not as I Say: Grounding Language in Robotic Affordances. In *Proceedings of the 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, 287–318. PMLR.

Bacchus, F. 2001. The AIPS '00 Planning Competition. *AI Magazine*, 22(3): 47–56.

Bai, D.; Singh, I.; Traum, D.; and Thomason, J. 2024. TwoStep: Multi-agent Task Planning Using Classical Planners and Large Language Models. arXiv:2403.17246.

Benyamin, Y.; Mordoch, A.; Shperberg, S. S.; and Stern, R. 2025. Toward PDDL Planning Copilot. arXiv:2509.12987.

Boiko, D. A.; MacKnight, R.; Kline, B.; and Gomes, G. 2023. Autonomous Chemical Research with Large Language Models. *Nature*, 624: 570–578.

Bran, A. M.; Cox, S.; Schilter, O.; Baldassari, C.; White, A. D.; and Schwaller, P. 2024. Augmenting Large Language Models with Chemistry Tools. *Nature Machine Intelligence*, 6(5): 525–535.

Chen, Y.; Arkin, J.; Dawson, C.; Zhang, Y.; Roy, N.; and Fan, C. 2024. AutoTAMP: Autoregressive Task and Motion Planning with LLMs as Translators and Checkers. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 6695–6702. IEEE.

Corrêa, A. B.; Pereira, A. G.; and Seipp, J. 2025. The 2025 Planning Performance of Frontier Large Language Models. arXiv:2511.09378.

Dagan, G.; Keller, F.; and Lascarides, A. 2023. Dynamic Planning with a LLM. arXiv:2308.06391.

Gestrin, E.; Kuhlmann, M.; and Seipp, J. 2024. NL2Plan: Robust LLM-driven Planning from Minimal Text Descriptions. arXiv:2405.04215.

Göbel, K.; Lorang, P.; Zips, P.; and Glück, T. 2026. Agentic LLM Planning via Step-Wise PDDL Simulation: An Empirical Characterisation. arXiv:2603.06064.

Gundawar, A.; Valmeekam, K.; Verma, M.; and Kambhampati, S. 2024. Robust Planning with Compound LLM Architectures: An LLM-modulo Approach. arXiv:2411.14484.

Guo, W.; Kingston, Z.; and Kavraki, L. E. 2025. CaStL: Constraints as Specifications through LLM Translation for Long-Horizon Task and Motion Planning. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, 11957–11964. IEEE.

Hua, K.; Wang, D.; Gu, Y.; and Ma, X. 2026. DUPLEX: Agentic Dual-System Planning via LLM-driven Information Extraction. arXiv:2603.23909.

Huang, C.; and Zhang, L. 2025. On the Limit of Language Models as Planning Formalizers. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 4880–4904. Association for Computational Linguistics.

Huang, S.; Wu, Y.; Shi, G.; Sukhatme, G. S.; and Kumar, V. 2025. SPAR: Scalable LLM-based PDDL Domain Generation for Aerial Robotics. arXiv:2509.13691.

Huang, W.; Abbeel, P.; Pathak, D.; and Mordatch, I. 2022. Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, 9118–9147. PMLR.

Huang, W.; Xia, F.; Xiao, T.; Chan, H.; Liang, J.; Florence, P.; Zeng, A.; Tompson, J.; Mordatch, I.; Chebotar, Y.; Sermanet, P.; Jackson, T.; Brown, N.; Luu, L.; Levine, S.; Hausman, K.; and Ichter, B. 2023. Inner Monologue: Embodied Reasoning through Planning with Language Models. In *Proceedings of the 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, 1769–1782. PMLR.

Huang, X.; Liu, W.; Chen, X.; Wang, X.; Wang, H.; Lian, D.; Wang, Y.; Tang, R.; and Chen, E. 2024. Understanding the Planning of LLM Agents: A Survey. arXiv:2402.02716.

Jiang, O.; Huang, C.; Sabharwal, A.; and Zhang, L. 2026. Language Model Planners Do Not Scale, but Do Formalizers? arXiv:2603.23844.

Kang, J. 2025. Scaling LLM Planning: NL2FLOW for Parametric Problem Generation and Rigorous Evaluation. arXiv:2507.02253.

La Malfa, E.; Zhu, P.; Marro, S.; Bernardini, S.; and Wooldridge, M. 2025. An End-to-End Planning Framework with Agentic LLMs and PDDL. arXiv:2512.09629.

Liang, J.; Huang, W.; Xia, F.; Xu, P.; Hausman, K.; Ichter, B.; Florence, P.; and Zeng, A. 2023. Code as Policies: Language Model Programs for Embodied Control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 9493–9500. IEEE.

Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biswas, J.; and Stone, P. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. arXiv:2304.11477.

Long, D.; and Fox, M. 2003. The 3rd International Planning Competition: Results and Analysis. *J. Artif. Int. Res.*, 20(1): 1–59.

Lu, C.; Lu, C.; Lange, R. T.; Yamada, Y.; Hu, S.; Foerster, J.; Ha, D.; and Clune, J. 2026. Towards End-to-End Automation of AI Research. *Nature*, 651: 914–919.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Mag.*, 21(2): 35–55.

Micheli, A.; Bit-Monnot, A.; Röger, G.; Scala, E.; Valentini, A.; Framba, L.; Rovetta, A.; Trapasso, A.; Bonassi, L.; Gerevini, A. E.; Iocchi, L.; Ingrand, F.; Köckemann, U.; Patrizi, F.; Saetti, A.; Serina, I.; and Stock, S. 2025. Unified Planning: Modeling, Manipulating and Solving AI Planning Problems in Python. *SoftwareX*, 29: 102012.

Monti, S.; Nicolini, C.; Pellegrini, G.; Staiano, J.; and Lepri, B. 2026. SokoBench: Evaluating Long-Horizon Planning and Reasoning in Large Language Models. *Transactions on Machine Learning Research*.

Shi, G.; Wu, Y.; Kumar, V.; and Sukhatme, G. S. 2025. PIP-LLM: Integrating PDDL-integer Programming with LLMs for Coordinating Multi-Robot Teams Using Natural Language. arXiv:2510.22784.

Singh, I.; Blukis, V.; Mousavian, A.; Goyal, A.; Xu, D.; Tremblay, J.; Fox, D.; Thomason, J.; and Garg, A. 2023. ProgPrompt: Program Generation for Situated Robot Task Planning Using Large Language Models. *Autonomous Robots*, 47: 999–1012.

Sumers, T. R.; Yao, S.; Narasimhan, K.; and Griffiths, T. L. 2024. Cognitive Architectures for Language Agents. arXiv:2309.02427.

Tantakoun, M.; Muise, C.; and Zhu, X. 2025. LLMs as Planning Formalizers: A Survey for Leveraging Large Language Models to Construct Automated Planning Models. In Che, W.; Nabende, J.; Shutova, E.; and Pilehvar, M. T., eds., *Findings of the Association for Computational Linguistics: ACL 2025*, 25167–25188. Vienna, Austria: Association for Computational Linguistics. ISBN 979-8-89176-256-5.

Valmeekam, K.; Marquez, M.; Olmo, A.; Sreedharan, S.; and Kambhampati, S. 2023a. PlanBench: An Extensible Benchmark for Evaluating Large Language Models on Planning and Reasoning about Change. In *Advances in Neural Information Processing Systems*, volume 36, 38975–38987.

Valmeekam, K.; Marquez, M.; Sreedharan, S.; and Kambhampati, S. 2023b. On the Planning Abilities of Large Language Models: A Critical Investigation. In *Advances in Neural Information Processing Systems*, volume 36, 75993–76005.

Vyas, K.; Graux, D.; Montella, S.; Vougiouklis, P.; Lai, R.; Li, K.; Ren, Y.; and Pan, J. Z. 2025. An Extensive Evaluation of PDDL Capabilities in Off-the-Shelf LLMs. arXiv:2502.20175.

Wang, G.; Xie, Y.; Jiang, Y.; Mandlekar, A.; Xiao, C.; Zhu, Y.; Fan, L.; and Anandkumar, A. 2024. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Transactions on Machine Learning Research*.

Wang, Z.; Cai, S.; Chen, G.; Liu, A.; Ma, X.; and Liang, Y. 2023. Describe, Explain, Plan and Select: Interactive Planning with Large Language Models Enables Open-World Multi-Task Agents. In *Advances in Neural Information Processing Systems*, volume 36.

Wu, Z.; Wang, Z.; Xu, X.; Lu, J.; and Yan, H. 2023. Embodied Task Planning with Large Language Models. arXiv:2307.01848.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proceedings of the 11th International Conference on Learning Representations*.

Yu, Z.; Yuan, Y.; Xiao, T. Z.; Xia, F. F.; Fu, J.; Zhang, G.; Lin, G.; and Liu, W. 2025. Generating Symbolic World Models via Test-Time Scaling of Large Language Models. *Transactions on Machine Learning Research*.

Zeng, H.; and Li, P. 2026. H-AIM: Orchestrating LLMs, PDDL, and Behavior Trees for Hierarchical Multi-Robot Planning. arXiv:2601.11063.

Zhang, B.; and Luo, P. 2025. OR-LLM-Agent: Automating Modeling and Solving of Operations Research Optimization Problem with Reasoning Large Language Model. arXiv:2503.10009.

Zhang, T.; Zhang, L.; Hou, Z.; Wang, Z.; Gu, Y.; Clark, P.; Callison-Burch, C.; and Tandon, N. 2024. PROC2PDDL: Open-domain Planning Representations from Texts. In *Proceedings of the 2nd Workshop on Natural Language Reasoning and Structured Explanations (@ACL 2024)*, 13–24. Bangkok, Thailand: Association for Computational Linguistics.

Zhang, Y.; Jiang, S.; Li, R.; Tu, J.; Su, Y.; Deng, L.; Guo, X.; Lv, C.; and Lin, J. 2026. DeepPlanning: Benchmarking Long-Horizon Agentic Planning with Verifiable Constraints. arXiv:2601.18137.

Zhao, Z.; Lee, W. S.; and Hsu, D. 2023. Large Language Models as Commonsense Knowledge for Large-Scale Task Planning. In *Advances in Neural Information Processing Systems*, volume 36, 31967–31987.

Zuo, M.; Velez, F. P.; Li, X.; Littman, M.; and Bach, S. 2025. Planetarium: A Rigorous Benchmark for Translating Text to Structured Planning Languages. In Chiruzzo, L.; Ritter, A.; and Wang, L., eds., *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 11223–11240. Albuquerque, New Mexico: Association for Computational Linguistics. ISBN 979-8-89176-189-6.

Appendixes

A Architecture and Implementation Details

This Appendix describes how we implemented the proposed architecture and conducted the experiments. The code will be made publicly available.

A.1 Modeling Tools

The modeling tools are offered at multiple granularities. The declaration interface is intentionally multi-granular. At the finest level, the model can formalize one atomic declaration (individual facts about the problem) with:

- `declare_object`: Allows the LLM to declare an object. It takes as parameters a name and optionally a type. This tool then uses the necessary API from UP to add an object to a UP problem instance.
- `declare_initial_condition`: Allows the LLM to declare an initial condition. It takes as parameters a name and, optionally, a list of object arguments and a value (which can be boolean or numeric). This tool then uses the necessary API from UP to add an initial condition to a UP problem instance.
- `declare_goal`: Allows the LLM to declare a goal. Similarly, it takes as parameters a name and, optionally, a list of object arguments and a value. This tool then uses the necessary API from UP to add a goal to a UP problem instance.
- `declare_metric`: Allows the LLM to declare a metric. It takes as parameters a type of metric and, optionally, a fluent name and a list of object arguments for that fluent. This tool then uses the necessary API from UP to add a metric to a UP problem instance.

Moreover, the model also has access to the `undo` tool, which allows repair of atomic declarations:

- `undo`: Allows the LLM to remove any previous declaration. It takes as parameters a type declaration to remove and its specification. This tool then removes that entity from the UP problem instance by specification matching, subject to consistency checks.

At a medium level, the model can formalize multiple atomic declarations of the same kind with:

- `declare_objects`: Allows the LLM to declare multiple objects, in the same way as its unary version.
- `declare_initial_conditions`: Allows the LLM to declare multiple initial conditions, in the same way as its unary version.
- `declare_goals`: Allows the LLM to declare multiple goals, in the same way as its unary version.
- `declare_metrics`: Allows the LLM to declare multiple metrics, in the same way as its unary version.

At a coarse level, the model may instantiate an entire problem in one call:

- `declare_problem`: Allows the LLM to declare multiple objects, initial conditions, goals and metrics, all at once.

This is not merely an implementation detail. Coarse declarations make one-shot formalization efficient when the natural-language request is clear, whereas fine-grained declarations and undo operations make local repair possible when the request is ambiguous or when the initial formalization is imperfect. The method therefore supports both fast compilation and targeted revision within the same protocol.

Additionally, preliminary experiments showed that some LLM families behave better with tools of some granularity level, whereas others empirically perform better with tools of another granularity level, and can even make use of all granularity levels depending on problem complexity and stage of problem modeling.

A.2 Non-modeling Tools

The tool `get_domain` allows for domain inspection. Let $\mathcal{D}$ denote any planning domain. Parsing the PDDL of $\mathcal{D}$ with Unified Planning yields a symbolic signature

$$\Sigma(\mathcal{D}) = (\mathcal{T}, \mathcal{F}, \mathcal{A}, \mathcal{C}),$$

where $\mathcal{T}$ is the set of types, $\mathcal{F}$ the set of fluents, $\mathcal{A}$ the set of actions, and $\mathcal{C}$ optional annotations extracted from the source description (e.g., comments clarifying modeling conventions). In Unified Planning, predicates and numeric functions are conceptually the same: fluents. PDDL predicates are termed boolean fluents, whereas PDDL functions are termed integer or real fluents; hence, both groups being attributed to $\mathcal{F}$.

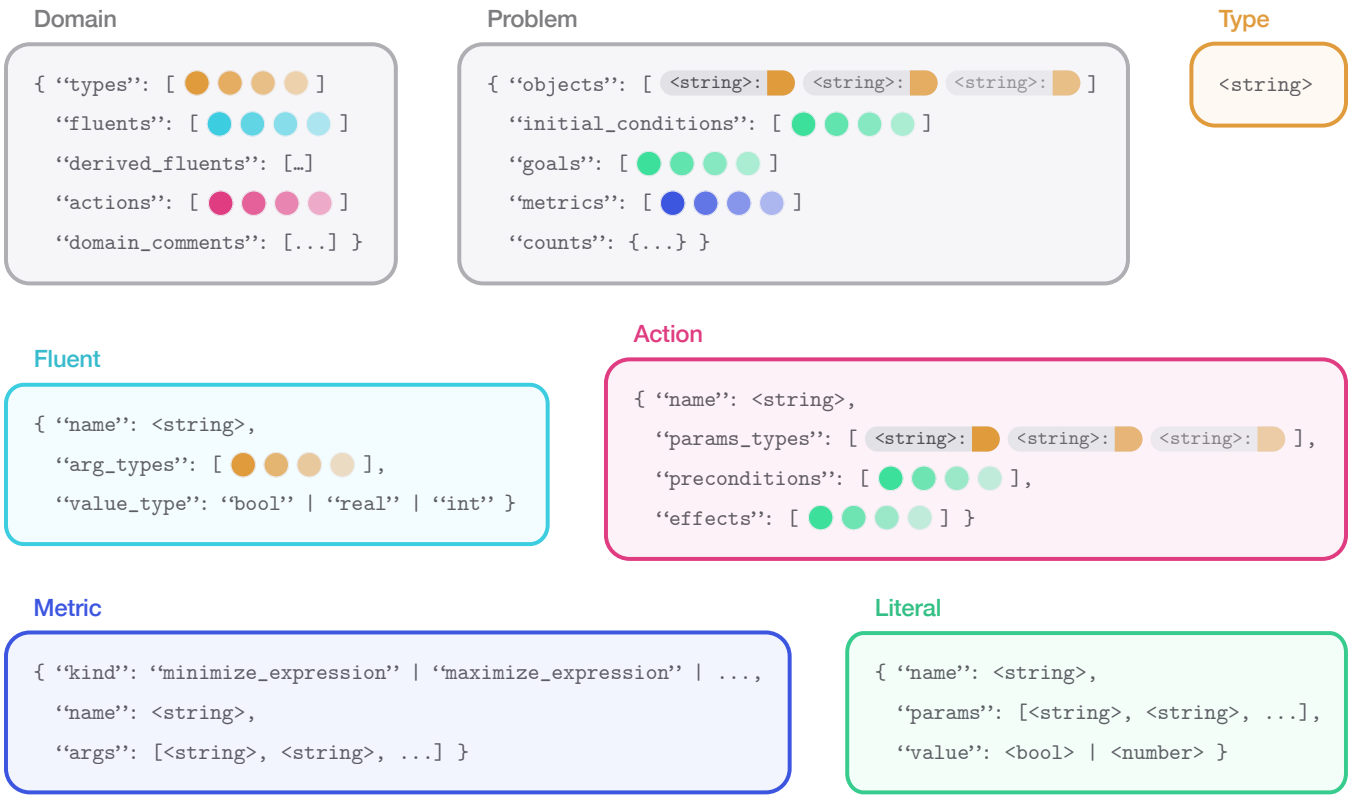

Figure 1: Visual representation of the domain and problem JSON schema, that serve as contracts to dialogue with an LLM about $\mathcal{D}$ and Π, respectively. Formal schema are given in Appendix X.

A.3 Tool Availability

The orchestrator does not expose all tools at all times. Instead, the admissible tool set is staged as

$$\mathcal{U}_t = \begin{cases} \{\texttt{get\_domain}\}, & \text{if } t = 1 \vee \text{domain has never been acquired,} \\ \{\texttt{create\_problem}\}, & \text{if } t = 2 \vee \text{no problem instance exists } (\nexists\, \Pi), \\ \text{all tools}, & \text{if } t \geq 3 \vee \text{a problem instance exists } (\exists\, \Pi). \end{cases}$$

This staged exposure is not merely a prompting convenience. It enforces the intended control flow of the pipeline: the model must first inspect the domain, then create an empty problem, and only then populate or revise that problem. The rationale is to ensure that formalization decisions are grounded in the actual domain signature before any problem content is committed.

A.4 JSON Contracts

Given that the majority of tool-calling standards rely on JSON, and that the LLMs we would be using for evaluation behave well with JSON, we decided to define JSON contracts for symbolic representation throughout the proposed method. One JSON schema for domain information ($\mathcal{D}$) and one JSON schema for problem (Π) can be found in Figure 1. Every time the LLM uses modeling tools it must comply with these schema, and in turn information and feedback in these schema is fed back to the LLM. For example, when calling `declare_goal`, the LLM must formulate the declaration as in the green block of Figure 1. The nomenclature and structure of these schema follow that of the Unified Planning.

A.5 Control Flow

The overall control flow is shown across Figures 2 and 3. Figure 2 focuses on the first part of the interaction, where the model receives the user prompt, inspects the planning domain, creates an empty problem instance, and incrementally formalizes the problem. Figure 3 shows the second part of the interaction, where an attempted solve call triggers verification, possibly leads to revision, and eventually reaches planner execution and a final answer.

At turn $t = 1$, the entry point is the user prompt x, preceded by an instructional system message that constrains the model's inference process. This initial context is illustrated in Figure 2, in the panel labeled *Start*. The system message instructs the

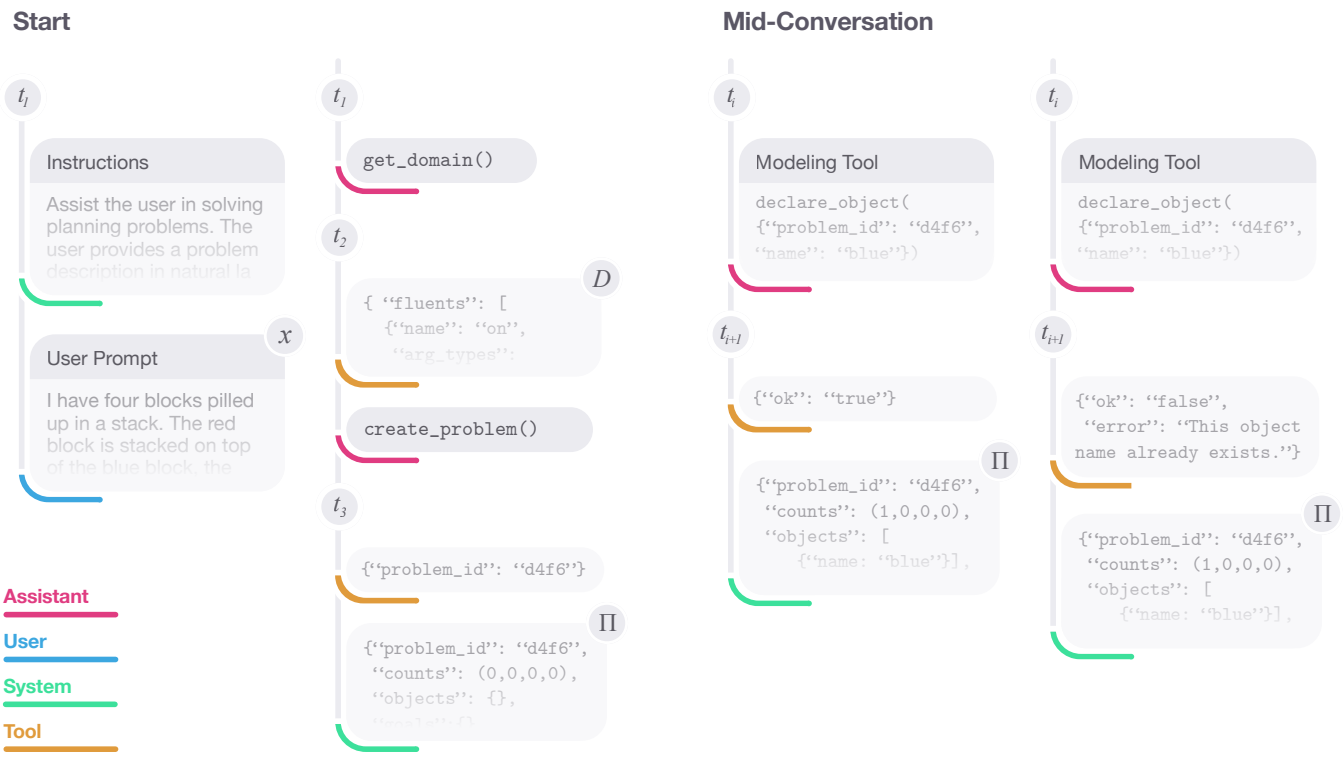

Figure 2: Illustration of a typical conversation (Part I).

model to follow a staged control flow: it must first inspect the domain, then create an empty problem instance in Unified Planning (UP), and only afterward populate that instance with objects, initial facts, goals, and metrics.

A.6 LLM Policy over Tool Calls

Once a problem identifier exists, the model enters the formalization stage, also shown in Figure 2. In this stage, it uses the available modeling tools to modify Π so that it matches the user's description x. These declarations may be issued through a few coarse-grained tool calls or through a sequence of finer-grained calls. The model is encouraged to call multiple tools per turn when appropriate, but it may also proceed one tool call at a time.

Given the user request x, the structured domain summary $\Sigma(\mathcal{D})$, the current symbolic state Π_t, and the currently available tools $\mathcal{U}_t$, the LLM induces a distribution over tool calls,

$$u_t \sim \pi_\theta(\cdot \mid x, \Sigma(\mathcal{D}), \Pi_t, \mathcal{U}_t).$$

When multi-tool calling is enabled, one model turn may produce a sequence

$$u_t = \big((\tau_t^1, a_t^1), \ldots, (\tau_t^{m_t}, a_t^{m_t})\big),$$

with $m_t \geq 1$. When it is disabled, $m_t = 1$. In both cases, each argument structure a_t^i must satisfy the JSON schema associated with tool τ_t^i. This schema validation is performed externally by the orchestrator before the call reaches the planning backend.

A modeling tool call is successful when it satisfies the tool schema, refers to an existing problem identifier, and is accepted by UP as a valid modeling delta. Since the update is performed by the planning backend rather than by the model, successful state transitions are deterministic and backend-defined. Invalid requests leave the problem state unchanged. For example, UP may reject an attempt to declare an object with a type that does not exist in the domain, to declare an initial condition over an undeclared object, to assert a goal using the wrong fluent arity, or to undo a declaration that is not present in Π. In such cases, corrective feedback is returned to the model, which can then revise its tool call.

The backend also enforces semantic invariants throughout this process. Objects must be unique and type-consistent. Literals are matched by fluent name and ordered argument tuple, so re-declaring the same literal updates it rather than duplicating it. Metrics are likewise deduplicated by semantic identity. The `undo` tool removes declarations only when the resulting state remains well-formed. Thus, the model is guided toward semantically meaningful edits not only by the system prompt, but also by the typed and executable semantics of the planning backend.

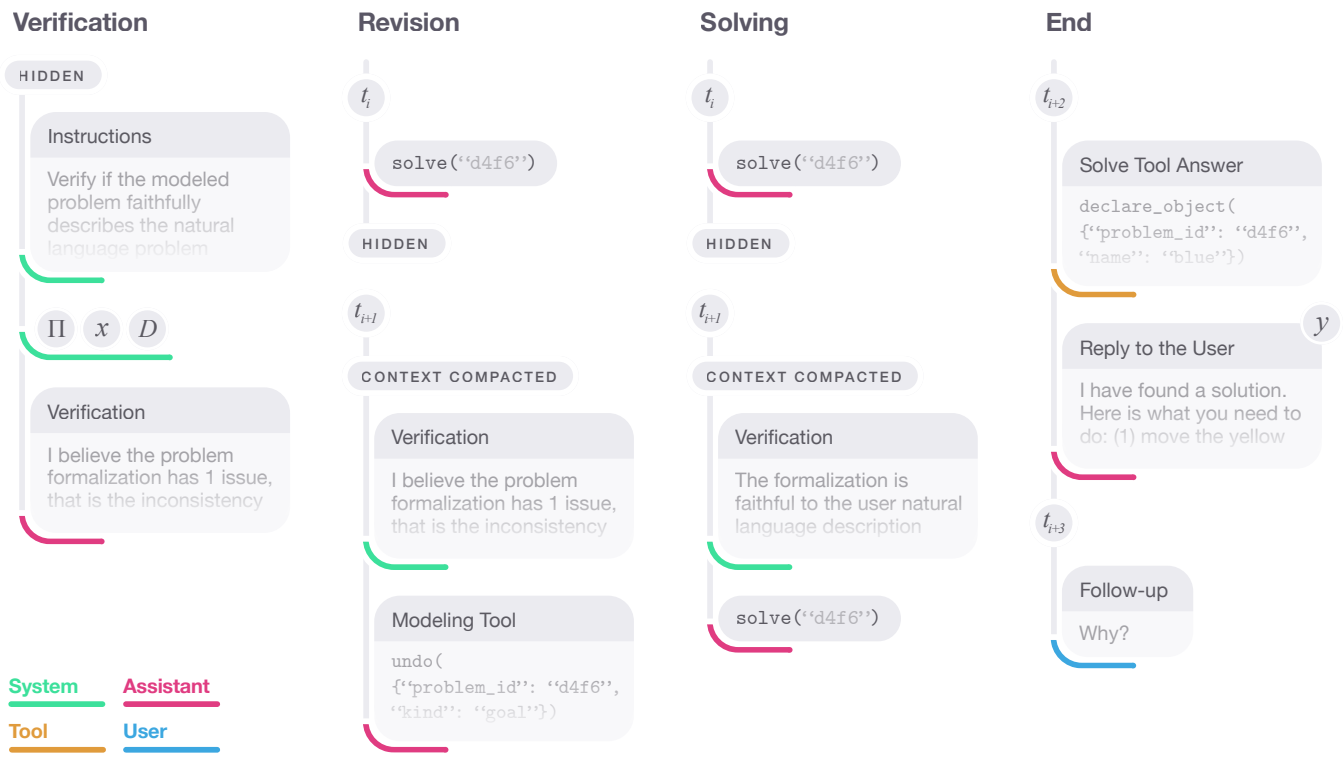

Figure 3: Illustration of a typical conversation (Part II).

A.7 Verification Mechanism

Once the LLM believes that the current state $\sigma_t(\Pi)$ faithfully captures the user's request, it calls `solve`. However, as shown in Figure 3, the first `solve` call does not immediately invoke a planner. Instead, the orchestrator intercepts it and opens an additional self-verification turn. In this verification step, the model is instructed to check whether the formalized problem is complete and faithful to both the original user prompt x and the domain $\mathcal{D}$. The full verification prompt is provided in Appendix X.

This verification mechanism is illustrated in the *Verification* panel of Figure 3. Its purpose is to make the model explicitly inspect the adequacy of its own formalization before planner execution. This is important because, given the generative nature of LLMs, the model may omit objects, misstate initial conditions, introduce goals that do not match the user's intent, or otherwise produce a formally valid but semantically unfaithful planning problem.

The verification output is then returned to the model as feedback, as if the original `solve` call had not yet proceeded to planner execution. If the model agrees that the formalization is adequate, it may call `solve` again. This second consecutive `solve` call opens the *Solving* stage in Figure 3, at which point the orchestrator actually requests UP to invoke a suitable planner on Π and $\mathcal{D}$.

If, however, the verification step identifies a problem, the model may revise Π using the modeling tools. This possibility is shown in the *Revision* branch of Figure 3. The model may add missing objects, correct initial facts, change goals, remove mistaken declarations, or, in the limit, create a fresh problem instance and start over. Whenever the model revises the problem after verification, the verification gate is closed again. Consequently, two consecutive `solve` calls are once more required before the planner is invoked. This design prevents the model from making a repair and immediately bypassing verification.

A.8 Solving

The planner is invoked only through `solve`. The first `solve` request on a state Π_t does not immediately launch the planner. Instead, it returns a verification payload

$$v_t = V(x, \Sigma(\mathcal{D}), \Pi_t),$$

which contains the current formal state, the original request, and the structured domain summary. The LLM must then either revise the state through declaration tools or issue a second consecutive `solve` on the unchanged state. Only this second solve request triggers actual planning. The rationale for this gate is architectural rather than evaluative. It ensures that planner invocation is preceded by explicit inspection of the formalized instance, while still preserving the role separation between formalization and search.

To avoid redundant planner calls, the orchestrator maintains a canonical hash

$$h(\Pi_t) = \mathrm{Hash}(\mathrm{Normalize}(\Pi_t)),$$

where $\mathrm{Normalize}$ removes order dependence in lists such as objects, initial conditions, goals, and metrics. If the same normalized state has already been submitted to the planner, the cached result is returned instead of re-running search.

Terminal outputs are handled differently. As shown in the *End* panel of Figure 3, once UP returns a terminal planner result, that result is fed back to the LLM in the following turn. Only then is the model authorized to answer the user in natural language with a response y. Therefore, the final answer is anchored in a validated planner output rather than in a plan hypothesized directly by the LLM. At this point, the interaction reaches its final turn and the session terminates.

B Prompts Used

B.1 First System Prompt

SYSTEM: Assist the user in solving planning problems. The user provides a problem description in natural language. The task is not to reason to solve the problem. The task is to model the problem in the given planning domain. The modelled problem is then given to a formal planner to solve it. Use the following tools to complete the task.

First, always call 'get_domain' to get information about the current planning domain.

Second, always call 'create_problem' to get a fresh empty problem. This tool may be called more times later on, if needed, to start fresh.

Then:

$<$ description of tools and usage $>$

...

If a modelling tool call is successful, the system will return the current state of the problem instance. If a tool call fails, the state remains unchanged; correct the inputs and retry. On multiple tool calls, if only some tools fail, the state only changes in respect to those that succeeded; correct the inputs and retry for those that failed.

Once the problem instance accurately reflects the user's description, without any missing or extra declarations, and is in accordance with the domain, call 'solve' to send the problem to a formal planner. If successful, it will either return a solution (a plan) or prove that no solution exists (unsolvable). Do not attempt to predict or guess the planner's output. When a plan (or the absence of one) is received, describe it in simple natural language to the user.

USER: $< x >$

B.2 Verification Gate

SYSTEM: The following formal representation of the planning problem might be wrong or incomplete. Verify if the modelled problem faithfully describes the natural language problem description and the static domain. The problem state CANNOT have any extra/wrong declarations, nor missing declarations, so it can be accurately be solved by a planning engine.

In your thinking/reasoning:

* Systematically cross-reference each entry with the natural language description: NO extra and NO missing declarations.

* Make a checklist and mark if ALL entries mentioned in the natural language description have corresponding entries in the formal problem.

* Traverse search in an object-oriented manner: Verify the presence of ALL specific attributes for each 'main' actor.

* Traverse search in a predicate-oriented manner: Verify the completeness of auxiliary predicates.

* Check if the counts in the formal state match the number of atomic entries required by the natural language description.

* Consider that the planning engine will assume a closed-word. That means, boolean fluents don't need to be explicitly declared if their initial value is false; and numeric fluents don't need to be explicitly declared if their initial value is 0.

(Note: The domain is static and correct, so it cannot change. The target of evaluation is only the problem formal representation.)

SYSTEM: $< \mathcal{D} >$

SYSTEM: $< \Pi >$

SYSTEM: $< x >$

C Planning Domains

We use three classical deterministic planning domains to evaluate the method: the Blocksworld, the Logistics, and the Rovers domains. Compared with Blocksworld, Logistics introduces a richer object system, although without typing, and a more explicit geographic structure, making it a useful benchmark for evaluating whether a model can correctly encode object categories, movement constraints, and cross-object relations in a formal representation. The Rovers domain is richer in terms of object types, action complexity, and relational structure compared to Logistics, making it useful for evaluating whether a model can correctly capture complex action constraints, multi-step dependencies, and sophisticated typed domain structures. Additionally, the Rovers domain requires modeling numeric fluents and metrics.

C.1 Blocksworld

This domain consists of a set of blocks arranged and stacked in a certain way on a table, and the goal is to end up with them arranged in a different way. In the variant we use here, an agent has 4 primitive actions to achieve the goal state: `pick-up`, `put-down`, `stack`, and `unstack`. With this domain, we evaluate wether an LLM can formalize multiple binary relations between pairs of objects, that together describe simple spatial structures. The target formalization expects no different types of objects and no metrics associated. Below is the PDDL definition we used (retrieved from (Valmeekam et al. 2023a)):

```
(define (domain blocksworld-4ops)
    (:requirements :strips)
      (:predicates (clear ?x) ; 'clear' means that there is nothing on top of the object
                   (on-table ?x) ; table is not an object
                   (arm-empty) ; arm is not an object
                   (holding ?x)
                   (on ?x ?y)) ; first argument of 'on' is on top of the second argument

      (:action pickup
        :parameters (?ob)
        :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
        :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob))
                     (not (arm-empty))))

      (:action putdown
        :parameters (?ob)
        :precondition (holding ?ob)
        :effect (and (clear ?ob) (arm-empty) (on-table ?ob)
                     (not (holding ?ob))))

      (:action stack
        :parameters (?ob ?underob)
        :precondition (and (clear ?underob) (holding ?ob))
        :effect (and (arm-empty) (clear ?ob) (on ?ob ?underob)
                     (not (clear ?underob)) (not (holding ?ob))))

      (:action unstack
        :parameters (?ob ?underob)
        :precondition (and (on ?ob ?underob) (clear ?ob) (arm-empty))
        :effect (and (holding ?ob) (clear ?underob)
                     (not (on ?ob ?underob)) (not (clear ?ob))
                     (not (arm-empty)))))
```

C.2 Logistics

This domain describes a canonical transportation world, where the goals usually require packages to move from source locations to target locations. Standard actions therefore include loading and unloading packages into trucks and airplanes, driving trucks between city locations, and flying airplanes between city airports. Trucks are restricted to intra-city movement and airplanes to inter-city transport. The target formalization expects categorization of objects (although not using types), and a more complex spatial and hierarchical structure, using more predicates with > 2 arguments, making it a useful benchmark for evaluating whether a model can correctly encode all these. Below is the PDDL definition we used (retrieved from (Valmeekam et al. 2023a)):

```
(define (domain logistics-strips)
  (:requirements :strips)

  ; (:types ) Objects do not have a type, but they are categorized with unary predicates
  ;           which convey their "type".
```

```
(:predicates  (OBJ ?obj) ; Packages are OBJ.
        (TRUCK ?truck) ; Trucks are TRUCK.
              (LOCATION ?loc) ; Locations are LOCATION.
              (AIRPORT ?airport) ; Airports are AIRPORTS and LOCATION.
              (CITY ?city) ; Cities are CITY.
        (AIRPLANE ?airplane) ; Airplanes are AIRPLANE.

  (at ?obj ?loc)
  (in ?obj1 ?obj2)
  (in-city ?obj ?city))

(:action LOAD-TRUCK
  :parameters (?obj ?truck ?loc)
  :precondition (and (OBJ ?obj) (TRUCK ?truck) (LOCATION ?loc)
                     (at ?truck ?loc) (at ?obj ?loc))
  :effect (and (not (at ?obj ?loc)) (in ?obj ?truck)))

(:action LOAD-AIRPLANE
  :parameters (?obj ?airplane ?loc)
  :precondition (and (OBJ ?obj) (AIRPLANE ?airplane) (LOCATION ?loc)
                     (at ?obj ?loc) (at ?airplane ?loc))
  :effect (and (not (at ?obj ?loc)) (in ?obj ?airplane)))

(:action UNLOAD-TRUCK
  :parameters (?obj ?truck ?loc)
  :precondition
   (and (OBJ ?obj) (TRUCK ?truck) (LOCATION ?loc)
        (at ?truck ?loc) (in ?obj ?truck))
  :effect (and (not (in ?obj ?truck)) (at ?obj ?loc)))

(:action UNLOAD-AIRPLANE
  :parameters (?obj ?airplane ?loc)
  :precondition (and (OBJ ?obj) (AIRPLANE ?airplane) (LOCATION ?loc)
                     (in ?obj ?airplane) (at ?airplane ?loc))
  :effect (and (not (in ?obj ?airplane)) (at ?obj ?loc)))

(:action DRIVE-TRUCK
  :parameters (?truck ?loc-from ?loc-to ?city)
  :precondition
   (and (TRUCK ?truck) (LOCATION ?loc-from) (LOCATION ?loc-to) (CITY ?city)
        (at ?truck ?loc-from) (in-city ?loc-from ?city) (in-city ?loc-to ?city))
  :effect (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))

(:action FLY-AIRPLANE
  :parameters (?airplane ?loc-from ?loc-to)
  :precondition (and (AIRPLANE ?airplane) (AIRPORT ?loc-from) (AIRPORT ?loc-to)
                     (at ?airplane ?loc-from))
  :effect (and (not (at ?airplane ?loc-from)) (at ?airplane ?loc-to))))
```

C.3 Rovers

This domain models a team of rovers exploring a planetary environment, featuring rocks and soil targets scattered across multiple waypoints, which they must approach, photograph and collect samples from, in order to achieve the problem goals. The target formalization expects typed objects with properties (camera, storage, etc.) and multiple state variables, which makes this domain useful to evaluate whether a model can formalize richer object systems, and if it organizes its chain-of-thought (CoT) in main actors (rovers) in terms of structural properties and functional dependencies. The domain also introduces temporal-spatial constraints: rovers must be at specific locations, instruments must be calibrated before use, and actions have complex preconditions and effects involving multiple state variables. Additionally, the target formalization includes numeric values to encode distances between waypoints, and metrics to minimize traverse cost, which is useful to evaluate the model ability to formalize numeric values, and large amounts of auxiliary state and environment variables. Here we use the IPC 2005 version, with numerical planning but without the explicit temporal aspect. Below is the PDDL definition we used (retrieved from https://users.cecs.anu.edu.au/~patrik/ipc5/rover_strips_cost.pddl):

```
(define (domain rover)
  (:requirements :strips :typing :fluents)
  (:types rover waypoint store camera mode lander objective)

  (:predicates
    (at ?x - rover ?y - waypoint)
    (at_lander ?x - lander ?y - waypoint)
    (can_traverse ?r - rover ?x - waypoint ?y - waypoint)
    (equipped_for_soil_analysis ?r - rover)
    (equipped_for_rock_analysis ?r - rover)
    (equipped_for_imaging ?r - rover)
    (empty ?s - store)
    (have_rock_analysis ?r - rover ?w - waypoint)
    (have_soil_analysis ?r - rover ?w - waypoint)
    (full ?s - store)
    (calibrated ?c - camera ?r - rover)
    (supports ?c - camera ?m - mode)
    (available ?r - rover)
    (visible ?w - waypoint ?p - waypoint)
    (have_image ?r - rover ?o - objective ?m - mode)
    (communicated_soil_data ?w - waypoint)
    (communicated_rock_data ?w - waypoint)
    (communicated_image_data ?o - objective ?m - mode)
    (at_soil_sample ?w - waypoint)
    (at_rock_sample ?w - waypoint)
    (visible_from ?o - objective ?w - waypoint)
    (store_of ?s - store ?r - rover)
    (calibration_target ?i - camera ?o - objective)
    (on_board ?i - camera ?r - rover)
    (channel_free ?l - lander))

  (:functions
    (traverse_cost ?r - rover ?f - waypoint ?t - waypoint)
    (sum-traverse-cost))


  (:action navigate
    :parameters (?x - rover ?y - waypoint ?z - waypoint)
    :precondition (and (can_traverse ?x ?y ?z) (available ?x) (at ?x ?y)
           (visible ?y ?z))
    :effect (and (not (at ?x ?y)) (at ?x ?z)
     (increase (sum-traverse-cost) (traverse_cost ?x ?y ?z))))

  (:action sample_soil
    :parameters (?x - rover ?s - store ?p - waypoint)
    :precondition (and (at ?x ?p) (at_soil_sample ?p)
           (equipped_for_soil_analysis ?x)
           (store_of ?s ?x) (empty ?s))
    :effect (and (not (empty ?s)) (full ?s) (have_soil_analysis ?x ?p)
     (not (at_soil_sample ?p))))

  (:action sample_rock
    :parameters (?x - rover ?s - store ?p - waypoint)
    :precondition (and (at ?x ?p) (at_rock_sample ?p)
           (equipped_for_rock_analysis ?x)
           (store_of ?s ?x) (empty ?s))
    :effect (and (not (empty ?s)) (full ?s) (have_rock_analysis ?x ?p)
     (not (at_rock_sample ?p))))

  (:action drop
    :parameters (?x - rover ?y - store)
    :precondition (and (store_of ?y ?x) (full ?y))
    :effect (and (not (full ?y)) (empty ?y)))

  (:action calibrate
```

```
66      :parameters (?r - rover ?i - camera ?t - objective ?w - waypoint)
67      :precondition (and (equipped_for_imaging ?r) (calibration_target ?i ?t)
68             (at ?r ?w) (visible_from ?t ?w) (on_board ?i ?r))
69      :effect (calibrated ?i ?r))
70
71    (:action take_image
72      :parameters (?r - rover ?p - waypoint ?o - objective ?i - camera ?m - mode)
73      :precondition (and (calibrated ?i ?r) (on_board ?i ?r)
74             (equipped_for_imaging ?r) (supports ?i ?m)
75             (visible_from ?o ?p) (at ?r ?p))
76      :effect (and (have_image ?r ?o ?m) (not (calibrated ?i ?r))))
77
78    (:action communicate_soil_data
79      :parameters (?r - rover ?l - lander ?p - waypoint ?x - waypoint ?y - waypoint)
80      :precondition (and (at ?r ?x) (at_lander ?l ?y)
81             (have_soil_analysis ?r ?p) (visible ?x ?y)
82             (available ?r) (channel_free ?l))
83      :effect (communicated_soil_data ?p))
84
85    (:action communicate_rock_data
86      :parameters (?r - rover ?l - lander ?p - waypoint ?x - waypoint ?y - waypoint)
87      :precondition (and (at ?r ?x) (at_lander ?l ?y)
88             (have_rock_analysis ?r ?p) (visible ?x ?y)
89             (available ?r) (channel_free ?l))
90      :effect (communicated_rock_data ?p))
91
92    (:action communicate_image_data
93      :parameters (?r - rover ?l - lander ?o - objective ?m - mode
94                   ?x - waypoint ?y - waypoint)
95      :precondition (and (at ?r ?x) (at_lander ?l ?y) (have_image ?r ?o ?m)
96             (visible ?x ?y) (available ?r) (channel_free ?l))
97      :effect (communicated_image_data ?o ?m)))
```

D Dataset Generation

D.1 Ground-truth PDDL Problems

For Blocksworld problems, we used the standard four-operator generator made available in (Valmeekam et al. 2023a). To facilitate comparison with other works, the complexity of a Blocksworld problem is here proportional to the number of blocks. Thus, we generated a corpus of 50 instances per complexity level, from 4 to 20 blocks, amounting to a total of 850 instances. This corresponds to problems with 11 (low complexity) to 63 (high complexity) total declarations.

For Logistics problems, we also used the standard benchmark generator made available in (Valmeekam et al. 2023a). However, we changed the PlanBench sampling criterion from solution difficulty to symbolic description size, given that our task is formalization rather than plan search. Thus, we generated a corpus of 50 instances per complexity level, where complexity increases linearly with the total number of declarations (sum of objects, initial conditions, goals, and metrics in the target formalization), starting from 30 until 204 atomic declarations, throughout 17 levels, yielding a total of 850 Logistics instances.

For Rovers problems, we used the benchmark generator made available in https://users.cecs.anu.edu.au/~patrik/ipc5.html. Once again, we changed the sampling criterion from solution difficulty to formalization size. Thus, we generated a corpus of 50 instances per complexity level, where complexity increases linearly with the total number of declarations, starting from 35 to 374 atomic declarations, throughout 17 levels, yielding a total of 850 Rovers instances.

A total of 2550 problems were generated as PDDL files and we call them ground-truth problems. throughout the generation process, we guaranteed there are no duplicate problems (by hashing), and no degenerate cases whose goals were tautological or already satisfied in the initial state.

This corpus is made available in the Supplementary Material.

D.2 Natural Language Prompts

For Blocksworld and Logistics, we used the natural-language query generator made available in (Valmeekam et al. 2023a). We generated one prompt per problem instance and adapted the results to be zero-shot prompts and without domain information. This means we only kept the text after `STATEMENT` of the prompts produced by the generator. Therefore, the prompts use as x follow the format *"As initial conditions I have that, (...). My goal is to have that (...).*", and contain nothing more.

Blocksworld Prompt Example: "*As initial conditions I have that, the blue block is clear, the hand is empty, the blue block is on top of the orange block, the orange block is on top of the yellow block, the yellow block is on top of the red block and the red block is on the table. My goal is to have that the red block is on top of the orange block and the yellow block is on top of the red block.*"

Logistics Prompt Example: "*As initial conditions I have that, location_0_0 is an airport, airplane_0 is at location_0_0, package_0 is at location_0_1, package_1 is at location_0_0, package_2 is at location_0_2, truck_0 is at location_0_2, location_0_0 is in the city city_0, location_0_1 is in the city city_0 and location_0_2 is in the city city_0. My goal is to have that package_0 is at location_0_1, package_1 is at location_0_1 and package_2 is at location_0_0.*"

Similarly, we generated zero-shot user prompts for the Rovers domain, by adapting this generator. This prompts are still templated but are more natural than the previous ones. We make this adapated generator available in the Supplementary Material.

Rovers Prompt Example: "*As initial conditions I have that, there is a general lander at waypoint 1 with a free communication channel. There is rover 0, available at waypoint 0. Its store is rover 0 store, and it is empty. It is equipped for imaging and rock analysis. It carries camera 0, which supports colour mode and has calibration target objective 0. There is a rock sample at waypoint 1. There are soil samples at waypoints 0 and 1. Rover 0 can traverse from waypoint 0 to waypoint 1 (cost is 111.5) and back (cost is 98.6); and from waypoint 0 to waypoint 2 (cost is 32.6) and back (cost is 77.6). The initial total traverse cost is zero. Waypoint 0 is visible from waypoints 1 and 2. Waypoint 1 is visible from waypoints 0 and 2. Waypoint 2 is visible from waypoints 0 and 1. Objective 0 is visible from waypoints 0 and 1. My goal is to have that image data of objective 0 in colour mode has been communicated and rock data from waypoint 1 has been communicated. I want to minimize the total traverse cost.*"

E Evaluation

E.1 Formalization Accuracy

Our first stage checks whether the predicted problem is *exactly correct up to object renaming*. Concretely, we search for a bijection between predicted and ground-truth objects that preserves object types and makes the initial and goal literals identical after renaming. If such a bijection exists, the instance is counted as an exact match.

To do this, we compute the *best partial alignment* between predicted and ground-truth objects using an exact mixed-integer linear program. Let $x_{ij} \in \{0,1\}$ indicate whether predicted object i is aligned to ground-truth object j, and let $y_{ab} \in \{0,1\}$ indicate whether predicted literal a is matched to ground-truth literal b. We require the object alignment to be a partial injective mapping and disallow alignments across different object types:

$$\sum_j x_{ij} \le 1, \qquad \sum_i x_{ij} \le 1, \qquad x_{ij} = 0 \text{ if } \mathrm{type}(i) \neq \mathrm{type}(j). \tag{1}$$

A literal pair can be matched only if it has the same polarity, predicate, and arity, and if all corresponding arguments are consistent with the chosen object alignment.

We optimize a lexicographic objective that prioritizes matching goal literals, then initial literals, and finally objects:

$$\max \; W_g \sum y^{\text{goal}} + W_i \sum y^{\text{init}} + W_o \sum x, \tag{2}$$

where the weights satisfy $W_g \gg W_i \gg W_o$, making the optimization equivalent to maximizing matched goal literals first, matched initial literals second, and matched objects third. This yields a globally optimal typed correspondence between the two PDDL instances rather than a heuristic local matching.

E.2 Plan Validity

In principle, if a predicted problem is exactly equivalent to the ground-truth problem up to renaming, and the returned plan is genuinely produced by a sound planner for that problem, then the translated plan should also be valid for the ground-truth instance. Formally, let $\hat{P}$ denote the predicted planning problem and let $P^\star$ denote the ground-truth problem. If there exists a bijection ϕ over objects such that $\phi(\hat{P}) = P^\star$, then $\hat{P}$ and $P^\star$ are isomorphic: they have the same initial state, goal condition, and action semantics up to renaming. Under this condition, if a sound planner returns a valid plan $\hat{\pi}$ for $\hat{P}$, then the renamed plan $\phi(\hat{\pi})$ is necessarily a valid plan for $P^\star$. In other words, exact equivalence under renaming implies preservation of plan validity. We nevertheless perform explicit validation because these assumptions may fail in practice: the returned plan may be stale, truncated, produced for an intermediate problem, or corrupted by parsing or post-processing errors. Finally, if a run terminates in error or no plan is produced, we mark plan validity as undefined and do not count that instance as a validated plan.

E.3 Experimental Setup

Experiments to evaluate the method were conducted in a Ubuntu 22.04.5 LTS machine, with one AMD EPYC 9554 64-Core Processor, approximately 540 GB of memory, and one NVIDIA H200 NVL with approximately 141 GB HBM3e memory. All large language models were loaded completely to GPU memory throughout all experiment time. Ollama 0.12.3 was used as model manager.

E.4 Models

Because the interface is standardized through MCP, the approach is modular: the same formalization protocol can be coupled with different LLMs on one side and different planning engines on the other, without changing the core method. We evaluated the method primarily using two large language models: Qwen 3.5 with 122 billion parameters (122B), `Q4_K_M` quantitized, and GPT-OSS with 120 billion parameters (120B), `MXFP4` quantitized. The first has approximately a 256000 token context window, whereas the later has approximately a 128000 token context window. The results presented here were obtained by setting the temperature of both models to 0.05, enabling thinking blocks, and enabling tools use. All other parameters toke their default values, defined by Ollama.

E.5 Unified Planning

Version 1.3.0 of the Unified Planning (UP) Python library was used. Version 0.5.2 of the FastDownward UP extension was used for Blocksworld and Logistics problems. Version 0.1.0 of the ENHSP UP extension was used for Rovers problems.

F Detailed Results

F.1 Inaccurate Formalizations

Table 1: Formalization errors by GPT-OSS

Domain	Instance	Type of Error
Blocksworld	739	init_extra: `(clear white)`
Blocksworld	1789	init_extra: `(clear magenta)`, `(clear orange)`
Blocksworld	3008	init_extra: `(clear cyan)`, `(clear white)`
Logistics	120	init_missing: `(airplane a0)`, `(city c0)`, `(city c1)`, `(location l0-0)`, `(location l0-1)`, `(location l0-2)`, `(location l0-3)`, `(location l0-4)`, `(location l1-0)`, `(location l1-1)`, `(location l1-2)`, `(location l1-3)`, `(location l1-4)`, `(obj p0)`, `(truck t0)`, `(truck t1)`
Logistics	1346	init_missing: `(at p22 l3-3)`; init_extra: `(at p22 l0-0)`
Logistics	1376	init_missing: `(at p23 l2-5)`; init_extra: `(at p23 l1-2)`
Rovers	20	init_extra: `(visible_from objective7 waypoint12)`
Rovers	25	init_extra: `(supports camera0 colour)`, `(supports camera0 low_res)`
Rovers	31	init_missing: `(= (traverse_cost rover1 waypoint6 waypoint5) 101.7)`, `(visible_from objective7 waypoint7)`, `(visible_from objective7 waypoint8)`, `(visible_from objective7 waypoint9)`; init_extra: `(visible_from obj6 wp3)`, `(visible_from obj6 wp4)`
Rovers	53	init_extra: `(supports cam3 low_res)`
Rovers	75	init_extra: `(supports camera1 mode_low)`
Rovers	111	init_extra: `(calibration_target cam3 obj4)`, `(calibration_target cam3 obj5)`, `(supports cam0 low_res)`, `(visible_from obj7 wp8)`
Rovers	147	init_extra: `(calibrated cam0 rover0)`, `(calibrated cam3 rover0)`, `(calibrated cam5 rover0)`
Rovers	232	init_extra: `(visible wp12 wp13)`, `(visible wp12 wp14)`, `(visible wp12 wp15)`
Rovers	501	obj_extra: `camera4`; init_extra: `(calibrated camera0 rover1)`, `(calibrated camera1 rover0)`, `(calibrated camera3 rover0)`, `(calibration_target camera4 obj2)`, `(on_board camera4 rover0)`, `(supports camera0 mode_colour)`, `(supports camera4 mode_colour)`
Rovers	988	init_extra: `(calibration_target cam5 obj2)`, `(supports cam0 high_res)`, `(supports cam5 high_res)`

Table 2: Formalization errors by Qwen 3.5

Domain	Instance	Type of Error
Logistics	1334	init_missing: `(at p18 l2-0)`; init_extra: `(at p18 l0-4)`
Rovers	88	init_missing: `(visible_from objective6 waypoint14)`
Rovers	89	init_extra: `(supports camera0 colour_mode)`, `(supports camera3 colour_mode)`, `(supports camera7 colour_mode)`
Rovers	179	init_missing: `(= (traverse_cost rover0 waypoint0 waypoint15) 48.8)`, `(= (traverse_cost rover0 waypoint15 waypoint0) 99)`, `(can_traverse rover0 waypoint0 waypoint15)`, `(can_traverse rover0 waypoint15 waypoint0)`; init_extra: `(= (traverse_cost rover0 waypoint0 waypoint1) 48.8)`, `(= (traverse_cost rover0 waypoint1 waypoint0) 99)`, `(can_traverse rover0 waypoint0 waypoint1)`, `(can_traverse rover0 waypoint1 waypoint0)`
Rovers	472	init_missing: `(supports camera3 high_res)`
Rovers	560	init_missing: `(can_traverse rover0 waypoint0 waypoint5)`, `(can_traverse rover0 waypoint1 waypoint3)`, `(can_traverse rover0 waypoint3 waypoint1)`, `(can_traverse rover0 waypoint3 waypoint5)`, `(can_traverse rover0 waypoint3 waypoint6)`, `(can_traverse rover0 waypoint3 waypoint8)`, `(can_traverse rover0 waypoint4 waypoint5)`, `(can_traverse rover0 waypoint5 waypoint0)`, `(can_traverse rover0 waypoint5 waypoint3)`, `(can_traverse rover0 waypoint5 waypoint4)`, `(can_traverse rover0 waypoint6 waypoint3)`, `(can_traverse rover0 waypoint6 waypoint7)`, `(can_traverse rover0 waypoint7 waypoint6)`, `(can_traverse rover0 waypoint8 waypoint3)`

F.2 Proposed Method vs. Baseline (McNemar Binomial)

Table 3: Statistical comparison of error rates on GPT-OSS (paired McNemar tests)

Complexity	Δ proposed–baseline (pp)	McNemar Statistic	p-value	q-value (BH)
1	-2.00	3.0000	0.25	0.266
2	-0.67	3.0000	1	1
3	-2.00	7.0000	0.453	0.467
4	-2.00	3.0000	0.25	0.266
5	-3.33	5.0000	0.0625	0.0787
6	-4.67	9.0000	0.0391	0.0531
7	-4.00	6.0000	0.0312	0.0443
8	-4.00	8.0000	0.0703	0.0824
9	-5.33	10.0000	0.0215	0.0348
10	-4.00	8.0000	0.0703	0.0824
11	-5.33	8.0000	0.00781	0.0156
12	-7.33	11.0000	0.000977	0.00277
13	-6.67	12.0000	0.00635	0.0144
14	-8.00	16.0000	0.00418	0.0102
15	-4.67	7.0000	0.0156	0.028
16	-8.00	12.0000	0.000488	0.00208
17	-12.00	20.0000	4.01×10^{-5}	0.000681

F.3 Generalization across Domains

To assess whether accuracy significantly varied by domain, we tested the association between domain and error occurrence using a Fisher–Freeman–Halton exact test on a 3×2 contingency table for each accuracy measure. Separately, for each model, each table crossed domain with outcome – accurate vs. inaccurate – using 850 examples per domain. For the formalization accuracy, the association between domain and error occurrence was statistically insignificant for GPT-OSS ($p = .123$) and Qwen 3.5 ($p = .053$). For the plan validity, the association between domain and error occurrence was statistically insignificant for GPT-OSS ($p = .070$) and Qwen 3.5 ($p = 1.0$).

For the formalization accuracy of the baseline experiments, a significant $p = 0.0001$ for both models was attained on the same Fisher–Freeman–Halton exact test (Tables 5 and 6). Given that the null hypothesis is that the domain is independent of

Table 4: Statistical comparison of error rates on Qwen 3.5 (paired McNemar tests)

Complexity	Δ proposed–baseline (pp)	McNemar Statistic	p-value	q-value (BH)
1	-5.33	8.0000	0.00781	0.0156
2	-7.33	11.0000	0.000977	0.00277
3	-4.00	6.0000	0.0312	0.0443
4	-7.33	11.0000	0.000977	0.00277
5	-8.67	13.0000	0.000244	0.00138
6	-3.33	5.0000	0.0625	0.0787
7	-3.33	9.0000	0.18	0.204
8	-7.33	11.0000	0.000977	0.00277
9	-6.00	9.0000	0.00391	0.0102
10	-4.00	6.0000	0.0312	0.0443
11	-8.67	13.0000	0.000244	0.00138
12	-12.00	20.0000	4.01×10^{-5}	0.000681
13	-8.00	12.0000	0.000488	0.00208
14	-4.67	7.0000	0.0156	0.028
15	-10.00	15.0000	6.1×10^{-5}	0.000692
16	-9.33	14.0000	0.000122	0.00104
17	-6.67	16.0000	0.0213	0.0348

formalization errors, we can conclude at $\alpha = .05$ that there is a significant dependence.

Table 5: Formalization error rates of Baseline method (GPT-OSS)

Domain	No Error	Error	Total	Error Rate
Blocksworld	828	22	850	0.0259
Logistics	821	29	850	0.0341
Rovers	760	90	850	0.1059

Table 6: Formalization error rates of Baseline method (Qwen 3.5)

Domain	No Error	Error	Total	Error Rate
Blocksworld	818	32	850	0.0376
Logistics	823	27	850	0.0318
Rovers	729	121	850	0.1424

F.4 Robustness to Problem Complexity

To test whether accuracy depended on problem complexity, we fitted binomial logistic regression models for each domain, model, and accuracy metric, using the number of accurate outputs out of 50 trials as the response and complexity level as an ordinal predictor. Table 7 shows the results. For Blocksworld with Qwen, both formalization accuracy and validity accuracy did not vary across all complexity levels. With OSS, complexity was not significantly associated with formalization accuracy, nor with validity accuracy. For Logistics with Qwen, formalization accuracy and validity accuracy were near-perfect and differed only at complexity level 17, leading to quasi-complete separation and unreliable regression coefficient estimation. With GPT-OSS, complexity was not significantly associated with formalization accuracy. For plan validity, the model exhibited quasi-complete separation, so the corresponding complexity coefficient was not reliably estimable. Similarly, for Rovers with Qwen, complexity was not significantly associated with formalization accuracy, nor with validity accuracy. For Rovers with OSS, complexity was not significantly associated with formalization accuracy, nor with validity accuracy.

Table 7: GLM regression against complexity, grouped by domain and model.

Domain	Model	Metric	Slope	p-value	CI 95%
Blocksworld	Qwen	Formalization Accurate	0	1	n.a.
		Valid Accurate	0	1	n.a.
	OSS	Formalization Accurate	0.118885	0.360484	$[-0.136,\ 0.374]$
		Valid Accurate	0.042121	0.773508	$[-0.245,\ 0.329]$
Logistics	Qwen	Formalization Accurate	-20.420038	¿ 0.999	n.a.
		Valid Accurate	-20.420038	¿ 0.999	n.a.
	OSS	Formalization Accurate	-0.136257	0.308175	$[-0.398,\ 0.126]$
		Valid Accurate	-21.130898	¿ 0.998	$[-32000,\ 31900]$
Rovers	OSS	Formalization Accurate	-0.149600	0.0589976	$[-0.305,\ 0.006]$
		Valid Accurate	-0.137138	0.0951951	$[-0.298,\ 0.024]$
	Qwen	Formalization Accurate	-0.136608	0.187833	$[-0.340,\ 0.067]$
		Valid Accurate	-20.420038	¿ 0.999	n.a.

F.5 Resource Consumption depending on modeling path

Figure 4 of the main text shows the output tokens and session total time across complexity levels, grouped by domain and model, where it is highlighted two different pathways: when `declare_problem` was the first tool call and when it was not.

For each model-domain pair and for each outcome (total tokens, total time), we compared instances that used `declare_problem` vs those that did not using a two-sided stratified permutation test (blocked randomization test), stratified by complexity level (17 levels, domain-specific definitions). Because outcomes were right-skewed, tests were performed on the log scale. Let (y_{is}) be the outcome in stratum (s); the test statistic was the weighted average stratum effect

$$T = \frac{\sum_s w_s \left(\bar{\log} ys, \texttt{declare\_problem} - \bar{\log} ys, \text{no-}\texttt{declare\_problem}\right)}{\sum_s w_s},$$

with $(w\_s = n\_s)$ (stratum sample size). Under the null, group labels were permuted within each stratum (preserving stratum counts) for 5,000 permutations; p-values were computed as $((1 + \#|T^*| \geq |T_{\text{obs}}|)/(B + 1))$. Effect size was reported as a multiplicative change $(\exp(T) - 1)$ (percent difference), and we also report raw- scale mean differences for interpretability." The Table below shows the results.

Table 8: Two-sided stratified permutation test by complexity level on weighted mean log differences between triangles and circles.

Model	Domain	Metric	Mean Diff.	Log Mean Diff.	Stratified Ratio	p-value
OSS	Blocksworld	tokens	0.817125	-0.234534	0.790940	0.00019996
OSS	Blocksworld	time	0.932346	-0.0966207	0.907900	0.00019996
OSS	Logistics	tokens	0.933280	-0.0745427	0.928168	0.00019996
OSS	Logistics	time	1.100060	0.0873110	1.091240	0.00019996
OSS	Rovers	tokens	0.817242	-0.262866	0.768845	0.00019996
OSS	Rovers	time	0.934681	-0.0559985	0.945541	0.0421916
Qwen	Blocksworld	tokens	NaN	NaN	NaN	NaN
Qwen	Blocksworld	time	NaN	NaN	NaN	NaN
Qwen	Logistics	tokens	1.183630	0.164805	1.179160	0.00019996
Qwen	Logistics	time	1.353570	0.286142	1.331280	0.00019996
Qwen	Rovers	tokens	1.015110	-0.477656	0.620235	0.00019996
Qwen	Rovers	time	1.072530	-0.172796	0.841309	0.0207958

F.6 Modeling Pathways Effect

Table 9: Pathway outcomes for GPT-OSS on Blocksworld problems

Pathway	N	Formalization Accuracy	Plan Validity ([CI])
Calls `declare_problem` first and only	474	0.998 ([0.988, 1.000])	1.000 ([0.992, 1.000])
Calls `declare_problem` and then revises	63	1.000 ([0.943, 1.000])	1.000 ([0.943, 1.000])
Calls `declare_problem` anywhere	537	0.998 ([0.990, 1.000])	1.000 ([0.993, 1.000])
Never calls `declare_problem`	313	0.994 ([0.977, 0.998])	0.994 ([0.977, 0.998])
Calls `undo` at least once	87	1.000 ([0.958, 1.000])	1.000 ([0.958, 1.000])
Never calls `undo`	763	0.996 ([0.989, 0.999])	0.997 ([0.990, 0.999])

Table 10: Pathway outcomes for oss on logistics

Pathway	N	Formalization Accuracy	Plan Validity ([CI])
Calls `declare_problem` first and only	185	0.995 ([0.970, 0.999])	0.995 ([0.970, 0.999])
Calls `declare_problem` and then revises	8	1.000 ([0.676, 1.000])	1.000 ([0.676, 1.000])
Calls `declare_problem` anywhere	315	0.997 ([0.982, 0.999])	0.997 ([0.982, 0.999])
Never calls `declare_problem`	535	0.996 ([0.986, 0.999])	0.998 ([0.989, 1.000])
Calls `undo` at least once	8	1.000 ([0.676, 1.000])	1.000 ([0.676, 1.000])
Never calls `undo`	842	0.996 ([0.990, 0.999])	0.998 ([0.991, 0.999])

Table 11: Pathway outcomes for oss on rovers

Pathway	N	Formalization Accuracy	Plan Validity ([CI])
Calls `declare_problem` first and only	139	0.993 ([0.960, 0.999])	0.993 ([0.960, 0.999])
Calls `declare_problem` and then revises	162	0.994 ([0.966, 0.999])	0.994 ([0.966, 0.999])
Calls `declare_problem` anywhere	385	0.992 ([0.977, 0.997])	0.992 ([0.977, 0.997])
Never calls `declare_problem`	465	0.987 ([0.972, 0.994])	0.989 ([0.975, 0.995])
Calls `undo` at least once	88	0.943 ([0.874, 0.975])	0.943 ([0.874, 0.975])
Never calls `undo`	762	0.995 ([0.987, 0.998])	0.996 ([0.988, 0.999])

Table 12: Pairwise pathway contrasts for OSS

Contrast	Formalization difference	Plan Validity difference
Calls `declare_problem` first and only vs Never calls `declare_problem`	$+0.386$ pp ($p = 0.392$)	$+0.359$ pp ($p = 0.336$)
Calls `declare_problem` anywhere vs Never calls `declare_problem`	$+0.357$ pp ($p = 0.304$)	$+0.286$ pp ($p = 0.389$)
Calls `undo` at least once vs Never calls `undo`	-2.310 pp ($p = 0.003$)	-2.437 pp ($p = 0.001$)

Table 13: Pathway outcomes for qwen on blocksworld

Pathway	N	Formalization Accuracy	Plan Validity ([CI])
Calls `declare_problem` first and only	0	–	–
Calls `declare_problem` and then revises	0	–	–
Calls `declare_problem` anywhere	0	–	–
Never calls `declare_problem`	850	1.000 ([0.996, 1.000])	1.000 ([0.996, 1.000])
Calls `undo` at least once	12	1.000 ([0.758, 1.000])	1.000 ([0.758, 1.000])
Never calls `undo`	838	1.000 ([0.995, 1.000])	1.000 ([0.995, 1.000])

Table 14: Pathway outcomes for qwen on logistics

Pathway	N	Formalization Accuracy	Plan Validity ([CI])
Calls `declare_problem` first and only	1	1.000 ([0.207, 1.000])	1.000 ([0.207, 1.000])
Calls `declare_problem` and then revises	1	1.000 ([0.207, 1.000])	1.000 ([0.207, 1.000])
Calls `declare_problem` anywhere	201	0.995 ([0.972, 0.999])	0.995 ([0.972, 0.999])
Never calls `declare_problem`	649	1.000 ([0.994, 1.000])	1.000 ([0.994, 1.000])
Calls `undo` at least once	9	1.000 ([0.701, 1.000])	1.000 ([0.701, 1.000])
Never calls `undo`	841	0.999 ([0.993, 1.000])	0.999 ([0.993, 1.000])

Table 15: Pathway outcomes for qwen on rovers

Pathway	N	Formalization Accuracy	Plan Validity ([CI])
Calls `declare_problem` first and only	14	1.000 ([0.785, 1.000])	1.000 ([0.785, 1.000])
Calls `declare_problem` and then revises	0	–	–
Calls `declare_problem` anywhere	160	0.988 ([0.956, 0.997])	1.000 ([0.977, 1.000])
Never calls `declare_problem`	690	0.996 ([0.987, 0.999])	0.999 ([0.992, 1.000])
Calls `undo` at least once	8	1.000 ([0.676, 1.000])	1.000 ([0.676, 1.000])
Never calls `undo`	842	0.994 ([0.986, 0.997])	0.999 ([0.993, 1.000])

Table 16: Pairwise pathway contrasts for Qwen

Contrast	Formalization difference	Plan Validity difference
Calls `declare_problem` first and only vs Never calls `declare_problem`	$+0.137$ pp ($p = 1.000$)	$+0.046$ pp ($p = 1.000$)
Calls `declare_problem` anywhere vs Never calls `declare_problem`	-0.694 pp ($p = 0.040$)	-0.231 pp ($p = 0.263$)
Calls `undo` at least once vs Never calls `undo`	$+0.238$ pp ($p = 1.000$)	$+0.079$ pp ($p = 1.000$)

Table 17: Adjusted logistic regression predicting plan validity from model, domain, pathway indicators, and problem complexity. Odds ratios are shown with 95% confidence intervals; standard errors and p-values are reported for each coefficient. Model, domain, and `declare_problem` usage do not show clear statistically significant effects after adjustment.

Term	Odds Ratio	Std. Error	p-value	N
Intercept	1.442×10^{13} ([0.00, ∞])	1.265×10^{7}	1.000	5100
C(model)[T.qwen]	3.67 ([0.73, 18.51])	0.824	0.115	5100
C(domain)[T.logistics]	0.48 ([0.07, 3.12])	0.957	0.442	5100
C(domain)[T.rovers]	0.24 ([0.05, 1.29])	0.829	0.096	5100
Calls `declare_problem` anywhere	1.56 ([0.39, 6.22])	0.707	0.531	5100
Calls `declare_problem` first and only	0.49 ([0.06, 3.87])	1.054	0.499	5100
Calls `undo` at least once	0.12 ([0.03, 0.46])	0.692	0.002	5100

F.7 Unified Planning Feedback Effect

Table 18: UP/Planner feedback recovery summary across trajectories, grouped by whether they received modeling feedback or planner feedback, and if those prevented invalid updates or inaccurate updates. Invalid updates are those that are structurally invalid or non-parseable, as per UP modeling rules. Inaccurate updates are those that would have introduced an inconsistency with ground-truth – UP does not access the ground-truth, but we observe post-run what would have been an inaccurate update. For each group, we report how many instances exhibited that event at least once, and what share of that later recovered to a accurate formalization in the same trajectory. Categories are not mutually exclusive, so row counts can overlap.

Backend Feedback	Instances	Recovered after	Formalization accuracy
At least one rejected modeling tool call	997	0.318	0.991
At least one rejected solve tool call	16	0.812	0.750
Feedback prevented invalid update	262	0.935	0.981
Feedback prevented inaccurate update	659	0.114	0.995

Model	Domain	Modeling feedback events	Prevented invalid updates	Prevented inaccurate updates	Planner feedback events
oss	blocksworld	270	0.689	0.000	0
	logistics	199	0.010	0.985	3
	rovers	586	0.229	0.290	58
qwen	blocksworld	82	1.000	0.000	0
	logistics	327	0.000	1.000	2
	rovers	84	1.000	0.000	1

Table 19: Feedback type summary for model = oss

Domain	Feedback type	N Events	N Instances
blocksworld	Duplicate declaration	83	57
blocksworld	Invalid undo	185	74
blocksworld	Unknown object	1	1
blocksworld	Others	1	1
logistics	Duplicate declaration	196	191
logistics	Invalid undo	1	1
logistics	Planner non-terminal	3	3
logistics	Unknown object	1	1
logistics	Others	1	1
rovers	Duplicate declaration	408	228
rovers	Invalid undo	56	53
rovers	Planner non-terminal	58	10
rovers	Unknown fluent/predicate	12	10
rovers	Unknown object	41	40
rovers	Wrong arity	11	10
rovers	Wrong object type	5	5
rovers	Others	53	51

Table 20: Feedback type summary for model = qwen

Domain	Feedback type	N Events	N Instances
blocksworld	Invalid undo	82	11
logistics	Duplicate declaration	327	321
logistics	Planner non-terminal	2	2
rovers	Duplicate declaration	1	1
rovers	Invalid undo	2	2
rovers	Unknown object	44	42
rovers	Wrong arity	10	10
rovers	Wrong object type	2	1
rovers	Others	26	26

Table 21: Examples of Preventing Inaccuracies

Instance	Model	Domain	Tool call attempted	Short feedback message
Unknown Object				
2782	oss	blocksworld	`declare_goals | {"literals": [{"args": ["red", "blue", "?"], "name": "on", "value": true}]}`	Unknown object '?'; add it first.
747	oss	logistics	`declare_initial_conditions | {"literals": [{"args": ["package_0"], "name": "OBJ", "value": true}, {"args": ["package_1"], "name": "OBJ", "value": true}, ...]}`	Unknown object 'package_0'; add it first.
Wrong Arity				
126	oss	rovers	`declare_initial_conditions | {"literals": [{"args": ["waypoint2"], "name": "at_rock_sample", "value": true}, ...]}`	'visible' takes 2 argument(s).
57	oss	rovers	`declare_problem | {"goals": [{"args": ["objective0", "low_res"], "name": "communicated_image_data", "value": true}, ...], "init": [...]}`	'visible' takes 2 argument(s).
Wrong Object Type				
111	oss	rovers	`declare_objects | {"objects": [{"name": "rover0"}, {"name": "rover1"}, {"name": "rover2"}, ...]}`	Missing object type for: 'rover0', 'rover1', 'rover2', 'rover3', 'wp0', 'wp1', …
1105	oss	rovers	`declare_initial_conditions | {"literals": [{"args": ["rover0", "waypoint8"], "name": "at", "value": true}, ...]}`	Argument type mismatch for at at position 1: object 'lander0' has type lander; expected rover.
Invalid Undo				
10	oss	blocksworld	`undo | {"declaration": {"kind": "object", "name": "orange"}, "problem_id": "blocksworld_problem_1"}`	Object cannot be removed, because it is referenced by existing declarations.
10	oss	blocksworld	`undo | {"declaration": {"kind": "object", "name": "orange"}, "problem_id": "blocksworld_problem_1"}`	Object cannot be removed, because it is referenced by existing declarations.
Prevented Inaccurate Update				
1000	oss	logistics	`declare_objects | {"objects": [{"name": "location_0_0"}, {"name": "location_0_1"}, {"name": "location_0_2"}, ...]}`	These objects already exist in the current state of logistics_problem_1.
Planner Diagnostic Recovery				
1280	oss	logistics	`solve | {"problem_id": ...}`	Planner Result: Unsolvable Incompletely. Most likely the problem is not well formalized.
178	oss	logistics	`solve | {"problem_id": ...}`	Planner Result: Unsolvable Incompletely. Most likely the problem is not well formalized.

F.8 Verification Mechanism Effect

Table 22: Ratio of instances per outcome of the verification mechanism, grouped by model and domain.

Transition	Model	Domain	Instances
Problem state (Π) gets repaired, from inaccurate to accurate	oss	blocksworld	1.65%
		logistics	0.59%
		rovers	8.71%
	qwen	blocksworld	0.12%
		logistics	0.47%
		rovers	7.29%
Problem state (Π) stays preserved, from accurate stays accurate	oss	blocksworld	98.00%
		logistics	99.06%
		rovers	90.35%
	qwen	blocksworld	99.88%
		logistics	99.41%
		rovers	92.12%
Problem state (Π) gets worse, from accurate to inaccurate	oss	blocksworld	0.12%
		logistics	0.00%
		rovers	0.35%
	qwen	blocksworld	0.00%
		logistics	0.00%
		rovers	0.00%
Problem state (Π) is not repaired, from inaccurate stays inaccurate	oss	blocksworld	0.24%
		logistics	0.35%
		rovers	0.59%
	qwen	blocksworld	0.00%
		logistics	0.12%
		rovers	0.59%

Table 23: Ratio of instances per post-verification scenario, grouped by model and domain.

Verification Action	Model	Domain	Instances
Proceeded without revision (called `solve` again)	oss	blocksworld	88.82%
		logistics	98.24%
		rovers	89.18%
	qwen	blocksworld	99.18%
		logistics	98.59%
		rovers	93.29%
Revised (single modeling tool call)	oss	blocksworld	2.94%
		logistics	0.71%
		rovers	8.00%
	qwen	blocksworld	0.24%
		logistics	0.82%
		rovers	3.18%
Revised (multiple modeling tool calls)	oss	blocksworld	7.76%
		logistics	0.71%
		rovers	2.35%
	qwen	blocksworld	0.47%
		logistics	0.59%
		rovers	3.53%
Created a new problem and started over	oss	blocksworld	0.47%
		logistics	0.35%
		rovers	0.47%
	qwen	blocksworld	0.12%
		logistics	0.00%
		rovers	0.00%
Failed or non-terminal outcome	oss	blocksworld	0.00%
		logistics	0.00%
		rovers	0.00%
	qwen	blocksworld	0.00%
		logistics	0.00%
		rovers	0.00%

Table 24: types of revisions, when they toke place, and their outomes.

Kind of Revision	Inaccurate before first verification	Accurate after first verification	Accurate final formalization	Inaccurate final formalization
Removed a goal	1	1	1	0
Modified a goal	5	4	5	0
Removed an initial condition	33	30	33	0
Added an initial condition	105	101	104	1
Removed an object	1	1	1	0
Multiple issues	5	4	5	0
None	26	0	11	15
Total	176	141	160	16