Bridging Language Models and Symbolic Solvers via the Model Context Protocol

Stefan Szeider
Algorithms and Complexity Group, TU Wien, Austria

Abstract

This paper presents the *MCP Solver*, a system that bridges large language models with symbolic solvers through the Model Context Protocol (MCP). The system includes a *server* and a *client* component. The server provides an interface to constraint programming (via MiniZinc Python), propositional satisfiability and maximum satisfiability (both via PySAT), and SAT modulo Theories (via Python Z3). The client contains an agent that connects to the server via MCP and uses a language model to autonomously translate problem statements (given in English) into encodings through an incremental editing process and runs the solver. Our experiments demonstrate that this neurosymbolic integration effectively combines the natural language understanding of language models with robust solving capabilities across multiple solving paradigms.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Knowledge representation and reasoning; Computing methodologies → Natural language processing; Theory of computation → Logic and verification; Software and its engineering → Software notations and tools; Computing methodologies → Artificial intelligence

Keywords and phrases Large Language Models, Agents, Constraint Programming, Satisfiability Solvers, Maximum Satisfiability, SAT Modulo Theories, Model Context Protocol

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.30

Supplementary Material *Software*: `https://github.com/szeider/mcp-solver`

Funding This work was supported by Austrian Science Fund (FWF) grants 10.55776/P36420 and 10.55776/COE12.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across diverse natural language tasks yet exhibit fundamental limitations in formal logical reasoning [6, 11, 12, 15, 18, 19, 22, 24]. In this work, we leverage the robust logical deduction capabilities of symbolic solvers to overcome these limitations. This enables language models to perform complex reasoning tasks with greater reliability. We present the *MCP Solver*, which uses the recently introduced *Model Context Protocol (MCP)* [2] for bridging language models with four complementary solving paradigms.

1. *MiniZinc* [14, 17]: a high-level constraint modeling language that supports global constraints, optimization, and diverse problem domains.
2. *PySAT* [8]: a Python interface to SAT solvers that enables propositional constraint modeling using CNF (Conjunctive Normal Form). The system supports various SAT solvers (including Glucose, Cadical, and Lingeling), with helpers for cardinality constraints.
3. *MaxSAT*: the same Python interface as for SAT enables to model and solve optimization optimization problems via various Maximum Satisfiability (MaxSAT) solvers (including R2) with helpers for cardinality constraints.
4. *Z3* [4, 5]: a SAT Modulo Theories (SMT) solver with Python bindings that supports rich type systems including booleans, integers, reals, bitvectors, and arrays, along with quantifiers for more expressive constraints.

The MCP Solver has several use cases. One use caseis its integration into an *AI chatbot interface* (like the Claude Desktop or the Cursor application). During a chat session, the user can state a problem (e.g., a scheduling, combinatorics, or verification problem) in plain English, and the language model will connect to the MCP solver via the provided tools and build an encoding, possibly with interactions from the user, solve the encoding with the backend solver, and report back and interpret the solution. The user can then modify the problem statement. Through this approach, the MCP solver offers an enhanced and highly dynamic user interface for the backend solver where encodings can be developed in a dialog with the language model based on immediate feedback from the solver. Once an encoding has been established, the encoding can be extracted and used in other contexts. This setup also provides educational benefits, as a user can observe how informally stated constraints are formalized for the backend solver, make adjustments, and receive explanations from the language model.

Another use case for the MCP Solver is to provide formal solving capabilities to an *autonomous multi-agent system*. To achieve this, one can connect the MCP Solver via the MCP interface to a *Reason and Act (ReAct)* agent [26], which itself is part of a multi-agent system. To exemplify this use case, we added a test client to the software package. The test client implements a basic two-agent system that automatically encodes problem descriptions provided in plain English. It consists of a ReAct agent that communicates with the server and a reviewer agent that checks the result. Our experiments show that this setup is sufficient for the autonomous encoding of problems with small or medium complexity. For more complex problems, we envisage a multi-agent system with a more refined division of work among agents, for instance, with an orchestrator-workers workflow [1].

The MCP Solver with MiniZinc mode was first released on GitHub in December 2024; PySAT and Z3 modes and the build-in client were added in March, the MaxSAT mode was added in June 2025.

2 Related Work

Several prototype systems for connecting language models with formal solvers have been proposed in recent years. PRoC3S [3] employs a two-stage architecture for robotics planning, generating parameterized skill sequences that undergo continuous constraint satisfaction. In a different direction, a counterexample-guided framework [9] merges an LLM synthesizer with an SMT verifier to enhance correctness guarantees for program synthesis. Several systems focus on translating natural language into solver-friendly formats. SATLM [27] converts natural language into logical formulas suitable for SAT solving, while LOGIC-LM [18] implements a comprehensive pipeline from LLM through symbolic solvers to interpreters. For program synthesis specifically, Lemur [25] offers a task-agnostic language model framework.

The integration between language models and verification tools appears in multiple configurations. The LLM-Modulo framework [10] pairs language models with external verifiers, while GenCP [20] incorporates language models into the domain generation of constraint solvers for text tasks. More specialized approaches include StreamLLM [23], which concentrates on language model-based generation of streamlining constraints to accelerate constraint solving and an LLM-based evolutionary search for SAT preprocessing algorithms [21]. Finally, LLMS4CP [13] shows how pre-trained language models can transform textual problem descriptions into executable constraint programming specifications through retrieval-augmented in-context learning.

While these approaches demonstrate the benefits of combining language models with formal solvers, they typically implement fixed pipelines or tight integration for specific use cases. In contrast, our MCP Solver provides a protocol-based architecture that supports iterative interaction within a range of use cases.

3 Tool Calling and the Model Context Protocol

At the heart of state-of-the-art language model integration is the concept of *tool calling*, where a language model, instead of generating only text, can produce structured data (typically JSON) that corresponds to a request to execute a specific function or tool. This allows the language model to interact with external systems, access live data, or delegate complex computations. The *ReAct (Reason and Act)* framework [26] is an agentic architecture that leverages tool calling in an iterative loop. In each iteration, the agent (which wraps a language model) generates a"thought" (its reasoning), an "action" (a tool call), and then receives an "observation" (the result from the tool). This loop continues until the task is complete. This incremental approach is particularly significant for generating a complex encoding because doing that in one attempt is error-prone and unlikely to succeed. Tool calling, however, enables the agent to build the encoding step-by-step, receiving immediate validation feedback after each addition.

The *Model Context Protocol (MCP)* is designed to provide (among other things) a standardized interface for tool calling. MCP operates between a *client*, hosting a language model, and a *server*, hosting various tools and resources. This interface is particularly valuable because it eliminates the need for each tool developer to create custom integrations with every model provider, and vice versa. The MCP creates a single integration point: tools implement the MCP server interface once, and any MCP-compatible client can immediately use them. At its core, MCP defines a stateful server-client communication that enables language models to maintain context across multiple interactions and invoke external tools through a standardized interface. This statefulness means the server remembers previous interactions within a session.

Since its introduction in late November 2024 [2, 16], the MCP has quickly become a foundational open standard for integrating language models with external tools and data sources. Officially released by Anthropic, it offered SDKs, reference implementations, and integration into the Claude Desktop app—alongside ready-to-use servers for systems like Google Drive, GitHub, Slack, Postgres, and Puppeteer. Open-source and enterprise ecosystems embraced MCP early: platforms such as Zed, Replit, Codeium/Cursor, and Sourcegraph implemented MCP support, reflecting widespread developer engagement. By early 2025, MCP had gained institutional recognition: OpenAI integrated support into its Agents SDK and Responses API, and confirmed upcoming MCP functionality in ChatGPT desktop and enterprise endpoints. Microsoft added MCP to Copilot Studio and Azure OpenAI workflows, providing a first-party C# SDK for tool-connected agents. All these data points underscore MCP's widespread production adoption across leading AI platforms.

4 System Architecture

The MCP Solver offers a server component and a client component. The server connects to any MCP compatible client application; our built-in client (described in Section 5) is just one possibility. The server supports four complementary solver backends: MiniZinc for constraint

programming, PySAT for propositional satisfiability, PySAT-MaxSAT for optimization, and Python Z3 for satisfiability modulo theories. Figure 1 gives a high-level architectural diagram of the system.

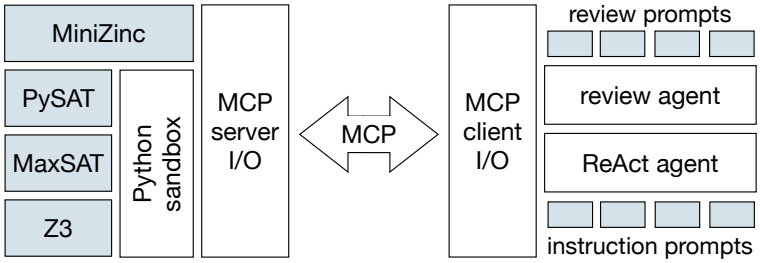

Figure 1 Overview of the MCP Solver architecture showing the main components and their relationships. The system supports four complementary solver backends accessible through a unified MCP interface.

Figure 2 shows the sequence diagram of the MCP Solver when used with an AI chat application as the client, illustrating the dynamic interaction flow between components.

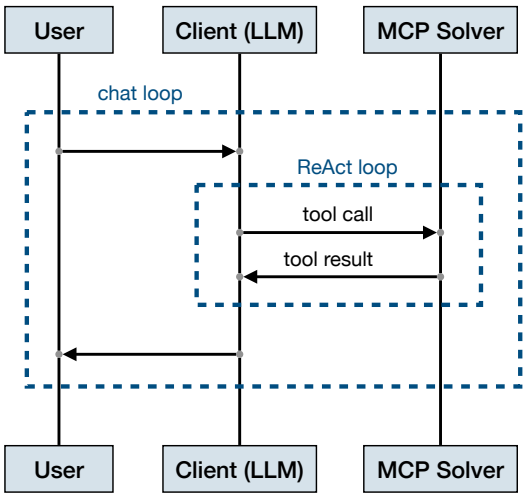

Figure 2 Sequence diagram of MCP Solver's interaction within an AI chat application.

For a unified terminology across all solver backends, we refer to PySAT code, Z3 Python code, or a MiniZinc model as "model", and to a small code entity, like a variable declaration, a MiniZinc constraint, or a Python function definition, as " item ."

4.1 Tools

The MCP Solver provides the following tools, whose item-based operations are demonstrated in Figure 3:

- `clear_model`: reset the solver model;
- `add_item`: add a new item at a specific index;
- `replace_item`: replace an item at a specific index;
- `delete_item`: delete an item at a specific index;
- `get_model`: view the current model with numbered items;
- `solve_model`: solve the model with a specified timeout and receive the solution.

The first version of the MCP solver offered several more tools, but it turned out that fewer tools perform better, as they put a smaller cognitive load on the language model.

In principle, one could run all four solving backends in parallel, with the client selecting on demand the appropriate backend for each problem. However, this burdens the language model with considerable complexity, as it needs to be instructed for all four solving backends. This increases the context size and token use and makes the entire operation potentially confusing for the language model. The current setup assumes that for each session, only one of the four solver backends is used. A command line flag chooses whether the MCP Solver is run in MiniZinc, PySAT, MaxSAT, or Z3 mode.

4.2 Prompt Structure and Design

For each backend (MiniZinc, PySAT, MaxSAT, Z3), a detailed *instruction prompt* is provided to the main ReAct agent via MCP. This prompt acts as its system guide, explaining the available tools and the incremental model-building paradigm. The prompt provides syntax examples and best practices tailored to the specific backend (e.g., MiniZinc syntax vs. Python with PySAT). This focused instruction ensures the agent generates valid code for the chosen solver.

The instruction prompts vary considerably in size across the four modes: MiniZinc requires less than 1000 words, while the other modes use between 3000 and 4000 words. The MiniZinc prompt's brevity stems from the language's declarative nature and standardized features. For instance, solution export is handled automatically by the MiniZinc runtime, whereas the Python-based backends require explicit instructions for formatting and exporting solutions.

The prompts in their current form have emerged via a meta-prompting process. We started with a basic, manually written prompt. Then we carried out an iterative refinement process: observing the performance of the agent, which mistakes or errors are made, followed by adjusting and extending the prompt accordingly.

4.3 Workflow

Next, we describe the full workflow from a problem statement in English to a solution computed by a solver.

1. *Problem Statement.* The process starts with a problem statement in natural language, which can be as brief as "Find a solution to the 10-Queens Problem," and can be entered as a prompt within an AI chatbot or provided as a text file for a command-line client application.
2. *Encoding.* The ReAct agent, guided by its backend-specific instructions prompt, receives the query. It begins to build an encoding by calling `add_item` and other tools.
3. *Incremental Validation.* After each `add_item` call, the MCP Solver's backend validates the new code fragment. It uses MiniZinc methods to evaluate the code (or Python's AST module to parse the code for Python-based backends). If the agent makes a mistake, the server returns a precise error, which the agent observes and corrects in the next step, often by using `replace_item`; if the item is correct, the server returns the encoding generated so far, with items labeled with indices to ensure consistent indexing between client and server. See Figure 3 for an example of this item-based editing approach.
4. *Solving.* Once the problem is fully encoded, the agent calls `solve_model` with a certain timeout. The server executes the backend solver in a secure, isolated process.

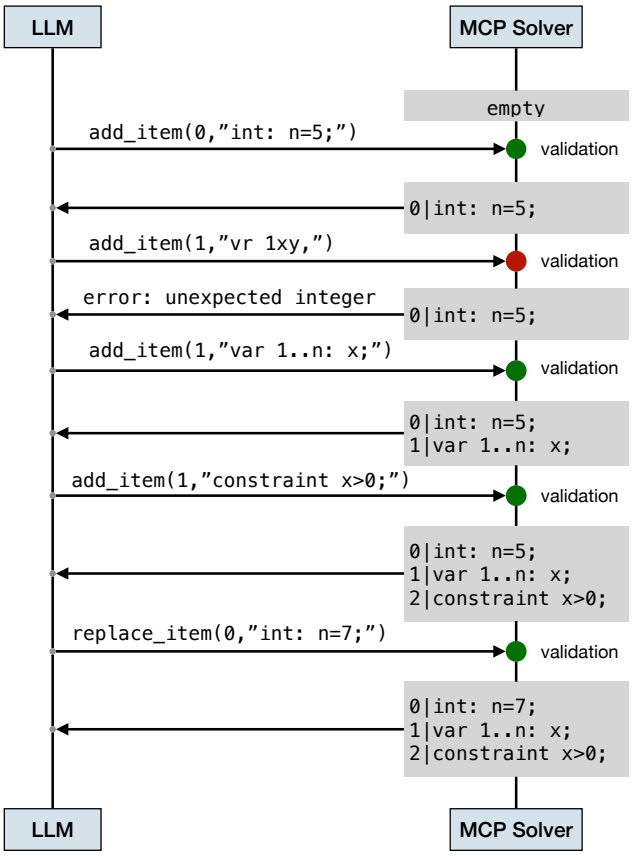

Figure 3 Example for MCP Solver's item-based model editing with validation. Each modification is validated before being applied, maintaining model consistency. Numbers indicate item indices.

5. *Review*. Once the solver has terminated or the timeout has been reached, the instructions tell the agent to critically review the result: in case of "satisfable," whether the solution provided by the solver satisfies all constraints of the problem statement, or, in the case of "unsatisfiable," whether all constraints in the encoding are justified by constraints in the problem statement. In a multi-agent setup, modeling and reviewing are split between two different agents (see Section 5).
6. *Output*. The result is then delivered to the user. In an interactive setup within an AI chatbot, the user can now adjust the problem, add or remove constraints, so that the workflow is repeated from Step 1. This way, the user and the language model can engage in a dynamic dialogue with real-time feedback from the solver.

4.4 Incremental Validation

Each backend performs multi-stage validation before solving, catching errors early, and providing precise diagnostics. In MiniZinc mode, the system reconstructs the complete model after each edit—automatically injecting necessary global definitions—and submits it to the MiniZinc analyzer. This single operation combines parsing, type checking, and model-level consistency checking. The system intercepts syntax, typing, or semantic errors and reports them with exact line and column details.

The Python-based SAT and MaxSAT modes execute user scripts in sandboxed subprocesses, isolating validation from the main application. The isolated environment parses code to detect syntax errors, then applies a restrictive import policy to block unauthorized modules. An Abstract Syntax Tree (AST) analysis pass identifies common logical errors, such as reassigning collections to scalars. Before executing the solving logic, the system verifies that scripts contain the required solver invocation and solution export calls, providing immediate warnings for missing calls.

Z3 mode validation balances safety with responsiveness. A pre-execution pass blocks disallowed operations—including arbitrary file I/O and dynamic code execution—while catching syntax errors through parsing. Unlike SAT backends, Z3 does not enforce solver-invocation patterns at parse time. A runtime wrapper monitors for missing model export or satisfiability check calls, implementing default extraction mechanisms when necessary.

4.5 Execution and Solution Processing

After validation, each backend executes the solving phase with isolation and timeout controls while maintaining a uniform synchronous interface. MiniZinc mode passes solve requests to the MiniZinc library within an asynchronous worker thread bounded by a hard timeout. The library manages solver processes with inherited isolation and cleanly cancels jobs on timeout to prevent resource leakage.

Python-based SAT and MaxSAT backends execute user scripts in dedicated subprocesses. The system captures standard output and errors for reporting, with CPU-time alarms enforcing configured time limits. After completion, subprocesses package satisfiability status, variable assignments, and objective values, returning them to the main process through secure channels.

Z3 mode runs in the main process within a restricted namespace, exposing only approved operations and the Z3 API. Run-time alarms prevent overruns, and the runtime wrapper that handles missing export calls captures valid models even when users omit explicit solution-export steps. All backends normalize errors from validation, sandbox enforcement, or solving into structured codes and messages, providing consistent feedback to MCP clients.

Currently, the solving is performed synchronously; when the `solve_model` tool is called, the entire system waits for the solver to return a solution or for the given timeout to be reached. The timeout is provided as a parameter for the tool call. For our tests, a maximum timeout of 30 seconds was sufficient and worked well in conjunction with the internal timeouts of the AI chatbot application. For problems with longer solving times, we consider adding an asynchronous solving tool that starts the solving in the background and another tool that queries the solving status.

4.6 Modularity

The MCP Solver is architected for modularity, enabling extension with new solver backends. The core design is built on abstract solver interfaces and a shared validation/security scaffolding. The system defines a solver manager interface, which mandates a set of methods that any backend solver must implement (the tools). A base class provides a default implementation that backends can inherit. To add a new solver, one needs to create a new module and implement a class that adheres to this interface. For Python-based solvers, the system provides the security and validation infrastructure described above, including the execution environment and AST-based analysis. This modularity extends to the client, which we will discuss next. Adapting the client to a new solver backend requires only providing a new review prompt.

5 MCP Client

Our package includes an MCP client that provides a command-line interface to the MCP Solver. The client implements a ReAct agent, which utilizes a language model; our model-agnostic interface allows the use of frontier models from all major providers. Although the client does not include a dynamic dialog, as is the case with an AI chatbot, the looping between the client and the server is unlimited. The instruction prompt for the ReAct client includes the request to verify the solution. This is an effective way of self-control, and we have observed that often, the agent identifies a wrong solution and modifies the encoding.

To enhance reliability, the client includes a dedicated *review agent* that independently reviews each solution, following the "LLM-as-a-judge" technique [7]. The reviewer's task is to independently scrutinize the solution: If the result is "satisfiable," does the provided solution correctly satisfy the model? If the result is "unsatisfiable," does the generated model accurately encode all constraints from the original problem statement, and no further spurious constraints?

We do not provide the review agent with the entire message history on purpose, only the problem description, the model, and the solution. This way, the review agent can focus only on this task and is not distracted. By default, the review and the ReAct agent use the same language model, but, in principle, they could use different ones.

Figure 4 shows a sequence diagram of the entire workflow. If the reviewer found an error, one could loop back to the ReAct agent to try again, equipped with feedback.

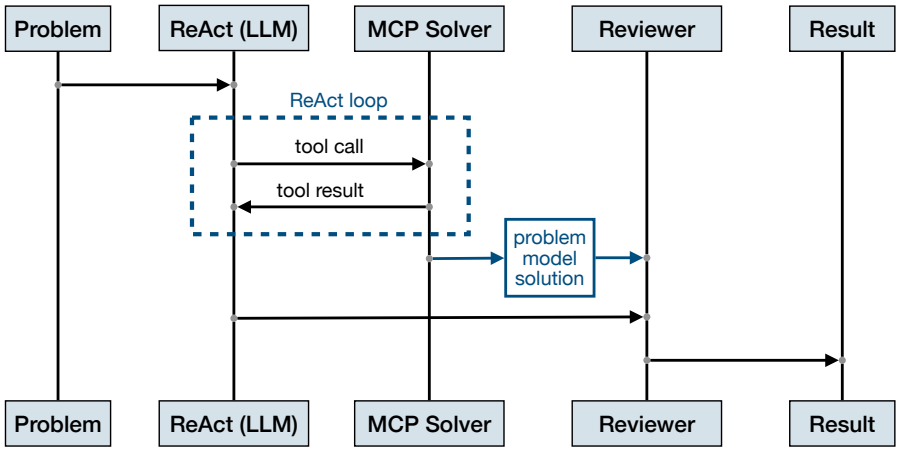

Figure 4 Sequence diagram of client-server interaction of the MCP solver.

The client has proven useful for developing and debugging the solver integration as it has all components (server, client, problems, prompts) at the same location, and hence can adjust the seamless communication between these components. However, the client lacks the interactive aspect provided by an AI chatbot and works as a one-shot encoder.

6 Experiments

We tested the MCP Solver on benchmark problems stated in natural language. All results refer to version 3.3.0 of the MCP Solver. It uses the Python packages mcp (1.5.0), minizinc (0.10.0), z3-solver (4.14.1.0), and python-sat (1.8.dev16). For the client, we use langgraph (0.3.21). As the language model, we use Claude Sonnet 4 version 20250514. Our client can

easily be adapted to other language models because of our model-agnostic architecture. In non-rigorous tests with other language models (GPT-4.1, O3, O4-mini, and Gemini-2.5-pro-preview) only GPT-4.1 performed comparably with Claude Sonnet 4.

We considered twenty problem descriptions, five for each of the four solver backends. The problems can be found as markdown files on the MCP Solver's GitHub repository. These problems were chosen to represent a wide range of problem types from various domains, including scheduling, optimization, verification, combinatorics, and logical puzzles. The selection demonstrates the diversity of possible applications and the individual strengths of the solver backends. For each solver backend, one of the five selected problems is unsatisfiable.

We ran all twenty problem descriptions five times. Table 1 shows the results, including statistics on the number of tool calls and token usage.

Table 1 The table shows average token usage in thousands (k) for *ReAct* (ReAct Agent Total) and *Rev* (Reviewer Total), followed by the average number of tool calls: C (`clear_model`), A (`add_item`), R (`replace_item`), D (`delete_item`), G (`get_model`), and S (`solve_model`). (5 runs per problem).

Mode	*Problem*	C	A	R	D	G	S	*ReAct*	*Rev*
MiniZinc	tsp	2.4	16.2	0.0	0.2	3.2	2.4	143.5k	2.8k
	carpet_cutting	1.0	9.8	0.0	0.0	1.0	1.0	58.6k	3.6k
	zebra	1.0	12.6	0.0	0.0	1.0	1.0	86.7k	3.0k
	university_scheduling	1.0	14.2	0.0	0.0	1.0	1.0	106.5k	3.2k
	university_scheduling_unsat	2.2	24.2	1.4	0.0	4.6	4.2	426.7k	2.9k
PySAT	furniture_arrangement	1.0	7.8	0.0	0.0	1.0	1.0	127.4k	4.1k
	sudoku_16x16	1.0	8.6	0.0	0.0	1.2	1.2	186.3k	8.8k
	equitable_coloring_hajos	1.0	6.4	0.2	0.0	1.2	1.2	120.0k	4.6k
	no_three_in_line_5x5	2.6	11.4	5.6	0.2	7.4	7.4	825.5k	5.2k
	petersen_12_coloring_unsat	1.6	8.2	0.4	0.2	2.4	2.2	269.4k	4.0k
MaxSAT	task_assignment	1.0	5.0	0.0	0.0	1.0	1.0	85.8k	3.4k
	equipment_purchase	1.0	5.6	0.0	0.0	1.2	1.0	98.5k	3.5k
	package_selection	1.0	5.6	0.6	0.0	1.4	1.2	114.5k	3.5k
	network_monitoring	1.2	6.0	0.6	0.0	2.0	1.8	152.6k	4.1k
	workshop_scheduling_unsat	1.8	9.0	3.0	0.0	5.2	4.8	396.9k	3.0k
Z3	cryptarithmetic	1.0	6.8	0.0	0.0	1.0	1.0	100.9k	2.6k
	sos_induction	1.0	8.0	0.2	0.0	1.4	1.4	155.6k	4.7k
	array_property_verifier	1.0	7.8	0.0	0.0	1.0	1.0	110.2k	2.0k
	processor_verification	1.8	11.8	0.4	0.0	2.0	2.0	284.4k	3.8k
	bounded_sum_unsat	1.2	10.6	0.0	0.0	2.2	2.2	173.5k	3.2k

All hundred runs produced solutions and were confirmed by the reviewer. By manual inspection, we could verify that indeed all results were correct.

Tool usage patterns show the difficulty a problem poses to the agent. Most problems required single solve attempts, with `solve_model` calls averaging 1.0–1.2 per problem. The `add_item` tool usage ranges from 5.0 calls for simple MaxSAT problems to 24.0 calls for the MiniZinc zebra puzzle. Increased use of `delete_item`, `replace_item`, and `clear_model` indicates that the agent requires several attempts (one call to `clear_model` is obligatory at start). The difficulty of a problem can also be seen from the token usage, which varies by more than a factor of 10 between the easiest and the hardest problem.

7 Conclusion

The MCP Solver provides language models access to formal solving and reasoning capabilities via the standardized MCP interface. By supporting multiple solving paradigms, the MCP Solver addresses a broad range of problems while maintaining a consistent interface. The flexible architecture enables various use cases, from dynamic problem refinement through natural language interaction when integrated into an AI chatbot to the integration into a multi-agent system for autonomous modeling and solving. The MCP Solver, although stable and application-ready, is still under development. Presently planned additions are Minimal Unsatisfiable Subset support for PySAT, Answer Set Programming, and Model Counting modules, as well as an asynchronous solving interface for longer timeouts. The support of encodings that process instance data (such as graph or tabular data or MiniZinc data files) would also be an interesting addition that enhances the system's versatility. The MCP solver can be integrated into a multi-agent system that uses an orchestrator-workers workflow to autonomously develop more complex encodings, where the encoding task is split into independent components. Such a system could include several solver backends with a routing agent deciding which one to use. Such an approach can optimize solving time by autonomously generating and testing alternative encodings for components.

References

1 Anthropic. Building effective agents, 2024. URL: `https://www.anthropic.com/engineering/building-effective-agents`.

2 Anthropic. Model context protocol: A standard for AI system integration, October 2024. URL: `https://modelcontextprotocol.io`.

3 Aidan Curtis, Nishanth Kumar, Jing Cao, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Trust the PRoC3S: Solving long-horizon robotics problems with llms and constraint satisfaction. In Pulkit Agrawal, Oliver Kroemer, and Wolfram Burgard, editors, *Conference on Robot Learning, 6-9 November 2024, Munich, Germany*, volume 270 of *Proceedings of Machine Learning Research*, pages 1362–1383. PMLR, 2024. URL: `https://proceedings.mlr.press/v270/curtis25a.html`.

4 Leonardo de Moura and Nikolaj Bjørner. Z3 API in Python. `https://ericpony.github.io/z3py-tutorial/guide-examples.htm`. Accessed: 2025-03-20.

5 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. `doi:10.1007/978-3-540-78800-3_24`.

6 Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: program-aided language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR, 2023. URL: `https://proceedings.mlr.press/v202/gao23f.html`.

7 Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Yuanzhuo Wang, and Jian Guo. A survey on LLM-as-a-Judge. *CoRR*, abs/2411.15594, 2024. `doi:10.48550/arXiv.2411.15594`.

8 Alexey Ignatiev, Zi Li Tan, and Christos Karamanos. Towards universally accessible SAT technology. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, volume 305 of *LIPIcs*, pages 16:1–16:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. `doi:10.4230/LIPICS.SAT.2024.16`.

9 Sumit Kumar Jha, Susmit Jha, Patrick Lincoln, Nathaniel D. Bastian, Alvaro Velasquez, Rickard Ewetz, and Sandeep Neema. Counterexample guided inductive synthesis using large language models and satisfiability solving. In *IEEE Military Communications Conference, MILCOM 2023, Boston, MA, USA, October 30 - Nov. 3, 2023*, pages 944–949. IEEE, 2023. doi:10.1109/MILCOM58377.2023.10356332.

10 Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Saldyt, and Anil Murthy. Position: Llms can't plan, but can help planning in LLM-modulo frameworks. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=Th8JPEmH4z.

11 Bill Yuchen Lin, Ronan Le Bras, Kyle Richardson, Ashish Sabharwal, Radha Poovendran, Peter Clark, and Yejin Choi. Zebralogic: On the scaling limits of llms for logical reasoning. *CoRR*, abs/2502.01100, 2025. doi:10.48550/arXiv.2502.01100.

12 Zhan Ling, Yunhao Fang, Xuanlin Li, Zhiao Huang, Mingu Lee, Roland Memisevic, and Hao Su. Deductive verification of chain-of-thought reasoning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL: http://papers.nips.cc/paper_files/paper/2023/hash/72393bd47a35f5b3bee4c609e7bba733-Abstract-Conference.html.

13 Kostis Michailidis, Dimos Tsouros, and Tias Guns. Constraint modelling with llms using in-context learning. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain*, volume 307 of *LIPIcs*, pages 20:1–20:27. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.20.

14 MiniZinc Development Team. *MiniZinc Python: Native Python Interface for the MiniZinc Toolchain*, 2025. Accessed: 2025-03-20. URL: https://python.minizinc.dev/.

15 Seyed-Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. GSM-Symbolic: understanding the limitations of mathematical reasoning in large language models. *CoRR*, abs/2410.05229, 2024. doi:10.48550/arXiv.2410.05229.

16 Model Context Protocol Development Team. Model context protocol: Seamless integration between llm applications and external data sources, 2025. Accessed: 2025-03-20. URL: https://github.com/modelcontextprotocol.

17 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

18 Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-LM: empowering large language models with symbolic solvers for faithful logical reasoning. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 3806–3824. Association for Computational Linguistics, 2023. doi:10.18653/V1/2023.FINDINGS-EMNLP.248.

19 Chengwen Qi, Ren Ma, Bowen Li, He Du, Binyuan Hui, Jinwang Wu, Yuanjun Laili, and Conghui He. Large language models meet symbolic provers for logical reasoning evaluation. *CoRR*, abs/2502.06563, 2025. doi:10.48550/arXiv.2502.06563.

20 Florian Régin, Elisabetta De Maria, and Alexandre Bonlarron. Combining constraint programming reasoning with large language model predictions. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain*, volume 307 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.25.

21 André Schidler and Stefan Szeider. Extracting problem structure with LLMs for optimized SAT local search. In *The 18th International Symposium on Combinatorial Search (SoCS 2025)*, 2025. To Appear.

22 Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity, 2025. arXiv:2506.06941.

23 Florentina Voboril, Vaidyanathan Peruvemba Ramaswamy, and Stefan Szeider. StreamLLM: Enhancing constraint programming with large language model-generated streamliners. In *2025 IEEE/ACM 1st International Workshop on Neuro-Symbolic Software Engineering (NSE)*, pages 17–22, Los Alamitos, CA, USA, 5 2025. IEEE Computer Soc. URL: https://doi.ieeecomputersociety.org/10.1109/NSE66660.2025.00010, doi:10.1109/NSE66660.2025.00010.

24 Yuxuan Wan, Wenxuan Wang, Yiliu Yang, Youliang Yuan, Jen-tse Huang, Pinjia He, Wenxiang Jiao, and Michael R. Lyu. Logicasker: Evaluating and improving the logical reasoning ability of large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 2124–2155. Association for Computational Linguistics, 2024. doi:10.18653/V1/2024.EMNLP-MAIN.128.

25 Haoze Wu, Clark W. Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=Q3YaCghZNt.

26 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 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, 2023. URL: https://openreview.net/forum?id=WE_vluYUL-X.

27 Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Satlm: Satisfiability-aided language models using declarative prompting. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL: http://papers.nips.cc/paper_files/paper/2023/hash/8e9c7d4a48bdac81a58f983a64aaf42b-Abstract-Conference.html.