Proceedings of the 23rd International Conference on Principles of Knowledge Representation and Reasoning
KR meets Machine Learning and Explanation

ALM–ASP: A Functional Agentic Architecture for Answer Set Programming

Luis Angel Rodriguez Reiners[1] , **Alice Tarzariol**[2] , **Mario Alviano**[1] ,
Manuel Borroto Santana[1] , **Kostantin Schekotihin**[2]

[1]Department of Mathematics and Computer Science, University of Calabria
[2]Department of Artificial Intelligence and Cybersecurity, University of Klagenfurt
{luis.reiners, mario.alviano, manuel.borroto}@unical.it, {alice.tarzariol, konstantin.schekotihin}@aau.at

Abstract

Answer Set Programming (ASP) is a declarative formalism widely used in knowledge representation and reasoning for modeling and solving combinatorial problems, yet current Large Language Models (LLMs) often struggle to generate correct programs from natural language specifications. This difficulty stems both from the limited presence of ASP in training corpora and from the strict syntactic and semantic constraints imposed by stable model semantics. We introduce **ALM–ASP** (**A**gentic **L**oop for **M**odeling in **ASP**), a multi-agent architecture for automatic ASP modeling grounded in a functional model of language agents equipped with tools and persistent state. ALM–ASP instantiates this model via two interacting agents: a *Modeler*, which incrementally constructs candidate ASP programs, and a *Validator*, which assesses their alignment with the original specification and provides feedback for refinement. The agents interact through a shared ASP execution environment backed by the CLINGO engine, yielding an iterative construct–validate loop. An empirical evaluation on a challenging subset of CP–Bench and on problems from recent LP/CP Programming Contests shows that ALM–ASP significantly improves both syntactic validity and end-to-end correctness over general-purpose LLM baselines, and also achieves improved instance coverage compared to the closest agentic alternative, CP–Agent.

1 Introduction

Large Language Models (LLMs) have recently demonstrated impressive performance across a wide range of natural language processing tasks (Minaee et al. 2024). Despite these advances, LLMs still exhibit significant limitations when applied to tasks requiring precise logical modeling, multi-step deductive reasoning, or strict adherence to formal semantics (Liu et al. 2023; Bang et al. 2023; Cohn 2023). These limitations are particularly evident when LLMs are used to generate programs in declarative knowledge representation languages, where even small syntactic or semantic errors can invalidate the entire specification.

Answer Set Programming (ASP) (Marek and Truszczynski 1999; Niemelä 1999) is a well-established declarative formalism in knowledge representation and reasoning, widely used for modeling and solving complex combinatorial problems under stable model semantics (Gelfond and Lifschitz 1990). While ASP engines can reliably perform reasoning and verification once a correct program is provided, automatically generating such programs from natural language specifications remains a challenging task. Current LLM-based approaches often struggle to produce syntactically valid and semantically faithful ASP programs, especially in the absence of direct interaction with ASP tools.

A growing body of work explores the integration of LLMs with symbolic reasoning systems, often referred to as neuro-symbolic approaches (Ishay, Yang, and Lee 2023; Wang, Sun, and Kuhn 2024; Alviano et al. 2025). In many such approaches, LLMs are used to parse or translate natural language input, while symbolic engines are employed for reasoning and verification. However, in practice, the modeling process itself remains largely manual: users must iteratively inspect generated programs, execute them using an ASP engine, and provide corrective feedback to the LLM. This manual validation loop is time-consuming, error-prone, and cognitively demanding, often leading to shallow feedback that fails to address deeper modeling issues.

In this paper, we address this gap by introducing an agentic architecture that automates the iterative modeling and validation process for ASP. We propose **ALM–ASP** (**A**gentic **L**oop for **M**odeling in **ASP**), a multi-agent system in which a *Modeler* agent incrementally constructs candidate programs, while a *Validator* agent evaluates them by interacting with an ASP execution environment and produces feedback to guide further refinement. This interaction gives rise to an automated construct–validate loop that mirrors the workflow of human ASP modelers. The goal of ALM–ASP is not to outperform existing agentic systems, but to provide a principled framework for studying how agentic loops behave when applied to low-resource declarative formalisms such as ASP. To this aim, beyond the concrete system, we introduce a functional model of language agents equipped with tools and internal representations retained across interactions (*persistent state*), providing a formal foundation for describing agentic interaction with external systems. The model is intentionally minimal: it separates agent reasoning, environment actions, and persistent state, without enforcing correctness or termination. It captures tool-augmented agents that interleave LLM-based reasoning steps with task-specific actions, in the spirit of ReAct-style architectures (Yao et al. 2023). ALM–ASP is presented as a concrete instantiation of this model in the context of ASP.

Lastly, we evaluate ALM–ASP on a challenging subset

of CP–Bench (Michailidis, Tsouros, and Guns 2025) and on problems from three editions of the LP/CP Programming Contest. The experimental results show that ALM–ASP substantially improves syntactic validity and end-to-end correctness over general-purpose LLM baselines, and solves additional instances compared to the closest agentic alternative, CP–Agent (Szeider 2025b).

The main contributions of this work can be summarized as follows:

- We introduce a functional model of language agents equipped with tools and persistent state, suitable for formalizing agentic interaction with external systems.
- We define ALM–ASP, a multi-agent architecture obtained as an instantiation of this model for the automated generation and refinement of ASP programs from natural language specifications.
- We present an implementation of ALM–ASP based on a CLINGO-backed ASP environment and an empirical evaluation on challenging ASP modeling benchmarks.

2 Background

Notation. Let Σ be a fixed set of symbols, Σ^* be the set of finite sequences of symbols over Σ, called *strings*, and $\Sigma^{[*]}$ be the set of finite lists over Σ^*. The empty string and the empty list are denoted by ϵ and $[\,]$, respectively. The *concatenation* of two strings (or lists) A and B is denoted by $A + B$. With a mild abuse of notation, we also write $A + B$ when A is a list and B is a string, to denote the list obtained by appending B to A. Moreover, whenever concatenation involves a non-string element x, we assume the existence of a representation function mapping x to a string (for instance, via a unique identifier), and we apply concatenation to this representation.

2.1 Answer Set Programming

Answer Set Programming (ASP) (Lifschitz 2008) is a declarative programming paradigm for modeling and solving combinatorial search and optimization problems. In ASP, a problem is specified by means of a logic program whose intended solutions correspond to its *answer sets*, i.e., collections of atoms satisfying the rules of the program under the stable model semantics. ASP engines automatically process a given program and search for its answer sets, possibly subject to additional constraints such as optimization criteria or resource limits.

In this work, we treat ASP programs at an abstract level, focusing on their representation and the outcome of the solving process rather than on syntactic details. Accordingly, we assume that an ASP program is represented as a list $S \in \Sigma^{[*]}$ of strings, where each string is a logic rule. We model the execution of an ASP engine by means of a (partial) function

$$search : \Sigma^{[*]} \times \mathbb{N} \to \Sigma^*$$

where $search(S, b)$ denotes the result of invoking an ASP engine on the program represented by S with a given computational budget $b \in \mathbb{N}$, interpreted, e.g., as a time limit or a bound on computational resources. The output of $search$ is a string encoding engine feedback, such as an answer set, an indication that no answer set exists, budget exhaustion, or diagnostic information reporting syntactic (e.g., malformed rules) or semantic (e.g., unsafe rules) issues in the program.

Example 1. *Consider an ASP program S, represented by the following list of strings, where every rule corresponds to a string:*

```
node(1..3).  edge(1,2).  edge(2,3).
{color(N,C) : col(C)} = 1 :- node(N).
:- edge(X,Y), color(X,C), color(Y,C).
#show color/2.
```

Invoking the ASP engine as $search(S, b)$ may return diagnostic feedback, for instance "predicate `col/1` does not occur in any rule head," indicating an issue in the program. Invoking the ASP engine on S extended with the strings

```
col(r).  col(g).
```

may instead yield a feedback encoding the answer set

```
color(1,r). color(2,g). color(3,r).
```

provided that the computational budget b is sufficient.

This abstract view allows us to reason uniformly about ASP engine outcomes and feedback, independently of the concrete ASP system used, and is sufficient for defining the interaction between language agents and ASP-based reasoning tools in the remainder of the paper.

2.2 LLMs and Language Agents

Large Language Models (LLMs) are neural network models typically based on the Transformer architecture (Vaswani et al. 2017), designed to process and generate strings. At an abstract level, an LLM can be viewed as a stochastic function mapping an input string to an output string. LLMs are *inherently stateless* in the sense that information does not persist across independent invocations.

To address complex tasks that require multi-step reasoning, interaction with external systems, or iterative refinement, LLMs are often embedded within *language agents*. Specifically, a language agent is equipped with (i) an LLM, (ii) a *memory*, and (iii) an *environment* comprising a set of *tools* and a *persistent state* (Sumers et al. 2024). Intuitively, a language agent executes an iterative process that, in each intermediate step, calls an LLM to perform reasoning (such as abductive or practical reasoning, problem decomposition, and planning). At each step, the memory of the agent is updated. For example, it may be transformed by a reasoning step or augmented with observations returned by a tool invocation. The agent stops when a termination condition is met, which may be triggered by an explicit termination decision produced by the LLM, or by external criteria such as a predefined bound on the number of interaction steps. Upon termination, the agent returns an output string together with the final memory; for technical convenience, the resulting persistent state of the environment is also returned, although it is conceptually external to the agent.

3 Language Agents with Tools: A Formal Model for ALM–ASP

In this section, we introduce a formal model of language agents with tools that is general and independent of any specific reasoning formalism. We then show how the model can be instantiated to obtain ALM–ASP.

3.1 A Functional Model of Language Agents

Let $\mathcal{L}$ denote the class of (abstract) LLMs, modeled as (stochastic) functions

$$\begin{array}{rccc} llm: & \Sigma^* & \to & \Sigma^* \\ & X & \mapsto & Y \end{array}$$

which sample an output string Y given an input *prompt* X.

We start our formalization with the notion of a tool, which is essentially a function that the agent may invoke when proposed by the LLM. Let $\mathcal{T}$ denote the class of tools, modeled as functions

$$\begin{array}{rccc} tool: & \Sigma^* \times \Sigma^{[*]} & \to & \Sigma^* \times \Sigma^{[*]} \\ & (A, S) & \mapsto & (O, S') \end{array}$$

where A is the input argument, S denotes the current persistent state of the environment, O represents the feedback provided by the tool (e.g., the expected output of the tool, or diagnostic or error information), and S' is the persistent state resulting from the tool execution.

We now introduce a parameterized function to model the *reasoning* of an agent equipped with an $llm \in \mathcal{L}$ and a set $\mathbf{T} \subset \mathcal{T}$ of tools:

$$\begin{array}{rccc} decide^{\mathbf{T}}_{llm}: & \Sigma^* \times \Sigma^{[*]} & \to & (\mathbf{T} \cup \{\epsilon\}) \times \Sigma^* \\ & (X, M) & \mapsto & (t, A) \end{array}$$

Here, X denotes the current prompt provided to the agent (e.g., an external input or tool feedback). Essentially, $llm(M + X)$ is used to decide the *target* t and the corresponding argument A based on both X and its accumulated memory M: if $t \in \mathbf{T}$, the agent performs the call $t(\mathcal{A}, S)$, where S is the persistent state of the environment; if $t = \epsilon$, the agent terminates and reports the string A. Note that $decide^{\mathbf{T}}_{llm}$ abstracts from the internal reasoning of the agent. The LLM may be queried multiple times within a single decision step (e.g., to revise an initially unsuitable action), but only the resulting choice of action (tool invocation or termination) is exposed by the model.

Algorithm 1 shows the loop implemented by an agent equipped with $llm \in \mathcal{L}$ and $\mathbf{T} \subset \mathcal{T}$. We can then introduce $\mathcal{A}^{\mathbf{T}}_{llm}$, the parameterized class of language agents equipped with $llm \in \mathcal{L}$ and $\mathbf{T} \subset \mathcal{T}$, which defines functions of the following form:

$$\begin{array}{rccc} Agent^{\mathbf{T}}_{llm}: & \Sigma^* \times \Sigma^{[*]} \times \Sigma^{[*]} & \to & \Sigma^* \times \Sigma^{[*]} \times \Sigma^{[*]} \\ & (X, M, S) & \mapsto & (Y, M', S') \end{array}$$

Here, X is the input prompt, M the current memory, and S the current persistent state of the environment. As for the output of the above function, Y is the output string, while M' and S' denote the updated memory and persistent state, respectively.

Algorithm 1 $\text{Agent}^{\mathbf{T}}_{llm}$

Input: Prompt, Memory, State

1: $(X, M, S) \leftarrow (\text{Prompt}, \text{Memory}, \text{State})$
2: **repeat**
3: $\quad (t, A) \leftarrow decide^{\mathbf{T}}_{llm}(X, M)$
4: $\quad M \leftarrow M + X + t + A$
5: $\quad$ **if** $t \in \mathbf{T}$ **then**
6: $\quad\quad (X, S) \leftarrow t(A, S)$
7: $\quad$ **end if**
8: **until** $t = \epsilon$
9: **return** A, M, S

Example 2. *Consider an agent equipped with an LLM llm and a tool $t \in \mathcal{T}$ that supports extending an ASP program and obtaining engine feedback via the function search introduced in Section 2.1. Let the persistent state S represent a partial ASP program for a graph coloring problem (see Example 1), and let the initial memory M contain traces of previous interaction leading to S. If the agent receives as input a diagnostic feedback $X =$ "predicate `col/1` does not occur in any rule head," the decision function (line 3 of Algorithm 1) uses the output of $llm(M + X)$ to conclude that the tool t must be called:*

$$decide^{\{t\}}_{llm}(X, M) = (t, A),$$

where A is "`col(r). col(g).`" If the output of the tool is an answer set X' and the updated persistent state S', at the next iteration the agent may use

$$decide^{\{t\}}_{llm}(X', M + X + t + A) = (\epsilon, A''),$$

to terminate reporting the output string A'' together with the accumulated memory $M + X + t + A + X' + A''$ and the final program state S'.

3.2 Functional Instantiation for ALM–ASP

We now instantiate the functional model of language agents introduced in Section 3.1 to define *ALM–ASP*, a multi-agent architecture for generating Answer Set Programming (ASP) programs from natural language specifications. The instantiation is defined at the level of abstract agents and environment, and is independent of any concrete implementation choices, which are discussed in Section 4.

ALM–ASP consists of two interacting language agents, a *Modeler* and a *Validator*, which differ in their roles, available tools, and memory management policies. For simplicity of presentation, we assume that both agents are equipped with the same large language model $llm \in \mathcal{L}$, although the framework does not depend on this assumption and allows different models to be used.

Environment. Let $\mathbf{T}_{ASP} \subset \mathcal{T}$ be the finite set

$$\mathbf{T}_{ASP} = \{edit, inspect, solve\}$$

of tools operating over a persistent ASP program state $S \in \Sigma^{[*]}$. Each tool in $\mathbf{T}_{ASP}$ has the signature introduced in Section 3.1, namely

$$tool: \Sigma^* \times \Sigma^{[*]} \to \Sigma^* \times \Sigma^{[*]},$$

and the state S encodes the current ASP program under construction. The *editing tool* updates the ASP program state by applying an edit specified by its argument:

$$edit: \quad (A, S) \quad \mapsto \quad (O, S')$$

where A encodes an editing operation (e.g., adding, removing, or replacing a fragment of the ASP program), S' is the resulting persistent state, and O is a string reporting the outcome of the operation (e.g., success or a diagnostic message). The *inspection tool* returns a representation of the current ASP program without modifying the persistent state:

$$inspect: \quad (A, S) \quad \mapsto \quad (O, S)$$

where the argument A is ignored, and O is a string representation of the current program stored in S. The *solving tool* invokes the ASP solving process over the current program:

$$solve: \quad (A, S) \quad \mapsto \quad (search(S, A), S)$$

where the argument A is the allotted budget, the persistent state S is left unchanged, and $search(S, A)$ is the ASP solving feedback produced by executing the current program, such as answer sets or error messages.

Modeler. The Modeler agent is responsible for constructing candidate ASP programs incrementally. Formally, it is an instance of the class $\mathcal{A}_{llm}^{\mathbf{T}_{ASP}}$ and defines a function of the form

$$Modeler: \quad (X, M, S) \quad \mapsto \quad (Y, M', S')$$

where the input X corresponds to the natural language problem specification or feedback received during the ALM–ASP loop (see Algorithm 2), M is its memory, and S is the shared ASP program state. Following Algorithm 1, the Modeler iteratively applies the decision function $decide_{llm}^{\mathbf{T}_{ASP}}$ from Section 3.1. Based on a reasoning step performed by llm, this function determines whether the next step is an environment action, realized by invoking a tool in $\mathbf{T}_{ASP}$, or termination.

Validator. The Validator agent assesses the alignment of a candidate ASP program with the original natural language specification. Formally, it is an instance of $\mathcal{A}_{llm}^{\mathbf{T}_V}$, where $\mathbf{T}_V = \{inspect, solve\} \subseteq \mathbf{T}_{ASP}$ is a restricted set of tools; essentially, the Validator is granted read-only access to the shared ASP program state and execution capabilities, but cannot modify the program. Given a candidate program S, it analyzes the output of $search(S, \cdot)$, e.g., an answer set or error messages, together with the original specification, and produces either an acceptance decision $\top$ or a feedback string highlighting potential syntactic or semantic inconsistencies. The Validator does not guarantee correctness of the program, but acts as a semantic filter guiding further refinement.

Interaction and Termination. ALM–ASP operates as an iterative interaction between the two agents, as summarized in Algorithm 2. Starting from an initial natural language prompt, the Modeler incrementally constructs a candidate ASP program by operating on the persistent state S (initially empty). At each iteration, the Validator evaluates the current program against the original specification and produces feedback, which is used as input X for the subsequent Modeler iteration. The overall process terminates when the Validator produces a terminating decision, corresponding to $t = \top$ in the abstract model, yielding the final ASP program together with the resulting memory and persistent state. Note that the two agents adopt different memory management policies. The Modeler is equipped with a long-term memory M, which is initialized once at the beginning of the ALM–ASP execution with the agent role description (ModRole) and is never cleared during the iterative refinement loop. In contrast, the Validator is equipped with a short-term memory, which is initialized with its role description (ValRole) at each invocation and discarded after producing its validation outcome. This design reflects the different roles of the agents, with the Modeler accumulating contextual information across iterations and the Validator performing stateless checks on individual candidate programs. This instantiation fully adheres to the functional model of language agents introduced in Section 3.1. In the next section, we describe how the ALM–ASP instantiation is concretely realized.

Algorithm 2 ALM–ASP

Input: Prompt, ModRole, ValRole

1: $(X, M, S) \leftarrow (\text{Prompt}, [\text{ModRole}], [\,])$
2: **repeat**
3: $\quad(\cdot, M, S) \leftarrow Modeler(X, M, S)$
4: $\quad(X, \cdot, \cdot) \leftarrow Validator(\text{Prompt}, [\text{ValRole}], S)$
5: **until** $X = \top$ // the Validator agent accepted S
6: **return** S

Example 3. *Consider the graph coloring problem introduced in Example 1. The ALM–ASP loop starts with memory* $M = [\text{ModRole}]$ *and program state* $S = [\,]$*, and invokes the Modeler agent with the natural language specification (lines 1–3 of Algorithm 2). During its execution, the Modeler incrementally constructs a candidate ASP program. For instance, it may first produce and store in* S *the partial program from Example 1 but with the choice rule replaced by*

```
{color(N,C) : col(C)} <= 1 :- node(N).
```

by means of a reasoning step followed by the invocation of the edit *tool. A subsequent reasoning step may conclude that invoking the* solve *tool is required in order to obtain feedback from the ASP engine. The resulting feedback may indicate that the predicate* `col/1` *is used but not defined. Based on this feedback, the Modeler can generate the missing* `col/1` *rules and extend the program state by invoking the* edit *tool again. A further call to* solve *then yields an answer set, after which the Modeler terminates.*

The Validator is then invoked with memory [ValRole]*, the current ASP program state, and the original natural language specification. Using the* inspect *tool and a reasoning step, the Validator analyzes the encoding with respect to the specification. In this case, it may detect that the choice*

rule does not enforce that every node is assigned a color. The Validator therefore terminates by producing a feedback message describing this issue. This feedback becomes the new input for the Modeler agent, which can then replace the incorrect rule with one enforcing exactly one color per node, and continue the ALM–ASP refinement loop.

Relation to Existing Agentic Systems. The functional model introduced in Section 3.1 and instantiated above is intentionally abstract and not specific to ASP. In particular, CP–Agent (Szeider 2025b) admits a natural interpretation within our formal model: Its use of a persistent Python execution environment, accessed through tools such as `python_exec` for code execution and `todo_write` for task management, corresponds to environment actions over a shared program state. The agent loop can thus be viewed as an instance of iterative decision-making over reasoning and environment actions. This observation is purely descriptive and does not assume that CP–Agent was designed with this formalization in mind. Rather, it serves to illustrate the generality of the proposed model and its applicability beyond ASP-specific settings.

4 Implementation of ALM–ASP

In this section, we describe the concrete implementation of ALM–ASP as an instantiation of the functional model introduced in Section 3.2. The goal of this section is not to redefine the behavior of agents or their interaction, fully specified by the formal model, but to illustrate how such a model can be concretely realized and executed in practice.

Overview. Figure 1 provides an overview of the ALM–ASP architecture and its execution flow. The Modeler and the Validator agents share a persistent environment that stores the ASP program under construction. Their interaction follows exactly the agentic loop formalized in Section 3.2, including memory management, tool usage, and termination conditions. At the implementation level, agent execution and control flow are realized using a stateful graph-based orchestration framework, which enables cyclic interactions and controlled termination, and is instantiated in our prototype using LangGraph (LangChain 2024). Each agent is initialized with a role-specific prompt template that encodes high-level behavioral instructions and serves as the starting context for interaction with the underlying LLM.

Environment. The environment of ALM–ASP is realized by an ASP engine that maintains a persistent program state S representing the ASP program under construction. The environment exposes a set of concrete tools operating over this state, implementing the abstract tool interface introduced in Section 3.1. In the current implementation, the ASP engine is instantiated using clingo (Gebser et al. 2019). Concretely, the environment provides the following tools:

- t_1: `add_rules`, which inserts a new rule or block of rules into the program;

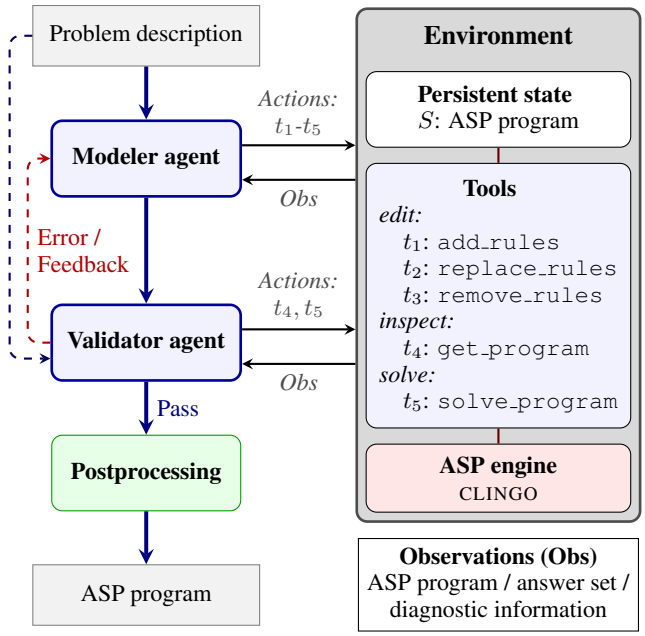

Figure 1: The ALM–ASP workflow. The Modeler agent iteratively constructs the ASP program state within the MCP server. The Validator agent checks compliance with the specification by inspecting and executing the program, and provides feedback to guide further refinement.

- t_2: `replace_rules`, which replaces an existing rule or block of rules identified by a given identifier;
- t_3: `remove_rules`, which removes a rule or block of rules from the program;
- t_4: `get_program`, which returns a representation of the current ASP program state;
- t_5: `solve_program`, which invokes the ASP engine on the current program with a specified computational budget.

These tools realize the three classes of operations defined in Section 3.2: *edit* (t_1–t_3), *inspect* (t_4), and *solve* (t_5). The environment and its tools are exposed to the agents through a standardized communication interface, instantiated in our implementation using the Model Context Protocol (MCP) (Anthropic 2024; Szeider 2025a).

Execution and Postprocessing. The two agents are instantiated and executed exactly according to the functional model defined in Section 3.2. No additional agent behavior is introduced at the implementation level. Execution terminates when the Validator signals acceptance of the current program or when a predefined maximum number of iterations is reached. After acceptance, a final postprocessing step is applied to the resulting ASP program. This step performs a lightweight analysis to identify and remove redundant or unused rules, improving readability and conciseness. The resulting program is returned as the final output of ALM–ASP.

5 Experimental Evaluation

We conducted a comparative evaluation to analyze how our ALM–ASP and the CP–Agent behave when tasked with generating ASP encodings, and to identify which design choices enable error detection and recovery in a low-resource declarative setting. The experiments were designed to evaluate whether the agentic loop helps in solving complex problems where the standard prompting fails, and to assess how well the system scale to programming contest problems.

5.1 Benchmarks and Task Selection

We evaluate ALM–ASP on two complementary sets of benchmarks designed to assess both (i) general constraint modeling capabilities from natural language and (ii) robustness on competitively challenging ASP problems. Specifically, we consider a selected subset of the CP–Bench dataset (Michailidis, Tsouros, and Guns 2025) and problems from three editions of the LP/CP Programming Contest[1].

CP–Bench Hard Subset. CP–Bench is a collection of 101 combinatorial problems introduced for evaluating LLM-based constraint modeling. While the dataset offers broad coverage, a substantial fraction of its instances can be solved by LLMs via direct prompting, without requiring execution, feedback, or iterative correction. Such instances are not informative for evaluating agentic architectures, since the benefits of tool use and refinement loops do not manifest when a single-shot solution suffices. Our focus is therefore on problems that cannot be reliably solved by the LLM alone and instead require repeated cycles of execution, diagnosis, and repair. These instances represent precisely the setting in which an agentic loop is meaningful, as they necessitate interaction with the solver and systematic handling of modeling errors. To isolate this subset, we adopt a data-driven selection strategy based on the execution traces of CP–Agent, selecting problems that require the highest number of iterative tool interactions (specifically, calls to the `python_exec` and `todo_write` tools). Based on this analysis, we select the 10 problems with the highest total number of tool interactions. We refer to this subset as **CP–Bench Hard**, and report the associated statistics in Table 1.

LP/CP Programming Contest Problems. To assess generalization beyond curated benchmarks, we additionally evaluate ALM–ASP on problems from the LP/CP Programming Contest, using the 2021, 2022, and 2023 editions. These problems are explicitly designed for expert human programmers and pose significant challenges for automated modeling systems. Contest problems are characterized by two key properties:

- **Syntactic complexity.** Solutions often require advanced ASP constructs, including aggregates, multi-level optimization statements, and symmetry-breaking constraints.
- **Out-of-distribution nature.** Unlike standard benchmark problems (e.g., graph coloring or N-Queens), contest problems are less likely to appear in LLM training data, reducing the risk of memorization and better testing genuine modeling capabilities.

[1]See contest editions 2021, 2022, and 2023 from https://lpcp-contest.github.io/.

Problem	python_exec	todo_write	in/out
autoref	23	0	581/18
crossfig	17	0	414/14
guard_app	10	5	254/ 7
pack_rect	8	8	263/ 7
subset100	8	7	239/ 6
add_corn	8	6	247/ 5
five_flr	7	7	226/ 6
movie_sch	7	6	215/ 5
hardy	6	6	208/ 5
climb_st	6	6	196/ 5

Table 1: The **CP–Bench Hard** subset. Problems are ranked by the number of tool interactions and tokens consumption in thousands (in/out) required by CP–Agent, serving as a proxy for modeling difficulty.

We consider the full problem sets from the selected editions. The 2021 and 2022 contests each include 5 problems with 10 test instances per problem (50 instances per year), while the 2023 contest includes 5 problems with 5 instances each (25 instances total). A solution is considered correct only if it successfully solves all test instances associated with a given problem. In contrast to CP–Bench, which emphasizes standard constraint formulations, the LP/CP contest problems serve as a stress test for ASP-specific modeling skills, requiring the generation of complete, robust encodings comparable to those produced by expert human participants.

5.2 Experimental Setup

All experiments were conducted using `GPT-5` (OpenAI 2025a) as the underlying LLM for all agentic systems, setting temperature 0 for higher stability. We additionally consider a local open-weights model, `gpt-oss:120b` (OpenAI 2025b), as a baseline for non-agentic direct prompting. Preliminary experiments showed that `gpt-oss:120b` is not sufficiently reliable in complex tool-use scenarios involving iterative reasoning and execution, and is therefore not used for agentic configurations. We compare ALM–ASP against the following baselines:

- **CP–Agent (ASP Mode):** A general-purpose coding agent that follows a ReAct-style reasoning-and-execution loop and is configured to generate ASP encodings by interacting with the CLINGO Python API.
- **Direct Prompting (No Agent):** A standard meta-prompting approach without tool access or feedback loops, where a single structured prompt provides task instructions and output format constraints, and the LLM generates the ASP encoding in one pass. Note that, unlike agentic approaches, direct prompting does not allow intermediate execution, error inspection, or pro-

gram revision. We evaluate this using both `GPT-5` and `gpt-oss:120b`.

ALM–ASP was configured with a maximum of 5 Modeler–Validator refinement iterations. For each system, we measure success rate (correct solution), total execution time, number of tool invocations, and total token consumption.

5.3 Results on the CP–Bench Hard Subset

We exclude direct prompting from this analysis, as these instances were explicitly selected to require iterative refinement. Since CP–Agent previously achieved a 100% success rate on the full CP–Bench when targeting Python/CPMpy, the solvability of these problems by LLM-based agents is not in question. Our focus here is instead on *robustness across formalisms*, namely whether agentic systems can reliably generate *valid ASP encodings*, a language with substantially lower representation in current training corpora.

Table 2 reports the results. ALM–ASP successfully solved all 10 instances, whereas CP–Agent failed on the `crossfig` problem. This difference highlights the benefit of explicitly separating program construction and validation when operating in ASP. The `crossfig` instance is particularly illustrative. ALM–ASP required two Modeler–Validator iterations to converge: an initial encoding was produced and executed, the Validator identified a modeling defect, and the Modeler corrected the program based on this feedback in the subsequent iteration. While this process incurred higher token consumption, tool calls, and execution time, these costs correspond to substantive program revisions rather than redundant execution.

For the remaining 9 instances, ALM–ASP converged in a single iteration, often with lower token consumption than CP–Agent. For example, on `guard_app`, ALM–ASP required 43.7k tokens compared to 156k for CP–Agent. This suggests that the combination of ASP-specific tool abstractions and role-specialized agents can improve modeling efficiency even before validation-driven refinement is triggered.

Problem	Agent	Solved	Tool Calls	Tokens	Time	Iter.
autoref	CP–Agent	✓	7	121.4k	314.1s	–
	ALM–ASP	✓	10	76.9k	592.1s	1
crossfig	CP–Agent	×	8	220.6k	1023.1s	–
	ALM–ASP	✓	31	603.3k	3146.9s	2
guard_app	CP–Agent	✓	11	156.0k	481.8s	–
	ALM–ASP	✓	6	43.7k	330.8s	1
pack_rect	CP–Agent	✓	10	161.0k	300.5s	–
	ALM–ASP	✓	15	147.7k	619.3s	1
subset100	CP–Agent	✓	13	177.3k	452.3s	–
	ALM–ASP	✓	14	102.5k	416.5s	1
add_corn	CP–Agent	✓	12	161.2k	668.4s	–
	ALM–ASP	✓	6	37.5k	221.1s	1
five_flr	CP–Agent	✓	14	190.6k	486.2s	–
	ALM–ASP	✓	6	41.7k	303.3s	1
movie_sch	CP–Agent	✓	16	234.7k	804.5s	–
	ALM–ASP	✓	7	56.9k	373.3s	1
hardy	CP–Agent	✓	25	402.0k	1270.1s	–
	ALM–ASP	✓	31	181.7k	1719.7s	1
climb_st	CP–Agent	✓	10	134.3k	356.1s	–
	ALM–ASP	✓	10	74.3k	491.7s	1

Table 2: Results on the CP–Bench Hard subset comparing ALM–ASP and CP–Agent in ASP mode

5.4 Results on the LP/CP Programming Contest

Table 3 reports the number of solved instances together with the official competition rankings. Non-agentic baselines perform poorly across all editions. Direct prompting with `GPT-5` solved only 1 (*Crazy Frog*), 3 (*Fortress*, *Beer Jugs*, and *Slant Puzzle*), and 1 (*Student Exercises*) problems in the 2021, 2022, and 2023 editions, respectively. These successes are confined to problems that can be addressed without solver interaction, plausibly due to the presence of familiar solution patterns in the prior knowledge of the LLM. The open-weights model `gpt-oss:120b` failed to solve any instances in the 2021 and 2022 editions and solved only the *Student Exercises* problem in 2023. These results indicate that, in the absence of iterative execution and repair, LLMs struggle to reliably model complex ASP problems.

ALM–ASP achieves competitive performance in this setting. In the 2021 edition, it solved two problems, doubling the performance of CP–Agent. In the 2022 edition, ALM–ASP solved four problems, matching the performance of the *tuw* team, while CP–Agent failed on two instances of the *RoBeep!* problem. Although a gap remains with respect to the contest winners (e.g., *Picat* and *Bule*), who achieved perfect scores, these results place ALM–ASP within the range of competitive human solutions. It is worth noting that the contest setting differed across editions: the 2021 and 2022 editions were conducted online, with human participants given 24h and 12h to submit their solutions, respectively, whereas the 2023 edition was held on-site with a 2h time limit for human teams. In this more time-constrained setting, both agentic approaches outperformed all human teams, with CP–Agent achieving the highest overall score.

To further analyze the operational differences between agentic approaches, Table 4 details performance on problems solved by at least one agent. ALM–ASP successfully solved several instances that were challenging for CP–Agent, such as *RoBeep!* and *Buttons & Scissors*. On commonly solved instances, ALM–ASP typically required fewer tokens, indicating a more concise encoding process, while execution times remained comparable.

The cost of the agentic loop is analyzed in Figure 2, which reports token consumption, tool usage, and execution time across contest problems. A consistent trade-off emerges: ALM–ASP achieves higher token efficiency, whereas CP–Agent generally exhibits faster execution. For example, on *Student Exercises* (2023), ALM–ASP reduced token usage by more than 80% compared to CP–Agent (77.3k vs. 408.3k). However, the dual-agent architecture introduces additional runtime overhead due to Modeler–Validator interactions. This effect is particularly visible in outlier cases, such as the 2021 Problem 3, where ALM–ASP entered a

2021 Edition		2022 Edition		2023 Edition	
Team Name	**Solved**	**Team Name**	**Solved**	**Team Name**	**Solved**
Picat (Winner)	50/50	*Bule (Winner)*	50/50	**CP–Agent**	**15/25**
Sebastiano	26/50	*Triple Negation*	50/50	**ALM–ASP**	**10/25**
ALM–ASP	**20/50**	**ALM–ASP**	**40/50**	*SpaghettiLP (Winner)*	10/25
Wolfs31	11/50	*tuw*	40/50	*aaunical*	10/25
CP–Agent	**10/50**	**CP–Agent**	**38/50**	*NotYet*	10/25
GPT-5	10/50	*GPT-5*	30/50	*GPT-5*	5/25
ASPirant	6/50	*PicatBBZ*	30/50	*GPT-OSS:120b*	5/25
One4All	4/50	*The Two Marateers*	10/50	*ASP lovers*	0/25
EyFiBo	4/50	*Ch'ti CP*	0/50		
GPT-OSS:120b	0/50	*Fodor*	0/50		
		GPT-OSS:120b	0/50		

Table 3: Number of solved instances in the LP/CP Programming Contests (2021–2023) by agentic systems and human teams

Problem	Agent	Solved	Tokens	Tool Calls	Time
2021 Edition					
P1 (Crazy Frog Puzzle)	CP–Agent	✓	228.0k	14	880.7s
	ALM–ASP	✓	95.3k	8	920.3s
P2 (Buttons & Scissors)	CP–Agent	×	225.3k	12	1026.3s
	ALM–ASP	✓	225.1k	17	1671.1s
2022 Edition					
P1 (Fortress)	CP–Agent	✓	213.8k	10	855.1s
	ALM–ASP	✓	155.4k	14	695.6s
P2 (Beer jugs)	CP–Agent	✓	484.3k	19	980.5s
	ALM–ASP	✓	250.8k	19	1142.9s
P4 (RoBeep!)	CP–Agent	×	726.4k	23	1053.5s
	ALM–ASP	✓	245.2k	15	1581.0s
P5 (Slant Puzzle)	CP–Agent	✓	216.5k	11	707.3s
	ALM–ASP	✓	115.2k	11	669.5s
2023 Edition					
P1 (Students Exercises)	CP–Agent	✓	408.3k	15	711.6s
	ALM–ASP	✓	77.3k	9	1062.9s
P3 (Street Builder)	CP–Agent	✓	127.4k	6	542.9s
	ALM–ASP	×	109.2k	9	1109.0s
P4 (Reduce the Sequence)	CP–Agent	✓	517.8k	20	1406.1s
	ALM–ASP	✓	304.4k	21	1766.2s

Table 4: Performance metrics on selected LP/CP Programming Contest problems solved by at least one agentic system

prolonged refinement loop, consuming approximately 877k tokens and 37 tool calls. Such cases illustrate the cost of validation-driven retries when convergence is slow.

Overall, these results confirm that tool-augmented agentic loops are crucial for tackling complex ASP modeling tasks, enabling autonomous error detection and repair where direct prompting fails. At the same time, they highlight that ASP remains a challenging target formalism for current LLMs: unlike widely used languages such as Python, low-resource declarative languages continue to expose the limits of general-purpose language models.

6 Related Work

ALM–ASP lies at the intersection of neuro-symbolic reasoning, Answer Set Programming (ASP), and agentic program synthesis with solver-grounded feedback. A central challenge addressed by this line of research is overcoming the limitations of pure LLMs on complex reasoning and modeling tasks, where unverified outputs often result in syntactic or semantic inconsistencies.

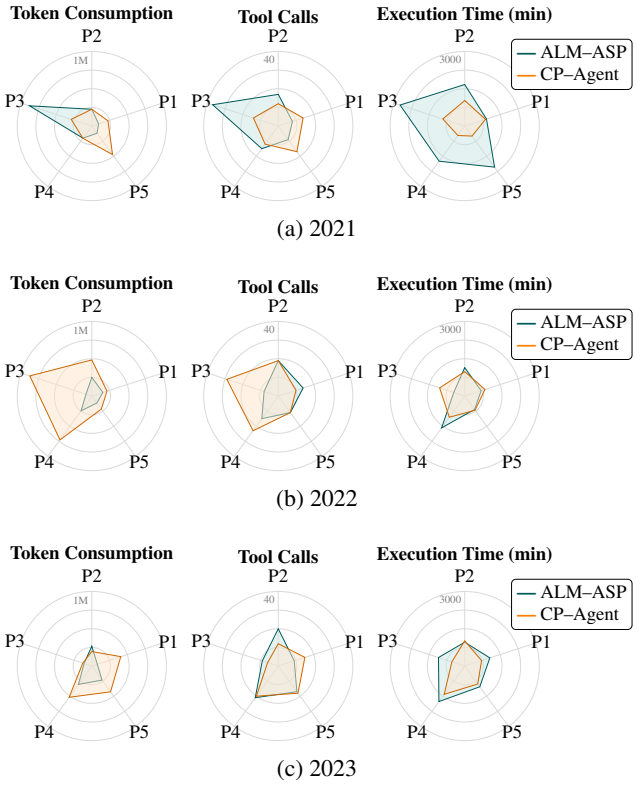

Figure 2: Resource usage comparison between ALM–ASP and CP–Agent on LP/CP Programming Contest problems

Neuro-symbolic ASP pipelines. Neuro-symbolic approaches combine the flexibility of neural models with the rigor of symbolic reasoning. A common decomposition strategy employs an LLM as a semantic parser and a symbolic solver as a reasoning backend. Early work such as the [LLM]+ASP framework (Yang, Ishay, and Lee 2023) demonstrated that LLMs can effectively translate natural language sentences into atomic ASP facts, which are then processed by an ASP engine equipped with background knowledge. This pipeline showed strong generalization across tasks without retraining, but focuses on fact extraction rather than full program synthesis.

Building on this idea, LLMASP (Alviano et al. 2024b; Alviano et al. 2025) addresses structured data extraction and query interpretation by combining LLMs and ASP. LLMASP emphasizes precise control over LLM output via explicitly structured prompts (e.g., configuration files defining predicates, terms, and formats), ensuring faithful translation of natural language into relational facts. The extracted facts are then combined with an ASP knowledge base for reasoning. While effective for controlled data extraction, these approaches do not aim at generating complete ASP encodings.

Iterative refinement and solver feedback. To mitigate the brittleness of one-shot generation, several frameworks introduce solver-driven feedback loops. LOGIC-LM (Pan et al. 2023) formalized a pipeline consisting of problem formulation, symbolic reasoning, and result interpretation, highlighting the importance of validating intermediate representations. DSPy (Wang, Sun, and Kuhn 2024) further demonstrated the effectiveness of iterative refinement guided by solver feedback, achieving significant accuracy gains on reasoning tasks such as spatial reasoning by repeatedly generating and correcting ASP predicates. These systems show that solver diagnostics are a critical component for improving logical consistency, but they typically operate within fixed pipelines rather than autonomous agentic loops.

ASP program synthesis from natural language. Other approaches focus explicitly on generating ASP programs. Ishay *et al.* (Ishay, Yang, and Lee 2023) proposed a three-stage pipeline (constants, predicates, rules) for translating natural language logic puzzles into ASP, observing that LLMs often produce programs containing simple, correctable errors. NL2ASP (Santana, Kareem, and Ricca 2024) adopts a different strategy by translating natural language into a controlled natural language (CNL) using neural machine translation, and subsequently converting the CNL into ASP using tools such as CNL2ASP (Caruso et al. 2024). These approaches rely on multi-stage, feed-forward pipelines and lack persistent state or autonomous repair mechanisms.

Agentic program synthesis and tool use. Recent work has highlighted the importance of augmenting LLMs with external tools for complex, multi-step tasks (Shen 2024). The ReAct framework (Yao et al. 2023) interleaves reasoning steps with tool invocations and observations, enabling LLMs to iteratively plan, execute, and correct their outputs. This paradigm has proven particularly effective for program synthesis and debugging tasks. In constraint programming, CP–Agent (Szeider 2025b), a general-purpose coding agent equipped with a persistent execution environment, could solve all instances of the CP–Bench benchmark (Michailidis, Tsouros, and Guns 2025). Although CP–Agent was not originally designed for ASP, its implementation supports the CLINGO Python API, enabling a direct comparison in the ASP setting. CP–Agent employs a single-agent ReAct loop and relies on execution-based debugging, without explicit separation between generation and validation roles.

Positioning of ALM–ASP. In contrast to prior neuro-symbolic pipelines and single-agent ReAct systems, ALM–ASP targets the direct synthesis of complete ASP encodings through a formally specified, multi-agent architecture. The system integrates two specialized agents, operating over a persistent ASP environment with engine-backed feedback. Unlike fact-level extraction approaches (e.g., LLMASP) or feed-forward translation pipelines (e.g., NL2ASP), ALM–ASP supports autonomous error detection and repair at the level of full programs. Moreover, the interaction between agents and tools is grounded in a functional model of language agents, providing a principled abstraction that is independent of concrete implementation choices.

7 Conclusion

In this work, we introduce a functional model of language agents equipped with tools and persistent state, suitable for formalizing agentic interaction with external systems. The aim is to formally characterize current and future agent-based systems, enabling a clear comparison between them with respect to their modules, defined as agents and shared resources. Through our functional model, we propose and formally define ALM–ASP, a multi-agent tool-augmented architecture for the automated modeling of problems in ASP. By integrating a persistent solver environment with a dedicated Modeler–Validator interaction loop, ALM–ASP enables iterative refinement of ASP encodings grounded in deterministic solver feedback. Our experimental results highlight a critical dichotomy in neuro-symbolic AI. While recent approaches, such as CP–Agent achieve near-perfect performance in constraint programming tasks using Python, our findings show that this level of robustness does not directly carry over to ASP. Indeed, when restricted to the ASP formalism, even sophisticated agentic loops exhibit substantially greater difficulty compared to Python-based constraint modeling. We attribute this gap primarily to representation bias in the training data of current large language models. Python, as one of the most widely used programming languages, is associated with a vast ecosystem of examples and libraries for constraint modeling (e.g., OR-Tools, PuLP), which are well represented in pretraining corpora. In contrast, ASP remains a niche academic formalism with limited exposure in large-scale web data, resulting in weaker probabilistic intuitions for its syntax and modeling idioms.

Nevertheless, the performance of ALM–ASP confirms that a specialized architecture with explicit validation roles can partially mitigate the knowledge gap. The Validator agent consistently identifies syntactic and semantic defects that are likely to persist in single-agent or non-agentic approaches. Simultaneously, our results indicate that the effectiveness of agentic loops depends critically on the alignment between the target formalism and the pretraining distribution of the model. Indeed, our approach provides a higher level of error filtering; but it is not infallible, and the produced encoding still requires human verification.

Future research should therefore explore: (1) fine-tuning LLMs on curated ASP corpora to improve formal-language competence, allowing the use of "smaller" models for ASP code generation within the Modeler; (2) the use of intermediate representations, such as Controlled Natural Languages (CNLs), allowing natural language specifications to be translated into structured forms before deterministic compilation into ASP, as proposed by Caruso et al. (Caruso et al. 2024); and (3) creation of large libraries of reusable ASP templates (Alviano et al. 2024a), providing documented code snippets enabling retrieval-based grounding and reduce syntactic and modeling errors. Such approaches may shield language models from strict syntactic requirements while preserving the expressive power of declarative reasoning.

Acknowledgments

This work was supported by the Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); under PN RIC project ASVIN "Assistente Virtuale Intelligente di Negozio" (CUP B29J24000200005); by Regione Calabria NextGenGuides "Innovazione Tecnologica per le Guide Turistiche del Futuro tramite l'ausilio di tecnologie innovative AI e sistemi di geolocalizzazione avanzata" (CUP J39I24002040005); by MOZART "Modelli e tecniche di mOdernizzazione di applicaZioni e data silos a supporto di clienti esterni, field force e impiegati di backoffice basati su AI GeneRativa e apprendimento auTomatico" (CUP J49I24001740005); by the LAIA lab (part of the SILA labs); by SIGENERA "Large Language Models (LLM) per il controllo di SIstemi informativi, la GENERazione automatica di testo, e l'accesso a basi di conoscenzA" (CUP J29I24001770005); by the Austrian Science Fund (FWF) projects 10.55776/PAT1599524 and 10.55776/COE12; and by the Austrian Research Promotion Agency (FFG) 930480ATRIA (Eureka labeled ATRIA project). Mario Alviano is a member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

AI Declaration

Large language models are used as components of the evaluated systems. All experimental design choices, analysis, and conclusions are the author's responsibility.

References

Alviano, M.; Ianni, G.; Pacenza, F.; and Zangari, J. 2024a. Rethinking Answer Set Programming Templates. In Gebser, M., and Sergey, I., eds., *Practical Aspects of Declarative Languages - 26th International Symposium, PADL 2024, London, UK, January 15-16, 2024, Proceedings*, Lecture Notes in Computer Science, 82–99. Springer.

Alviano, M.; Scudo, F. L.; Grillo, L.; and Reiners, L. A. R. 2024b. Answer Set Programming and Large Language Models Interaction with YAML: Second Report. In *Workshop on Symbolic and Neuro-Symbolic Architectures for Intelligent Robotics Technology (SYNERGY)*, volume 3876 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Alviano, M.; Grillo, L.; Scudo, F. L.; and Reiners, L. A. R. 2025. Integrating Answer Set Programming and Large Language Models for Enhanced Structured Representation of Complex Knowledge in Natural Language. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, 4330–4338. ijcai.org.

Anthropic. 2024. Model Context Protocol: A Standard for AI Sytem Integration. Accessed: 2025-10-28.

Bang, Y.; Cahyawijaya, S.; Lee, N.; Dai, W.; Su, D.; Wilie, B.; Lovenia, H.; Ji, Z.; Yu, T.; Chung, W.; Do, Q. V.; Xu, Y.; and Fung, P. 2023. A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. In *Proceedings of the International Joint Conference on Natural Language Processing (Volume 1: Long Papers) (IJCNLP)*, 675–718. Association for Computational Linguistics.

Caruso, S.; Dodaro, C.; Maratea, M.; Mochi, M.; and Riccio, F. 2024. CNL2ASP: Converting Controlled Natural Language Sentences into ASP. *Theory Pract. Log. Program.* 24(2):196–226.

Cohn, A. G. 2023. An Evaluation of ChatGPT-4's Qualitative Spatial Reasoning Capabilities in RCC-8. *CoRR* abs/2309.15577.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* 19(1):27–82.

Gelfond, M., and Lifschitz, V. 1990. Logic Programs with Classical Negation. In *Logic Programming, Proceedings of the Seventh International Conference, Jerusalem, Israel, June 18-20, 1990*, 579–597. MIT Press.

Ishay, A.; Yang, Z.; and Lee, J. 2023. Leveraging large language models to generate answer set programs. In *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023*, 374–383.

LangChain. 2024. LangGraph. https://github.com/langchain-ai/langgraph. Accessed 2026-01.

Lifschitz, V. 2008. What Is Answer Set Programming? In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 1594–1597. AAAI Press.

Liu, H.; Ning, R.; Teng, Z.; Liu, J.; Zhou, Q.; and Zhang, Y. 2023. Evaluating the Logical Reasoning Ability of ChatGPT and GPT-4. *CoRR* abs/2304.03439.

Marek, V. W., and Truszczynski, M. 1999. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm - A 25-Year Perspective*, Artificial Intelligence. Springer. 375–398.

Michailidis, K.; Tsouros, D.; and Guns, T. 2025. CP-Bench: Evaluating Large Language Models for Constraint Modelling. In *ECAI 2025 - 28th European Conference on Artificial Intelligence, 25-30 October 2025, Bologna, Italy - Including 14th Conference on Prestigious Applications of Intelligent Systems (PAIS 2025)*, Frontiers in Artificial Intelligence and Applications, 861–868. IOS Press.

Minaee, S.; Mikolov, T.; Nikzad, N.; Chenaghlu, M.; Socher, R.; Amatriain, X.; and Gao, J. 2024. Large Language Models: A survey. *CoRR* abs/2402.06196.

Niemelä, I. 1999. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Ann. Math. Artif. Intell.* 25(3-4):241–273.

OpenAI. 2025a. GPT-5 system card. https://openai.com/index/gpt-5-system-card/. Accessed 2026-01.

OpenAI. 2025b. GPT-OSS-120b Model Documentation. https://platform.openai.com/docs/models/gpt-oss-120b. Accessed 2026-01.

Pan, L.; Albalak, A.; Wang, X.; and Wang, W. Y. 2023. Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, Findings of ACL, 3806–3824. Association for Computational Linguistics.

Santana, M. A. B.; Kareem, I.; and Ricca, F. 2024. Towards Automatic Composition of ASP Programs from Natural Language Specifications. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*, 6198–6206. ijcai.org.

Shen, Z. 2024. LLM with tools: A survey. *CoRR* abs/2409.18807.

Sumers, T. R.; Yao, S.; Narasimhan, K.; and Griffiths, T. L. 2024. Cognitive architectures for language agents. *Transactions on Machine Learning Reseach* 2024.

Szeider, S. 2025a. Bridging language models and symbolic solvers via the model context protocol. In Berg, J., and Nordström, J., eds., *28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)*, volume 341 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 30:1–30:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Szeider, S. 2025b. CP-Agent: Agentic Constraint Programming. *CoRR* abs/2508.07468.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 5998–6008.

Wang, R.; Sun, K.; and Kuhn, J. 2024. A Pipeline of Neural-Symbolic Integration to Enhance Spatial Reasoning in Large Language Models. *CoRR* abs/2411.18564.

Yang, Z.; Ishay, A.; and Lee, J. 2023. Coupling Large Language Models with Logic Programming for Robust and General Reasoning from Text. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, Findings of ACL, 5186–5219. Association for Computational Linguistics.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K. R.; and Cao, Y. 2023. React: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.