LLM+AL: Bridging Large Language Models and Action Languages for Complex Reasoning About Actions

Adam Ishay[1], **Joohyung Lee**[1,2]
[1]Arizona State University, AZ, USA
[2]Samsung Research, S. Korea
{aishay, joolee}@asu.edu

Abstract

Large Language Models (LLMs) have made significant strides in various intelligent tasks but still struggle with complex action reasoning tasks that require systematic search. To address this limitation, we propose a method that bridges the natural language understanding capabilities of LLMs with the symbolic reasoning strengths of action languages. Our approach, termed LLM+AL, leverages the LLM's strengths in semantic parsing and commonsense knowledge generation alongside the action language's proficiency in automated reasoning based on encoded knowledge. We compare LLM+AL against state-of-the-art LLMs, including CHATGPT-4, CLAUDE 3 OPUS, GEMINI ULTRA 1.0, and O1-PREVIEW, using benchmarks for complex reasoning about actions. Our findings indicate that, although all methods exhibit errors, LLM+AL, with relatively minimal human corrections, consistently leads to correct answers, whereas standalone LLMs fail to improve even with human feedback. LLM+AL also contributes to automated generation of action languages.

1 Introduction

Large Language Models (LLMs) have made significant strides in various intelligent tasks (Brohan et al. 2023; Kojima et al. 2022; Huang et al. 2023; Zeng et al. 2022; Yao et al. 2024; Besta et al. 2024), yet they often struggle with complex reasoning about actions, particularly in problems that demand systematic search. An emerging alternative is to use an LLM as a semantic parser to convert natural language into symbolic representations, such as Python programs (Gao et al. 2023; Nye et al. 2021; Olausson et al. 2023), Planning Domain Definition Language (PDDL) (Liu et al. 2023; Guan et al. 2023; Xie et al. 2023), or logic programs (Ishay, Yang, and Lee 2023). These symbolic representations are then processed by dedicated symbolic reasoners.

However, these methods have limitations. As demonstrated in this paper, for complex reasoning tasks, LLMs almost always fail to generate Python programs for searching for solutions, except in cases where the problem is a typical search task that LLMs may have memorized from the training corpus. Even in such instances, when small variations are introduced, LLMs struggle to adapt to the changes.

Using LLMs to generate PDDL could help address a broader range of action reasoning problems. However, PDDL is primarily designed as a standard language for task planning and is not well-suited for expressing more general forms of knowledge about actions, such as state constraints, indirect effects, default reasoning, and recursive definitions (Giunchiglia et al. 2004).

This paper presents a novel method that bridges the natural language understanding capabilities of LLMs with the symbolic reasoning strengths of action languages (Gelfond and Lifschitz 1998; Giunchiglia et al. 2004). Our approach, termed "LLM+AL," leverages the LLM's strengths in semantic parsing and commonsense knowledge generation alongside the action language's proficiency in automated reasoning about actions based on encoded knowledge.

Action languages are particularly well-suited for this purpose. They have an intuitive, natural language like syntax, feature formal semantics, and are supported by efficient computational tools. Action languages are designed for more knowledge-intensive reasoning than PDDL, encompassing not only task planning problems but also temporal prediction problems, which involve predicting what would happen if a sequence of actions is executed, and postdiction problems, where one infers the initial state given the current state and a past sequence of actions. Even when focused solely on task planning problems, action languages offer greater expressivity, such as representing indirect effects to address the ramification problem (e.g., the banana's location is determined by the monkey's location if the monkey is holding it; thus, any action that moves the monkey will indirectly affect the banana's location), and defaults (e.g., by default, a pendulum swings back and forth unless it is being held). In particular, this paper leverages one of the latest members in this family, the action language $\mathcal{BC}$+ (Babb and Lee 2015, 2020) due to its simplicity and expressivity, as well as the availability of the efficient $\mathcal{BC}+$ reasoner called CPLUS2ASP (Babb and Lee 2013).

The LLM+AL pipeline leverages an LLM effectively across multiple stages, each serving a different purpose. First, given a reasoning problem in natural language, we use an LLM to generate a program signature and extract commonsense and domain-specific knowledge. Next, the LLM is tasked with converting this knowledge into $\mathcal{BC}+$ rules, guided by a prompt that details $\mathcal{BC}+$ syntax and semantics

and is supplemented with several translation examples. After generating a complete $\mathcal{BC}+$ program, the LLM is tasked with doing a series of revisions if necessary, based on the output of the $\mathcal{BC}+$ reasoner. This pipeline facilitates the generation of correct $\mathcal{BC}+$ programs (with plans) or mostly correct $\mathcal{BC}+$ programs to users. Our code is available on GitHub https://github.com/azreasoners/llm-al, and the appendix can be found in the long version of the paper (Ishay and Lee 2025).

Our findings indicate that LLM+AL, even with basic descriptions of $\mathcal{BC}+$ and a few translation examples, is surprisingly adept at translating English into $\mathcal{BC}+$ rules and extracting relevant knowledge, thanks to the rich semantic understanding that LLMs acquired during pre-training. Despite occasional errors in knowledge extraction and rule generation—a challenge even for human experts—our approach demonstrates proficiency comparable to that of human experts, requiring only a few manual corrections to modify the erroneous $\mathcal{BC}+$ programs and produce correct solutions.

Our focus is on problems that require deep rather than shallow reasoning, often involving systematic search, which LLMs typically struggle with. We evaluated our method using a benchmark proposed by McCarthy (1998), which serves as a "Drosophila" for assessing the elaboration tolerance of human-level AI—highlighting the importance of an AI's ability to represent and reason about new phenomena or altered circumstances. The problem set includes several elaborations of the well-known Missionaries and Cannibals Puzzle (MCP), such as "what if only one missionary and one cannibal can row?" and "what if there are four missionaries and four cannibals instead?" We found that the most capable LLMs today, such as CHATGPT-4 (OpenAI 2024a), CLAUDE 3 OPUS (Anthropic 2024), GEMINI 1.0 ULTRA (Gemini Team 2023), generally fail to produce correct solutions for this benchmark, even after multiple iterations with human feedback pointing out errors in the answers. We also conducted experiments with O1-PREVIEW (OpenAI 2024b), a novel type of LLM that utilizes test-time compute to handle more complex reasoning tasks. While it outperformed other LLMs on the benchmarks, it still exhibited notable limitations.

In contrast, our approach, with relatively few human corrections, consistently leads to correct answers. This suggests that while current LLMs possess strong natural language understanding, they lack the systematic reasoning capability required to adapt effectively to new or altered scenarios. By integrating LLMs with action languages, our method demonstrates the potential for achieving a more robust and adaptable AI system.

2 Preliminaries

2.1 LLMs for Planning

Several works have proposed applying LLMs to planning tasks (Huang et al. 2022; Brohan et al. 2023; Huang et al. 2023; Singh et al. 2023; Yao et al. 2023). For instance, SayCan (Brohan et al. 2023) combines high-level actions with value functions, grounding LLMs in an environment. Inner Monologue (Huang et al. 2023) integrates environmental feedback, including human feedback, into its pipeline, thereby enhancing robustness against agent errors. However, as noted in (Valmeekam et al. 2022, 2023), these methods struggle with more complex planning tasks.

One approach to address these limitations is using an LLM as an interface for symbolic reasoning engines. This includes generating executable Python code, as explored in recent work where natural language and Python program pairs are used to produce code for reasoning tasks (Olausson et al. 2023; Gao et al. 2023; Chen et al. 2023b; Lyu et al. 2023; Singh et al. 2023). While this approach offloads much of the computation to the Python interpreter, it is not well-suited for planning tasks that involve constraints that are not easily expressible in a procedural language.

Another alternative is the use of Planning Domain Definition Language (PDDL) with LLMs. Some studies (Liu et al. 2023; Xie et al. 2023) have focused on translating English instructions into PDDL goals, assuming pre-existing PDDL action descriptions. Since only an instance file or goal needs to be generated, this setting is considerably simpler. Some recent works embrace a human-in-the-loop approach with LLMs, using human feedback when constructing domain models and executing plans (Guan et al. 2023; Huang et al. 2023; Yao et al. 2023). Closest to our approach, Guan et al. (2023) employed an LLM for generating PDDL descriptions but noted that many manual corrections by PDDL experts were necessary due to errors in GPT-4's translations, which impacted their execution by a PDDL solver.

A number of recent works show some success using LLMs to iteratively revise their own output, surpassing baseline LLM performance while bypassing expensive human feedback (Madaan et al. 2024; Kim, Baldi, and McAleer 2024). In particular, LLMs are well-suited for self-revision when they have access to external forms of feedback, such as external knowledge or tools (e.g., a code interpreter) (Kamoi et al. 2024; Stechly, Valmeekam, and Kambhampati 2024b; Guan et al. 2023).

2.2 Action Languages

Action languages, such as $\mathcal{A}$ (Gelfond and Lifschitz 1993), $\mathcal{B}$ (Gelfond and Lifschitz 1998), $\mathcal{C}$ (Giunchiglia and Lifschitz 1998), $\mathcal{C}+$ (Giunchiglia et al. 2004), $\mathcal{BC}$ (Lee, Lifschitz, and Yang 2013), and $\mathcal{BC}+$ (Babb and Lee 2015, 2020), represent subsets of natural language specifically designed for describing actions and their effects. These languages are often viewed as high-level notations of answer set programs (Lifschitz 2008; Brewka, Eiter, and Truszczyński 2011; Lee and Meng 2008; Gebser, Lee, and Lierler 2006), structured to effectively represent transition systems. Key research topics in this field include the exploration of their expressive possibilities, such as indirect effects, triggered actions, defaults, and additive fluents (Giunchiglia et al. 2004; Gelfond and Lifschitz 1998; Lee and Lifschitz 2003; Inclezan and Gelfond 2016). Such languages offer greater expressiveness than PDDL, which has been well-studied in the literature (Eyerich et al. 2006; Jiang et al. 2019). Despite the rich body of research surrounding action languages, a significant challenge remains: automation of action language generation,

Listing 1: $\mathcal{BC}+$ signature for Blocks World

```
:- sorts
   loc >> block.

:- objects
   b1, b2, b3, b4      :: block;
   table               :: loc.

:- constants
   loc(block)         :: inertialFluent(loc);
   move(block, loc)   :: exogenousAction.
```

Listing 2: $\mathcal{BC}+$ rules for Blocks World (% is for comments)

```
% Moving a block changes its location.
move(B,L) causes loc(B)=L.

% Can't move block with something on it.
nonexecutable move(B,L) if loc(B1)=B.

% Two blocks cannot be on the same block.
impossible loc(B1)=B & loc(B2)=B & B1\=B2.
```

which we address in this paper.

Constants in $\mathcal{BC}$+ are categorized into 'fluent' and 'action' constants. For instance, in the Blocks World domain, $move(B, L)$ represents an action constant with Boolean values (indicating whether the action is executed), while $loc(B)$ is a fluent constant with location values, where B is a variable spanning over blocks.

The rules in $\mathcal{BC}$+ are called *causal laws*. An example is

$$move(B, L) \textbf{ causes } loc(B) = L \tag{1}$$

which represents that moving a block B to a location L results in the block's location being L. Another rule

$$\textbf{nonexecutable } move(B, L) \textbf{ if } loc(B_1) = B \tag{2}$$

states that moving a block B is not executable if another block B_1 is on top of it. Additionally,

$$\textbf{impossible } loc(B_1) = B \wedge loc(B_2) = B \tag{3}$$

$(B_1 \neq B_2)$ illustrates a state constraint where two distinct blocks cannot occupy the same block. The entire Blocks World domain can be described by these rules and a few extra ones. This succinctness is thanks to the separation between the representation language and the efficient constraint satisfaction algorithm for it. For a comprehensive review of $\mathcal{BC}$+, we refer the reader to (Lee, Lifschitz, and Yang 2013; Babb and Lee 2020).

The input language of the $\mathcal{BC}$+ reasoner provides a convenient way of expressing $\mathcal{BC}$+ descriptions. It allows for declaring `sorts`, `objects` that belong to some `sort`, and `constants`, such as fluents and actions. For example, the signature for the Blocks World is shown in Listing 1 Additionally, the causal laws (1)–(3) above can be expressed in the language of the $\mathcal{BC}$+ as in Listing 2.

3 Our Method

Our framework, as depicted in Figure 1, comprises four principal components: $\mathcal{BC}+$ Signature Generation, English Knowledge Generation, $\mathcal{BC}+$ Rule and Query Generation, and Self-Revision. The $\mathcal{BC}+$ Signature Generation is responsible for defining necessary symbols. English Knowledge Generation involves extracting and structuring relevant information from the natural language problem description, while $\mathcal{BC}+$ Rule Generation focuses on translating this structured knowledge into formal $\mathcal{BC}+$ rules, thereby bridging the gap between natural language understanding and symbolic reasoning. Finally, Self-Revision iteratively refines the $\mathcal{BC}+$ signature, rules, and query, with feedbacks from the $\mathcal{BC}+$ reasoner when run on a set of queries generated by the LLM. The result is either a correct program (and a correct solution) or a program that requires a typically small number of corrections to yield a correct solution.

3.1 Input

The input is a natural language description of the problem, including descriptions of types, objects, and actions involving them, along with a query in natural language. For example, the input for the Missionaries and Cannibals puzzle (MCP) is given in Appendix A.1.

3.2 $\mathcal{BC}+$ Signature Generation

Given the problem description in English, this step generates a signature in $\mathcal{BC}$+ syntax. Writing such a $\mathcal{BC}+$ program typically starts with understanding the problem and considering the dynamics (knowledge) required, thinking about what fluent and action constants are useful, and then writing rules about them. We present the LLM with the problem and prompt it to generate important parts of the problem in natural language before signature generation. The Signature Generation prompt (See Appendix B.1) contains an introduction to $\mathcal{BC}+$ and a few example translations of the English description to important knowledge, an analysis of constants and their natural language reading, and finally, a $\mathcal{BC}$+ signature. Only the natural language reading of constants and signature are passed to the rest of the pipeline. For the MCP puzzle, the generated signature can be found in Appendix A.2.

3.3 Knowledge Generation

We leverage an LLM to extract relevant knowledge from a problem description for use by the $\mathcal{BC}$+ reasoner. This is achieved by using a prompt that includes instructions and a few example domain descriptions in English, along with their corresponding knowledge statements in English. The generated knowledge statements broadly fall into two categories, *commonsense knowledge* and *domain-specific knowledge*. Commonsense knowledge refers to information that is not explicitly stated in the problem description and usually takes the form of cause-and-effect relationships. For example, in the MCP domain, this step correctly generates the commonsense knowledge statements: *"crossing a vessel causes the location of the vessel to change"* and *"crossing a vessel causes the number of a group at a location to decrease by the amount of members on the vessel."* Enumerating such commonsense knowledge can be tedious and prone to omissions. Therefore, we find it beneficial to use an LLM

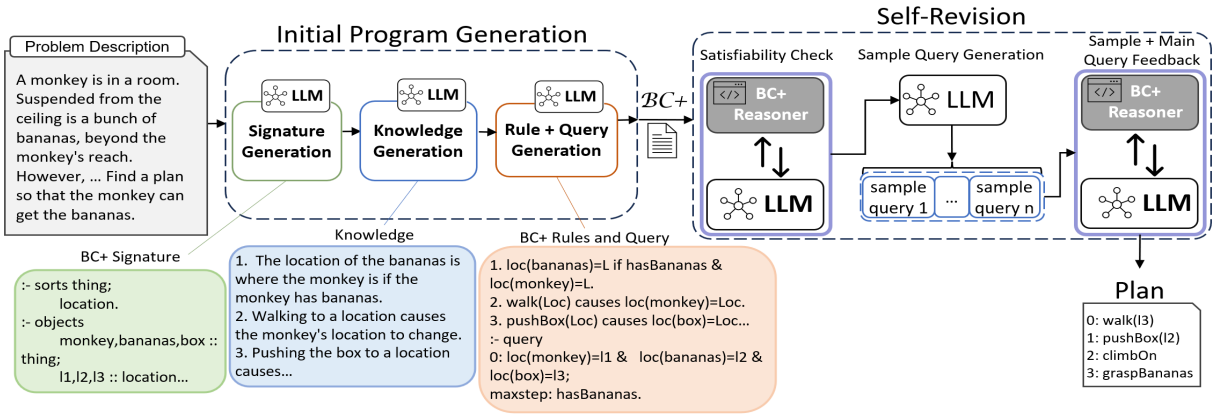

Figure 1: LLM+AL pipeline

for this task. Domain-specific knowledge is directly tied to the given information about the problem, for instance, *"Missionaries should not be outnumbered by cannibals, or they will be eaten."* The full prompt for extracting knowledge is shown in Appendix B.2. The complete knowledge generated for MCP is shown in Appendix A.3.

3.4 Rules and Query Generation

Further leveraging the capabilities of LLMs, we use them to convert natural language into a symbolic representation. This is done with a prompt that contains a brief introduction of action language $\mathcal{BC}+$ and few-shot examples of natural language knowledge translated into $\mathcal{BC}+$ rules. This prompt, together with the natural language reading of constants, signatures, and the English knowledge generated in previous steps, is given as input to the LLM, which then generates $\mathcal{BC}+$ causal laws. The full prompt is given in Appendix B.3.

For example, the knowledge *"crossing a vessel causes its location to change"* is turned into:

```
cross(V) causes loc(V)=Loc if going(V)=Loc.
```

3.5 Self-Revision

The programs generated so far may have errors, either syntactically or semantically. This step leverages an LLM to revise the generated $\mathcal{BC}+$ program based on the reasoner's output.

Satisfiability Check. First, without considering any initial or goal state, the pipeline performs a *satisfiability check*, which runs the $\mathcal{BC}+$ reasoner to ensure if the generated $\mathcal{BC}+$ signature and rules (without a query) are satisfiable. Typical errors detected in this step are syntax errors and reported by the $\mathcal{BC}+$ reasoner. Based on the output message of the $\mathcal{BC}+$ reasoner (e.g., syntax errors on some rules or undeclared constants), the LLM is prompted to update the signature and/or rules accordingly. This step repeats until either the program is satisfiable, in which case the pipeline proceeds to Sample Query Generation, or the maximum number of allowed revisions is reached, in which case the pipeline ends on this step, settling on the current signature, rules, and the main query.

Sample Query Generation. Next, the pipeline prompts the LLM to generate a small set of simple sample queries. These queries will later serve to check that the program correctly implements domain-specific rules and constraints (e.g., actions lead to expected changes in the state, preconditions for actions are respected, etc.). The LLM is instructed to append either "(satisfiable)" or "(unsatisfiable)", depending on whether the query is expected to be satisfiable or unsatisfiable based on the LLM's discretion. For example, a sample query for the Missionaries and Cannibals puzzle could involve an action which is is expected to be disallowed, such as crossing on the boat with more than the allowed capacity, which would be appended with "(unsatisfiable)".

Sample and Main Query Feedback. The pipeline automatically executes each sample query and the main query using the $\mathcal{BC}+$ reasoner, and the outputs are provided to the LLM. The LLM is then tasked with verifying that the outputs align with the domain and revising the $\mathcal{BC}+$ signature, causal laws, main query, and/or sample queries as needed. If no changes are required or the maximum number of revisions is reached, the resulting $\mathcal{BC}+$ program and the $\mathcal{BC}+$ reasoner's output are finalized. Otherwise, the process repeats. See Appendix A.6. for an example of the Self-Revision step.

4 Experiments

Benchmarks. We consider benchmarks focusing on complex reasoning. The first set is from (McCarthy 1998), where McCarthy proposed several variations of the well-known Missionaries and Cannibals puzzle. The second set consists of several well-known puzzles along with our own variations.

Baselines. For the baseline LLMs, we use ChatGPT-4, Claude 3 Opus, Gemini 1.0 Ultra, and o1-preview.[1] These baseline models are provided with problem descriptions in natural language, as outlined in Section 3.1, and

[1] All baseline experiments were conducted in May 2024, except for o1-preview, which was conducted in September 2024. For other experiments with more recent LLM, please refer to our github page.

tasked with finding a solution. Additionally, we evaluate CHATGPT-4 using its code interpreter feature to generate Python programs for solving the problems. This method is referred to as CHATGPT-4+CODE. Similar to our pipeline, we prompt CHATGPT-4+CODE to iteratively revise its generated programs as many times as needed.

Recent studies have shown that methods like Chain-of-Thought (CoT) (Wei et al. 2022), Program-of-Thought (PoT) (Chen et al. 2023b), and similar prompting techniques applied to enterprise LLMs have not been effective and may even perform worse (Valmeekam et al. 2023; Chen et al. 2023a; Stechly, Valmeekam, and Kambhampati 2024a). This is likely because such methods are already incorporated during the instruction following training of these LLMs. In preliminary experiments, we observed the same, so we did not include these methods in the final experiments.

4.1 Experiment Results

Benchmark Performance. As shown in Table 1, the baseline LLMs perform poorly on the MCP elaboration problems. Neither CLAUDE 3 OPUS nor GEMINI ULTRA 1.0 solve any MCP problems correctly, while CHATGPT-4 solves only three. O1-PREVIEW does better, solving 7 problems correctly. CHATGPT-4+CODE produces correct plans for 5 elaborations and occasionally generates solutions that are mostly correct but include minor issues that can be easily fixed manually. These cases are denoted with Δ in Table 1. LLM+AL[2] automatically solves 7 MCP problems. For the remaining problems, an average of 3.1 manual corrections to the generated $\mathcal{BC}+$ program are required to produce correct plans.

We further include superficial variations of standard puzzles and observe that, even for these simple elaborations, the results are similar, as shown in Table 2. These variations involve minor changes to the initial states (e.g., in Tower of Hanoi variations, disks are distributed among pegs in no particular order). The baseline LLMs perform poorly on these variations, with CHATGPT-4 and O1-PREVIEW solving 1 and 2 variations correctly, respectively. While CHATGPT-4+CODE performs better than the baseline LLMs, it still struggles with these variations. In contrast, LLM+AL outperforms the baseline LLMs, and for the problems it fails to solve correctly, an average of only 2.2 corrections are required to produce the correct output.

Effectiveness of Self-Revision. Self-Revision provides a notable improvement in the quality of the generated $\mathcal{BC}+$ programs. Across all 30 problems in Tables 1 and 2, only 22.3% (7/30) of programs generated prior to the Self-Revision step are executable without syntax errors. This percentage increases substantially to 86.7% (26/30) after the Self-Revision step. Similarly, the proportion of programs that produce correct answers when run through the $\mathcal{BC}+$ reasoner rises from 16.6% (5/30) before Self-Revision to 50% (15/30) afterward. In terms of issues requiring correction, there are 70 issues in the $\mathcal{BC}+$ programs prior to the Self-Revision step, but this number decreases to 42 following the Self-Revision step, with the detailed breakdown as explained in the next paragraph.

Program Issues. Table 3 enumerates all issues encountered in the final $\mathcal{BC}+$ programs produced by the LLM+AL pipeline. Overall, we classify the 42 total issues into three categories: signature issues, rule issues, and query issues. Detailed descriptions of all issue cases are provided in Appendix D.

Signature Issues. There are 12 signature issues in total that can be categorized into three subcategories. The first and most common subcategory involves missing declarations for sorts, objects, variables, or constants, accounting for 7 cases. The second subcategory, comprising a single case, pertains to syntactically incorrect declarations. The third subcategory includes 4 cases and involve semantic issues in declarations, such as an incorrect supersort statement, which erroneously specify that certain objects are default members of another type.

Rule Issues. There are 25 rule issues in total that can be categorized into three subcategories. The first subcategory, comprising 48.0% (12/25) of the issues, involves missing necessary rules required to solve the problems. The second subcategory accounts for 20.0% (5/25) of the rule issues and involves harmful rules that represent constraints or conditions not specified in the problem. For example, in MCP #6 (where only one missionary and one cannibal can row), the pipeline generates an unnecessary rule that disallows both the missionary rower and the cannibal rower from being on the boat simultaneously. The final subcategory accounts for 32.0% (8/25) of rule issues and involves harmful rules that attempt to represent an aspect of the problem but do so incorrectly. For example, in MCP #17 (where cannibals can become hungry), the pipeline generates an incorrect rule: "On either bank, if there are missionaries present, the number of cannibals cannot exceed the number of missionaries." This rule fails to accurately reflect the elaboration, which specifies that the cannibals won't become hungry as long as the strong missionary is rowing.

Query Issues. There are 5 query issues in total that can be categorized into two subcategories: Syntactic issues, which account for 2 cases, involve errors such as using invalid keywords like "initially" or "goal." Semantic issues, which account for 3 cases, pertain to incorrect initial and/or goal state conditions.

4.2 Analysis

LLMs struggle to consistently adhere to state constraints. Of the 17 problems in Table 1, 14 are solvable, meaning they have valid plans to reach the goals. CLAUDE 3 OPUS and GEMINI 1.0 ULTRA fail to solve any of them correctly, while CHATGPT-4 produces 11 incorrect solutions. Among these incorrect plans, the state constraint—mandating that missionaries must not be outnumbered by cannibals—is frequently violated, despite clear instructions to adhere to it. Notably, 89.7% (35 out of 39) of these violations occur within the first three steps of the

[2]For LLM+AL , we use O1-PREVIEW ("o1-preview" in the API) as the underlying LLM.

Problem	Opt. Length	Chat GPT4	Claude 3 Opus	Gemini 1 Ultra	ChatGPT4 Code	o1 (preview)	LLM +AL
MCP (basic)	11	×	×	×	✓[1]	✓	✓
1 (the boat is a rowboat)	11	×	×	×	✓[1]	✓	✓
2 (missionaries and cannibals can exchange hats)	11	✓	×	×	✓[2]	×	△(1)
3 (there are 4 missionaries and 4 cannibals)	(unsolvable)	×	×	×	Δ[6]	✓	△(2)
4 (the boat can carry three, 5 missionaries/cannibals)	(unsolvable)	×	×	×	×[2]	✓	✓
5 (an oar on each bank)	13	×	×	×	×[1]	×	△(2)
6 (only one missionary and one cannibal can row)	13	×	×	×	×[1]	×	△(4)
7 (missionaries cannot row)	(unsolvable)	×	×	×	Δ[6]	✓	✓
8 (a very big cannibal must cross alone)	15	×	×	×	×[2]	×	△(2)
9 (big cannibal and small missionary)	11	×	×	×	×[1]	×	△(6)
10 (a missionary can walk on water)	7	×	×	×	✓†[1]	×	△(7)
11 (missionaries can convert cannibals)	9	×	×	×	✓†[2]	×	✓†
13 (there is bridge that can cross two, 5 missionaries/cannibals)	4	×	×	×	×[6]	✓†	✓†
14 (the boat leaks with two people on it)	11	✓	×	×	×[1]	×	△(3)
16 (there is an island, 5 missionaries/cannibals)	19	×	×	×	×[4]	×	✓
17 (cannibals can become hungry, 4 missionaries/cannibals)	13	×	×	×	×[4]	×	△(3)
19 (there are two sets of groups)	22	✓	×	×	×[1]	✓	△(1)
Total		3	0	0	5+2△	7	7+10△

Table 1: Performance on MCP and its elaborations. (We exclude a few elaborations because they require probabilistic reasoning or the descriptions are vague.) Δ indicates human intervention was used to produce a correct result. $[n]$ indicates the number of attempts ChatGPT-4+Code makes at writing a program, and (n) indicates the number of manual corrections required for LLM+AL. † indicates that the solution found was not optimal. All elaborations are listed in Appendix E.

Problem	Opt. Length	Chat GPT4	Claude 3 Opus	Gem. 1.0 Ultra	ChatGPT-4 +Code	o1 preview	LLM+ AL
River Cross (basic)	7	✓	✓	✓	✓[5]	✓	✓
River Cross (var1)	6	×	×	×	×[3]	✓	✓
Tower of Hanoi (3-disk, basic)	7	×	×	×	✓[1]	✓	✓
Tower of Hanoi (3-disk, var1)	6	×	×	×	×[1]	✓	△(1)
Tower of Hanoi (5-disk, basic)	31	✓*	×	×	✓[1]	✓	✓
Tower of Hanoi (5-disk, var1)	27	×	×	×	×[4]	×	✓
Tower of Hanoi (7-disk, basic)	127	×	×	×	✓[1]	×	✓
Tower of Hanoi (7-disk, var1)	11	×	×	×	×[2]	×	△(1)
Sudoku1	0	✓*	×	×	✓[1]	×	△(2)
Sudoku2	0	✓*	×	×	✓[1]	×	✓
Sudoku3	0	×*	×	×	✓[3]	×	✓
Sudoku (var1)	(unsolvable)	✓*	×	×	Δ[3]	×	△(1)
Sudoku (var2)	(unsolvable)	×*	×	×	✓[0]	×	△(6)
Total		5	1	1	8+1Δ	5	8+5Δ

Table 2: Performance on some puzzles and their variations.* indicates that the LLM voluntarily generated Python code.

plan.[3] Even when these LLMs output intermediate states during plan generation, they fail to address apparent state constraint violations and continue generating flawed plans. In contrast, o1-preview demonstrates a better adherence to constraints, producing fewer plans that violate state constraints and exhibiting more thoughtful consideration of the effects of actions and validation of states. However, it occasionally refrains from generating plans for the problems known to be solvable, as discussed in the next paragraph.

LLMs do not reliably distinguish between solvable/unsolvable problems. Some MCP elaborations are inherently unsolvable (MCP elaborations #3, #4, #7). Even for these instances, ChatGPT-4, Claude 3 Opus, and Gemini 1.0 Ultra generate (incorrect) plans, which can be considered hallucinations. Interestingly, these plans share some similarities. For example, in Elaborations #3 and #4, all plans include a state where cannibals outnumber missionaries. For Elaboration #7, where no missionaries are allowed to row, all plans generated by these LLMs include a missionary rowing despite this being a clear violation of the elaboration. o1-preview performs better in recognizing unsolvable problems, correctly identifying that no plans are possible for the three unsolvable elaborations. However, it is unclear how o1-preview arrives at these conclusions, as its (partial) output does not seem to rule out all possibilities.[4] Moreover, it incorrectly concludes that no solution exists in five solvable instances (#5, #6, #8, #10, and #17). Notably, o1-preview is the only baseline model to explicitly claim that a solution is impossible, but it suffers from a high rate of false negatives. In contrast, LLM+AL reliably ensures that there are no possible plans *up to certain fixed lengths* for Elaborations #4 and #7 automatically, and for Elaboration #3 with minimal modifications, leveraging the formal semantics of $\mathcal{BC}+$ to validate its conclusions.[5]

[3] All plans produced by baseline LLMs are available in the code repository.

[4] o1-preview does not show all of its CoT reasoning to users.

[5] Strictly speaking, our pipeline is limited because it doesn't

Programs	Issues								
	Signature			Rules			Query		Total
	MD	Syn	Sem	MN	HU	HN	Syn	Sem	
Before	18	11	3	16	5	10	1	6	70
After	7	1	4	12	5	8	2	3	42

Table 3: Number of issues manually examined before (Before) and after Self-Revision (After). The signature issues are missing sorts, object, variables, or constants in the declaration (MD), syntactic issues (Syn), or semantic issues (Sem). The rule issues are missing necessary rules (MN), harmful rules attempting to represent something not specified in the problem (HU), and harmful rules attempting to represent an aspect of the problem (HN). The query issues are either syntactic (Syn) or semantic (Sem).

LLM code generation struggles to adapt to problem elaborations. We observe that CHATGPT-4+CODE struggles to correctly incorporate new information into the Python programs it generates. In particular, it frequently fails to model actions accurately. As shown in Table 1, 5 of the 12 failures by CHATGPT-4+CODE on the elaborations (#4, #5, #6, #14, #17) occur because the actions in the generated plans are underspecified. For example, some plans omit critical information, such as identifying who is rowing, which renders the plans unusable. The remaining 7 failures stem from a mix of issues: 2 precondition violations, 2 instances where no plans were generated, and 3 cases where unsolvable problems were not identified as such. This behavior is further corroborated by the results in Table 2, where CHATGPT-4+CODE fails all problem variations except for one Sudoku example, despite correctly solving all the corresponding basic problems.

CHATGPT-4+CODE coincidentally succeeds in cases where it reuses the solution for the standard problem (MCP #1, #2, #10, and #11), producing identical plans for all. While the original solution happens to be valid for these elaborations, it may be suboptimal, as seen in #10 and #11. Notably, although these programs generate valid—and occasionally optimal—plans, they fail to accurately model the specific details introduced in the elaborations. Additionally, CHATGPT-4+CODE exhibits an over-reliance on the standard problem's code, misapplying it to 5 MCP elaborations (#5, #6, #7, #13, and #14). As in the coincidentally correct cases, CHATGPT-4+CODE ignores the relevant details specific to the elaborations and demonstrates a bias toward reproducing code for the basic MCP problem. For MCP problems, all programs generated by CHATGPT-4+CODE use naive search algorithms: either breadth-first search (11 cases) or depth-first search (5 cases), often taking too long to find plans. In 9 instances, CHATGPT-4+CODE revises the code at least once.

Declarative semantics of $\mathcal{BC}+$ works well with LLMs. Unlike the LLM+Code approach, LLM+AL does not require specifying which algorithms to use, thanks to the declarative semantics of action languages. The search algorithm implemented in the $\mathcal{BC}+$ reasoner is highly optimized for constraint satisfaction problems and can generate plans instantly.

Self-Revision with the solver feedback significantly improves $\mathcal{BC}+$ program quality. Much like how a human might debug a program by testing its behavior and refining it based on feedback, Self-Revision enhances program quality through iterative refinement. It executes the program to verify satisfiability and employs simple queries generated by the LLM to ensure that the $\mathcal{BC}+$ program accurately models the problem. Additionally, it addresses both syntax and semantic errors by directly incorporating feedback from the $\mathcal{BC}+$ reasoner, progressively improving the program's correctness. This approach significantly reduces the burden on the user, as many issues are resolved automatically. Even when some issues remain, they are relatively straightforward to address, further streamlining the process of creating accurate and robust $\mathcal{BC}+$ programs.

LLM+AL benefits from human corrections, unlike LLM or LLM+Code. Occasionally requiring human corrections is a limitation of LLM+AL, stemming from the fact that LLMs still make mistakes when generating action language representations. Despite this, these mistakes are relatively easy to correct due to the declarative semantics of $\mathcal{BC}+$. On the other hand, using only LLMs to solve these benchmarks fails to benefit from human corrections. We attempted 50 iterations of human corrections with CHATGPT-4 by indicating which parts of its answer were incorrect, but this did not help. For instance, in MCP elaboration #13 (The Bridge), it repeatedly violated the constraint that cannibals should not outnumber missionaries. Regarding CHATGPT-4+CODE, Python code is much less constrained and harder to interpret for action domains compared to $\mathcal{BC}+$, making it considerably more difficult for a human to correct the errors.

5 Conclusion

We propose LLM+AL, a framework that bridges LLMs with action languages, enabling them to complement each other. Compared to the direct use of LLMs, LLM+AL achieves more robust and accurate reasoning about actions by leveraging the expressiveness and formal reasoning capabilities of action languages. While the generation of action language descriptions traditionally requires human expert knowledge, LLM+AL simplifies this process through an automated process. Additionally, we employ a Self-Revision mechanism, an iterative approach in which an LLM generates sample queries to test the correctness of its previously generated $\mathcal{BC}+$ program. Based on feedback from the $\mathcal{BC}+$ reasoner, the LLM revises its program, significantly improving the quality of the final output. While some mistakes may persist in the final programs, the generative capabilities of LLMs make creating action descriptions significantly easier compared to crafting them from scratch. It is likely that future LLM improvements will further reduce such errors. Moreover, fine-tuning LLMs could further enhance the performance of LLM+AL provided it is feasible.

guarantee the non-existence of a plan of *arbitrary* length, for which one can use the method in Sec 6.6 of (Lee 2005).

Acknowledgements

We are grateful to the anonymous referees for their useful comments. This work was partially supported by the National Science Foundation under Grant IIS-2006747.

References

Anthropic. 2024. The Claude 3 Model Family: Opus, Sonnet, Haiku. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf. Accessed: 2024-05-01.

Babb, J.; and Lee, J. 2013. Cplus2ASP: Computing Action Language $\mathcal{C}$+ in Answer Set Programming. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 122–134.

Babb, J.; and Lee, J. 2015. Action Language BC+: Preliminary Report. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*.

Babb, J.; and Lee, J. 2020. Action language BC+. *Journal of Logic and Computation*, 30(4): 899–922.

Besta, M.; Blach, N.; Kubicek, A.; Gerstenberger, R.; Podstawski, M.; Gianinazzi, L.; Gajda, J.; Lehmann, T.; Niewiadomski, H.; Nyczyk, P.; et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 17682–17690.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM*, 54(12): 92–103.

Brohan, A.; Chebotar, Y.; Finn, C.; Hausman, K.; Herzog, A.; Ho, D.; Ibarz, J.; Irpan, A.; Jang, E.; Julian, R.; et al. 2023. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on robot learning*, 287–318. PMLR.

Chen, J.; Chen, L.; Huang, H.; and Zhou, T. 2023a. When do you need chain-of-thought prompting for chatgpt? *arXiv preprint arXiv:2304.03262*.

Chen, W.; Ma, X.; Wang, X.; and Cohen, W. W. 2023b. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks. *Transactions on Machine Learning Research*.

Eyerich, P.; Nebel, B.; Lakemeyer, G.; and Claßen, J. 2006. GOLOG and PDDL: What is the Relative Expressiveness? In *Proceedings of the 2006 international symposium on Practical cognitive agents and robots*, 93–104.

Gao, L.; Madaan, A.; Zhou, S.; Alon, U.; Liu, P.; Yang, Y.; Callan, J.; and Neubig, G. 2023. PAL: Program-aided language models. In *International Conference on Machine Learning*, 10764–10799. PMLR.

Gebser, M.; Lee, J.; and Lierler, Y. 2006. Elementary Sets for Logic Programs. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*.

Gelfond, M.; and Lifschitz, V. 1993. Representing action and change by logic programs. *Journal of Logic Programming*, 17: 301–322.

Gelfond, M.; and Lifschitz, V. 1998. Action languages. *Electronic Transactions on Artificial Intelligence*, 3: 195–210.

Gemini Team. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Giunchiglia, E.; Lee, J.; Lifschitz, V.; McCain, N.; and Turner, H. 2004. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2): 49–104.

Giunchiglia, E.; and Lifschitz, V. 1998. An action language based on causal explanation: Preliminary report. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, 623–630. AAAI Press.

Guan, L.; Valmeekam, K.; Sreedharan, S.; and Kambhampati, S. 2023. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36: 79081–79094.

Huang, W.; Abbeel, P.; Pathak, D.; and Mordatch, I. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, 9118–9147. PMLR.

Huang, W.; Xia, F.; Xiao, T.; Chan, H.; Liang, J.; Florence, P.; Zeng, A.; Tompson, J.; Mordatch, I.; Chebotar, Y.; et al. 2023. Inner Monologue: Embodied Reasoning through Planning with Language Models. In *Conference on Robot Learning*, 1769–1782. PMLR.

Inclezan, D.; and Gelfond, M. 2016. Modular action language. *Theory and Practice of Logic Programming*, 16(2): 189–235.

Ishay, A.; and Lee, J. 2025. LLM+AL: Bridging Large Language Models and Action Languages for Complex Reasoning About Actions. *arXiv preprint arXiv:/2501.00830*.

Ishay, A.; Yang, Z.; and Lee, J. 2023. Neuro-Symbolic Reasoning with Large Language Models and Answer Set Programming: A Case Study on Logic Puzzles. In *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning*.

Jiang, Y.-q.; Zhang, S.-q.; Khandelwal, P.; and Stone, P. 2019. Task planning in robotics: an empirical comparison of pddl-and asp-based systems. *Frontiers of Information Technology & Electronic Engineering*, 20: 363–373.

Kamoi, R.; Zhang, Y.; Zhang, N.; Han, J.; and Zhang, R. 2024. When Can LLMs Actually Correct Their Own Mistakes? A Critical Survey of Self-Correction of LLMs. *Transactions of the Association for Computational Linguistics*, 12: 1417–1440.

Kim, G.; Baldi, P.; and McAleer, S. 2024. Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36.

Kojima, T.; Gu, S. S.; Reid, M.; Matsuo, Y.; and Iwasawa, Y. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213.

Lee, J. 2005. Automated Reasoning about Actions. Ph.D. thesis, University of Texas at Austin.

Lee, J.; and Lifschitz, V. 2003. Describing Additive Fluents in Action Language $\mathcal{C}$+. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 1079–1084.

Lee, J.; Lifschitz, V.; and Yang, F. 2013. Action Language $\mathcal{BC}$: Preliminary Report. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

Lee, J.; and Meng, Y. 2008. On loop formulas with variables. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, 444–453.

Lifschitz, V. 2008. What Is Answer Set Programming? In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1594–1597. MIT Press.

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 preprint arXiv:2304.11477*.

Lyu, Q.; Havaldar, S.; Stein, A.; Zhang, L.; Rao, D.; Wong, E.; Apidianaki, M.; and Callison-Burch, C. 2023. Faithful Chain-of-Thought Reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, 305–329.

Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

McCarthy, J. 1998. Elaboration tolerance. In *Working Papers of the Fourth Symposium on Logical Formalizations of Commonsense Reasoning*.

Nye, M.; Tessler, M.; Tenenbaum, J.; and Lake, B. M. 2021. Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning. *Advances in Neural Information Processing Systems*, 34: 25192–25204.

Olausson, T.; Gu, A.; Lipkin, B.; Zhang, C.; Solar-Lezama, A.; Tenenbaum, J.; and Levy, R. 2023. LINC: A Neurosymbolic Approach for Logical Reasoning by Combining Language Models with First-Order Logic Provers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 5153–5176.

OpenAI. 2024a. Introducing ChatGPT. https://openai.com/index/chatgpt. Accessed: 2024-05-03.

OpenAI. 2024b. o1-preview System Card. Accessed: 2024-09-13.

Singh, I.; Blukis, V.; Mousavian, A.; Goyal, A.; Xu, D.; Tremblay, J.; Fox, D.; Thomason, J.; and Garg, A. 2023. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 11523–11530. IEEE.

Stechly, K.; Valmeekam, K.; and Kambhampati, S. 2024a. Chain of thoughtlessness: An analysis of cot in planning. *arXiv preprint arXiv:2405.04776*.

Stechly, K.; Valmeekam, K.; and Kambhampati, S. 2024b. On the self-verification limitations of large language models on reasoning and planning tasks. *arXiv preprint arXiv:2402.08115*.

Valmeekam, K.; Marquez, M.; Sreedharan, S.; and Kambhampati, S. 2023. On the planning abilities of large language models-a critical investigation. *Advances in Neural Information Processing Systems*, 36: 75993–76005.

Valmeekam, K.; Olmo, A.; Sreedharan, S.; and Kambhampati, S. 2022. Large Language Models Still Can't Plan (A Benchmark for LLMs on Planning and Reasoning about Change). In *NeurIPS 2022 Foundation Models for Decision Making Workshop*.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q. V.; Zhou, D.; et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35: 24824–24837.

Xie, Y.; Yu, C.; Zhu, T.; Bai, J.; Gong, Z.; and Soh, H. 2023. Translating natural language to planning goals with large-language models. *arXiv preprint arXiv:2302.05128*.

Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.; Cao, Y.; and Narasimhan, K. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

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 *International Conference on Learning Representations (ICLR)*.

Zeng, A.; Attarian, M.; Choromanski, K. M.; Wong, A.; Welker, S.; Tombari, F.; Purohit, A.; Ryoo, M. S.; Sindhwani, V.; Lee, J.; et al. 2022. Socratic Models: Composing Zero-Shot Multimodal Reasoning with Language. In *The Eleventh International Conference on Learning Representations*.