Logic.py: Bridging the Gap between LLMs and Constraint Solvers

Pascal Kesseli
Meta
pkesseli@meta.com

Peter O'Hearn
Meta
peteroh@meta.com

Ricardo Silveira Cabral
Meta
rdsc@meta.com

Abstract

We present a novel approach to formalise and solve search-based problems using large language models, which significantly improves upon previous state-of-the-art results. We demonstrate the efficacy of this approach on benchmarks like the logic puzzles tasks in ZebraLogicBench. Instead of letting the LLM attempt to directly solve the puzzles, our method prompts the model to formalise the problem in a logic-focused, human-readable, domain-specific language (DSL) called *Logic.py*. This formalised representation is then solved using a constraint solver, leveraging the strengths of both the language model and the solver. Our approach achieves a remarkable 65% absolute improvement over the baseline performance of Llama 3.1 70B on ZebraLogicBench, increasing its accuracy to over 90%. This significant advancement demonstrates the potential of combining language models with domain-specific languages and auxiliary tools on traditionally challenging tasks for LLMs.

1 Introduction

Large language models have revolutionised the field of natural language processing, achieving state-of-the-art results in various tasks such as language translation, text summarisation, and question answering. However, despite their impressive performance, LLMs have historically struggled with certain tasks that require a deeper understanding of mathematical and logical concepts. For instance, Kambhampati *et al.* [2024] demonstrated that LLMs are unable to plan and reason about complex problems, highlighting the need for further research in this area. In this paper, we focus on improving the performance of LLMs in solving Logic Grid Puzzles, which we explain in more detail in Sec. 1.2, as well as first order logic tasks. We present the following research contributions:

1. ***Logic.py***: We introduce a domain-specific, human-readable language called *Logic.py*, which facilitates expressing logic and search-based problems by LLMs, and simplifies human annotation processes for fine-tuning.
2. **Polymath Logic Agent**: We implement an agentic solver engine called *polymath* which accepts search-based, informal problem statements, formalises them in *Logic.py* and solves them using a constraint sovler.
3. **ZebraLogicBench and FOLIO Evaluation**: We evaluate the efficacy of this approach on the logic puzzle benchmark *ZebraLogicBench* Lin *et al.* [2024] and the first order logic benchmark FOLIO Han *et al.* [2022].

The general case for a DSL is not unlike that for intermediate languages in compilation. Given a new programming language it is "obvious" from the Church-Turing thesis that a mapping to assembly language exists, but is not necessarily obvious if a mapping with good efficiency properties (say) exists, and the compilation research community has found it helpful to use intermediate languages

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

https://doi.org/10.52202/085713-2093

to structure the design of compilers. Here, it might seem likely that constraint solvers (automatic theorem provers) could be helpful to approach logic problems stated in natural language, but just how helpful or the best way to do so is not a priori obvious; like in compilation, our thesis is that DSLs can help. A similar analogy is the relation between SQL and first-order logic; SQL provides facilities that make for briefer or more direct human expression than their expansions into FOL. A similar example for *Logic.py* is presented in Section 3.1.

Our results support the case that this DSL can be helpful in structuring the mapping from natural language to that of a solver. In particular, Berman *et al.* [2024] evaluate their approach against a benchmark set of 114 Zebra puzzles. Their multi-agent system has a more sophisticated translation process which includes a refinement loop, but this approach raises the puzzle accuracy of GPT-4 from 23.7% to 55.3%, compared to our improvement from 24.9% to 91.4%.

1.1 Related Work

1.1.1 Formal Reasoning and Theorem Provers

Prior research has explored techniques to enhance the ability of LLMs in these central reasoning tasks, such as chain-of-thought prompting and introducing symbolic representations. However, according to Berman *et al.* [2024], these frameworks often struggle with complex logical problems like Zebra puzzles, partly due to the inherent difficulty of translating natural language clues into logical statements. They propose integrating LLMs with theorem provers to tackle such challenges, demonstrating significant improvements in puzzle-solving capabilities. Our approach significantly outperforms the gains of 10-15% reported in their work.

Logic-LM is a framework that combines LLMs with symbolic solvers to enhance logical reasoning capabilities, demonstrating substantial performance improvements over traditional LLM-based approaches Pan *et al.* [2023]. Ye *et al.* [2023] introduce Satisfiability-aided Language Modeling (SatLM), a method that combines large language models with automated theorem provers to enhance reasoning capabilities, demonstrating state-of-the-art performance on multiple datasets. Our approach again compares favorably to their results, and we also focus on deriving human-readable constraints that can be easily debugged and applied in human annotations for training.

1.1.2 Neuro-Symbolic Approaches

Al-Negheimish *et al.* [2023] highlight that the challenge of numerical reasoning in machine reading comprehension has been addressed by various prompting strategies for LLMs. However, these approaches often struggle to provide robust and interpretable reasoning. They contrast these techniques against their neuro-symbolic approach, which has shown promising results by decomposing complex questions into simpler ones and using symbolic learning methods to learn rules for recomposing partial answers. Our work focuses on logical rather than numerical reasoning, but the two techniques are ultimately orthogonal and could perhaps even be combined.

1.1.3 Augmenting LLMs with External Knowledge

The LLM-Augmenter system, proposed by Peng *et al.* [2023], addresses the limitations of LLMs in real-world applications by augmenting them with plug-and-play modules that ground responses in external knowledge and iteratively revise prompts to improve factuality. Similarly, the Logic-Enhanced Language Model Agents (LELMA) framework, proposed by Mensfelt *et al.* [2024], integrates LLMs with symbolic AI to enhance the trustworthiness of social simulations, addressing issues such as hallucinations and logical inconsistencies through logical verification and self-refinement. Our approach is agentic as well, but focuses on making formal reasoning tools as accessible as possible to the model.

1.1.4 Improving Mathematical Reasoning

Imani *et al.* [2023] propose a technique to improve the performance of LLMs on arithmetic problems by generating multiple algebraic expressions or Python functions to solve the same math problem in different ways, thereby increasing confidence in the output results. Similarly, Fedoseev *et al.* [2024] propose a method to enhance LLMs mathematical problem-solving capabilities by fine-tuning them on a dataset of synthetic problems and solutions generated using Satisfiability Modulo Theories

(SMT) solvers, specifically the Z3 API. Both approaches are focussed on mathematical reasoning, whereas our engine focuses on logical reasoning in propositional formulas.

1.2 Logic Grid Puzzles

We evaluate the effectiveness of our approach on the *ZebraLogicBench* benchmark presented in Lin *et al.* [2024]. *ZebraLogicBench* is a dataset of 1000 Logic Grid Puzzles, also referred to as Zebra Puzzles. These puzzles consist of a series of clues about features of entities in a described environment. In order to solve the puzzle, one has to guess the correct features of all entities, while respecting all the information provided in the cluses. We provide a full example of such a Zebra puzzle, including the expected solution, in the technical appendix in Fig. 6 and Tab. 5, respectively.

2 Preliminaries

In this section, we provide an overview of the formal reasoning engines that serve as the foundation for our approach. Specifically, we focus on their key properties and capabilities that are crucial to the success of our implementation.

2.1 Constraint Solvers

Constraint solvers are computational tools that enable the efficient solution of complex constraint satisfaction problems. In the context of formal reasoning, constraint solvers play a crucial role in automating logical deductions and verifying the correctness of systems. At their core, constraint solvers explore possible assignments of unknown or nondeterministic variables in order to satisfy a set of constraints in a formula. Boolean satisfiability (SAT) solvers are among the most well-known types of solvers in computer science, where the solver is allowed to assign free boolean variables a truth value in order to satisfy the clauses in the formula. While SAT solvers can solve NP-complete problems, mapping problems directly to SAT can be challenging. In this paper we instead use higher level solvers and APIs that provide support for more complex domains as well as other built-in features.

2.2 SMT solvers

SAT solvers are incredibly powerful for solving propositional logic formulas, but they provide limited supports when reasoning over complex domains that involve arithmetic, arrays, or other data structures. In essence, all these domains need to be mapped to boolean formulas when using SAT solvers, such as mapping finite integer types in programming languages to a vector of boolean variables, one for each bit of the finite integer type.

Satisfiability Modulo Theories solvers extend the capability of SAT solvers by incorporating additional theories, allowing them to reason about richer domains. An SMT solver allows users to express constraints not just using boolean variables, but also variables of numerical, string, array, or data structure types. Additionally, SMT solvers add support for explicit quantifiers, allowing users to express quantifier alternations (e.g. "for each X there exists a Y such that..."). This is also a key feature why we use SMT solvers as one possible reasoning back-end for *Logic.py*, specifically for the first-order logic problems in FOLIO Han *et al.* [2022].

2.3 CBMC - Bounded Model Checker for C and C++ programs

CBMC Clarke *et al.* [2004] is a static analysis tool designed for C and C++ programs. It operates by mapping programs to formulas in a back-end solver, typically SAT or SMT, which are satisfiable if and only if the mapped program exhibits a specific property. This capability enables CBMC to effectively check programs for bugs or other properties of interest. Notably, CBMC is powered by the underlying CPROVER static analysis engine, which also supports other language front-ends, such as the JBMC Cordeiro *et al.* [2018] Java front-end.

Since CBMC implements a mapping between programs and SAT or SMT formulas, it can serve as a convenient front-end for such constraint sovlers, exposing an API that allows expressing SAT or SMT formulas as C programs with free input variables. Expressing formulas in this fashion makes

 https://doi.org/10.52202/085713-2093

many constraint solver tasks more accessible for human developers, and it is a core hypothesis of this paper that it equally simplifies the use of constraint solvers for LLMs. We describe this DSL that we expose to the LLM in more detail in Sec. 3.

3 The *Logic.py* language

Our goal in designing *Logic.py* was to provide a streamlined API language that allows an LLM to efficiently express search-based problems and leverage a constraint solver. We optimised for the folowing criteria:

1. **Human-Readable**: *Logic.py* should be human-readable, so that it can be interpreted by any programmer with Python experience. This faciliates debugging as well as annotation of training data for fine-tuning to improve the performance of models when using *Logic.py*.
2. **Robustness**: The language should minimise the surface area for syntax errors.
3. **Conciseness**: The model should be able to express the necessary constraints to solve a search-based problem without boilerplate or needing to worry about unrelated implementation details of the programming language.
4. **Expressiveness**: While common constraints, such as uniqueness of a property, should be easy to express in our DSL, the language must not be restricted to just these common cases. Instead, it must allow the LLM to express arbitrary constraints in the underlying constraint solver framework if necessary.

To provide a good basis for our DSL in terms of robustness and conciseness, we decided to not start with CBMC's native C as a base language for *Logic.py* but instead, as its name suggests, we selected Python for this purpose. We use libCST[1] to transform *Logic.py* to C for analysis by CBMC, as explained in more detail in Sec. 5. Python allows the model to introduce new variables without the need for explicit, typed declarations, and similarly, these variables can be reused for other values later on without needing to worry about type compatiblity.

3.1 Type Decorators

In order to further improve the conciseness of *Logic.py*, we borrow well-known language features from existing languages and combine them in our DSL. As an example, expressing uniqueness of a property in SMT directy requries code similar to the example in Fig. 1.

```
(forall ((x T) (y T))
    (=> (distinct x y)
        (not
            (= (id x) (id y)))))
```

Figure 1: Uniqueness constraint in SMT

```
id INT UNIQUE
```

Figure 2: Uniqueness constraint in SQL

The snippet in Fig. 1 states that two distinct objects of type *T* should not have the same *id* property. Compare this to expressing the same constraint in the Data Definition Language (DDL) of the Structured Query Language (SQL) in Fig. 2. Motivated by this example, we introduced a set of custom type decorators to *Logic.py* which allow to express common property constraints in a much shorter, less error-prone way. We provide the full list of *Logic.py* type decorators in Tab. 1.

3.2 Free Variables, Assumptions, and Assertions

A key concept in using constraint solvers are free variables. By default, any variable which is not explicitly initialised in *Logic.py* is assumed to be a free variable. Consequently, the constraint solver is allowed to assign it any value in order to find a satisfying assignment for all the specified constraints. We also say that such a variable has a *nondeterministc* value.

[1] https://github.com/Instagram/LibCST

Decorator	Description
$Unique[T]$	Equivalent to type T, but no two objects of the same class can have the same value for a property marked $Unique$. The behaviour is equivalent to the SQL *UNIQUE* keyword.
$Domain[T, D]$	Restricts the possible values of the property of type T to the values in D, where D is either a sequence of values, or a numerical range. This a more concise way of expressing value domains in the CPROVER framework using assumptions.
$list[T, S]$	Creates a fixed-size list of element type 'T' and size 'S'. This is equivalent to fixed size arrays in ANSI C.

Table 1: *Logic.py* type decorators

Feature	Description
Uninitialised variables	Uninitialised variables receive a nondeterministic value. When solving search-based problems, the solution is automaically marked nondeterministic by our engine and the model can constraint the solution according to its requirements.
$assume(pred)$	Constrains search paths for solutions to only the values where $pred$ is true.
$assert\ pred$	In static analysis verification scenarios, assertions must be true for all possible inputs satisfying assumptions. In our search-based problem harness, they are equivalent to assumptions.
$nondet(list)$	Returns a nondeterministic but valid element from $list$. This can be used in combination with assumptions to find elements in a list that satisfy a predicate, then express additional constraints about them.

Table 2: *Logic.py* nondet features

The CPROVER framework further allows users to specify assumptions and assertions. These two features work in tandem: Assumptions act as preconditions, usually expressed over free variables, to constrain the solver's search to valid or interesting inputs. Assertions, on the other hand, represent safety properties, for which the solver tries to find a falsifying assignment in order to prove the presence of a bug. Both of these features are exposed in *Logic.py* via the $assume(...)$ function and the $assert...$ statement, respectively.

In *Logic.py*and the *polymath* engine, these two are actually treated interchangeably. That means we interpret assertions specified by the model not as safety properties to be checked for violations, but instead as requirements on a valid solution. Interpreting assertions as assumptions and passing only a single reachability assertion to the constraint solver is a common pattern in program synthesis use cases David *et al.* [2018]. We will explain the structure of constraints we produce at CPROVER intermediate representation level in more detail in Sec. 5. All nondeterminism-related *Logic.py* features are summarised in Tab. 2.

4 Search Problem Formalisation

After reviewing the features of *Logic.py* in Sec. 3, we now review how we prompt the model to express search problems in this language. We provide the full prompts in our open source agent project *polymath*[2]. Note that all our prompts are zero-shot in that we do not provide any full Zebra puzzles as examples to the model. Instead, we only explain *Logic.py* and how to use its extended features to express generic search-based problems.

4.1 Solver backends

At the time of writing, *polymath* supports two solver back-ends: CBMC for SAT and Z3 for SMT problems. CBMC also has an SMT solver engine, but has limited support for quantifiers, which is why we implemented a dedicated mapping from *Logic.py* to SMT. The CBMC back-end is intended for the ZebraLogcBench benchmark in our experiments, whereas the Z3 back-end is used for the first order logic tasks in FOLIO. Note that while the *Logic.py*language remains fundamentally the same irrespective of the back-end, the Z3 back-end allows us to reason more easily over unconstrained lists using universal quantifiers. The running examples in the following sections will focus on a ZebraLogicbench example and the CBMC back-end.

[2]https://github.com/facebookresearch/polymath

 https://doi.org/10.52202/085713-2093

4.2 Describing a Result Data Structure

Before attempting to convert the clues in the natural language puzzle description to solver constraints, we first prompt the model to simply define a data structure that can best represent a valid solution. We ask the model to define this data structure in *Logic.py* and be as precise as possible with respect to type annotations. This allows us to constrain the search space for valid solutions purely based on the domain of the problem, irrespective of explicit clues and constraints. Fig. 3 provides an example of the kind of data structure the LLM will generate.

```
class House:
  house_number: Unique[Domain[int, range(1, 7)]]
  name: Unique[Domain[str, "Alice", "Eric", ...]]
  # ...

class PuzzleSolution:
  houses: list[House, 6]
```

Figure 3: Model Output Example: Result Data Structure

4.3 Constraining a Correct Solution

Once the data structure is defined, we prompt the model to generate a validation function that accepts an argument of this type and asserts that it is the correct solution we are searching for. In the case of Zebra puzzles, this leads to the model adding assertions corresponding to the clues in the puzzle. This approach transforms the challenge for the LLM fundamentally: Instead of searching the solution space for configurations that satisfy the clues stated in the puzzle, it just needs to be able to reason about the clues themselves. Fig. 4 shows a few example clues and how they can be formalised in a *Logic.py* data strucutre chosen by the model.

```
def validate(solution: PuzzleSolution) -> None:
  # Clue 1: Bob is the person who uses a Xiaomi Mi 11.
  bob = nondet(solution.houses)
  assume(bob.name == "Bob")
  assert bob.phone == "xiaomi␣mi␣11"
  # ...
  # Clue 3: The Dragonfruit smoothie lover is somewhere to the
  # left of the person in a ranch-style home.
  d = nondet(solution.houses)
  assume(d.smoothie == "dragonfruit")
  r = nondet(solution.houses)
  assume(r.house_style == "ranch")
  assert d.house_number < r.house_number
  # ...
```

Figure 4: Model Output Example: Solution Constraints

5 Logic Agent Architecture

Fig. 5 illustrates the full implementation architecture of our solver engine, starting with the formalisation steps outlined in Sec. 4. Our engine converts these *Logic.py* constraints into an equivalent, lower-level C representation using a libCST transformer. During this process, the type decorators introduced in Sec. 3.1 are mapped to matching initialisation helpers in CBMC's IR. An example of this mapping is illustrated in the technical appendinx in Fig. 7.

The validation function containing the constraints derived from the Zebra Logic clues is equally converted into a C representation, and embedded into a search harness. In this harness, a nondeterministic instance of the result data structure proposed by the model is initialised using the type

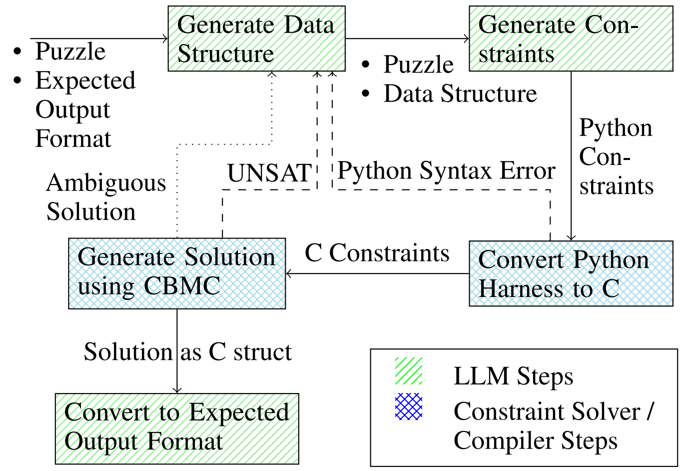

Figure 5: Logic Agent Architecture

decorator information, then constrained using the converted validation function. All assertions in the validation function are converted into assumptions, and we add a single reachabilty assertion to prompt the constraint solver to find an assignment for the nondeterministc puzzle solution such that it satisifes all constraints. We provide an example of a CBMC ANSI-C harness in the technical appendix in Fig. 8.

5.1 Error Recovery

Due to the heuristic nature of the formalisation and constraint solver steps, our solver engine can fail at various steps along the pipeline. The model might produce invalid *Logic.py* code to begin with, leading to syntax errors in libCST. These errors are caught during the C harness generation, in which case we revert to the original data structure generation step and start the process from scratch. We currently do not provide any information about the syntax error to the model or ask it to fix its prior mistakes, and such approaches could be explored in future work.

Our solver back-ends can also detect that a formalisation has no solution, i.e. is unsatisfiable, or whether there are multiple solutions, i.e. the constraints are ambiguous. In general, logic problems provided as user input may indeed be inherently contradictory, or might allow for multiple independent solutions. Thus it cannot automatically be concluded that unsatisfiable or ambigous constraints are a formalisation mistake by the LLM. However, if it is known beforehand that the problem must have a solution, or must have exactly one solution, it can of course only improve outcomes to treat UNSAT or ambiguous constraints as errors and restart.

For example, for ZebraLogicBench in our experimental evaluation, it is known beforehand that each sample must have exactly one solution. For this reason we implemented restarting on UNSAT, but did not yet implement restarting on ambiguity. For this particular benchmark, we did not observe any significant difference in score in our experimental evaluation in Sec. 6 whether we restart on UNSAT or not, since UNSAT results were extremely rare in our experimental runs. Future work should also explore how well models can predict based on the input puzzle whether multiple valid solutions or no valid solutions are plausible.

6 Experimental Evaluation

To evaluate the effectiveness of our approach, we conducted experiments on ZebraLogicBench, a benchmark suite consisting of 1000 logic grid tasks. In order to run the evaluation independently, researchers must request access to a private dataset hosted huggingface.co. We implemented a benchmark runner that accepts this dataset as its input, which we share in our open source project

https://doi.org/10.52202/085713-2093

Model	Puzzle Accuracy					Cell Accuracy
	All	Small	Medium	Large	XL	
grok-3-mini-fast-beta-high	92.6	98.75	96.43	93.5	76.5	94.63
o3-mini-2025-01-31-high	91.7	99.69	97.14	87.5	75.5	95.7
Polymath Meta-Llama-3.1-70B-Instruct	90.7	93.75	93.21	90	83	92.56
o3-mini-2025-01-31-medium	88.9	99.69	97.86	88	60	90.41
o1-2024-12-17	81	97.19	92.14	78	42.5	78.74
grok-3-mini-fast-beta-low	80.7	98.75	96.43	77	33.5	84.22
deepseek-R1	78.7	98.44	95.71	73.5	28.5	80.54
Polymath gpt-4o	77.8	91.88	79.64	75.5	55	76.34
Polymath claude-3-5-sonnet-20241022	76.9	85.31	78.57	76	62	80.58
o3-mini-2025-01-31-low	74.8	99.38	91.07	64.5	23	72.6
o1-preview-2024-09-12	71.4	98.12	88.21	59.5	17	75.14
claude-3-5-sonnet-20241022	36.2	84.69	28.93	4	1	54.27
Llama-3.1-405B-Inst-fp8@together	32.6	81.25	22.5	1.5	0	45.8
gpt-4o-2024-08-06	31.7	80	19.64	2.5	0.5	50.34
gemini-1.5-pro-exp-0827	30.5	75.31	20.71	3	0	50.84
Mistral-Large-2	29	75.94	15	2.5	0	47.64
Qwen2.5-72B-Instruct	26.6	72.5	12.14	0	0	40.92
Meta-Llama-3.1-70B-Instruct	24.9	67.81	10.36	1.5	0	27.98

Table 3: ZebraLogicBench Puzzle Accuracy Results

polymath. The output result of the benchmark runner is evaluated using the ZeroEval[3] evaluation suite provided by the benchmark authors.

Due to resource and time constraints, we only evaluated the effect of *polymath* on the models Llama 3.1 70B, GPT-4o, and Claude 3.5 Sonnet. We run the logic agent implementation on a developer server with 56 cores and 114GB RAM. We self-hosted Llama 3.1 70B on a clust wither 700 GPUs. In this environment, running 100 tasks concurrently, full benchmark run takes approximatley 15 minutes. We evaluted only our own *polymath* agent implementation using this setup. All other results listed were taken directly from the ZebraLogicBench leaderboard Lin *et al.* [2024]. All experiments were performed on these off-the-shelf models without additional fine-tunging.

Tab. 3 illustrates that using *polymath* boosts the performance of all three models. Most notably, Llama 3.1 70B previously attained a new SOTA using our agent, with 91.4% accuracy, achieving a 20% margin over OpenAI o1-preview. However, o3-mini and grok-3-mini-fast-beta-high since exceeded this result by 1% and 1.9% respectively. However, our implementation still solves the most puzzles in the hardest XL puzzle category, where it consistently provides the highest gains across all models. This demonstrates that *polymath* can support models particularly on challenging reasoning problems.

As mentioned in Sec. referror-recovery, our experimental setup includes restarting on UNSAT, but not on ambiguity. Removing this restart on UNSAT logic did not lead to any measurable decrease in score across our experimental evaluation runs. In general, we restart at most five times in case of formalisation syntax errors or UNSAT results before aborting. Independently, we also retry at most five times if we receive an HTTP error from our LLM inference API, e.g. due to throttling. In our final ZebraLogicBench benchmark run of 1000 samples, we observe 51 occurrences of unsatisfiable constraints across all attempts and tasks, and 70 occurrences of syntax errors in *Logic.py*. Of the 9.3% of puzzles where our approach did not produce an entirely correct solution, in 2.9% of cases we were not able to produce a solution at all in five attempts, due to the aforementioned syntax or unsatisfiability errors. In the remaining 6.4% of cases, *polymath* produced a valid, but incorrect constraint and thus produced an incorrect or only partially correct solution.

Overall, our performance gain is particularly striking when compared to how our base model, LLama 3.1 70B Instruct, performs without the help of a constraint solver. On average, it reached 24.9% accuracy across all puzzles, compared to 91.4% accuracy with the constraint solver. We were unable to evaluate the effect of *polymath* on grok-3 and o3-mini due to time and resource constraints.

[3]https://github.com/WildEval/ZeroEval

Model	Accuracy
Polymath claude-3-5-sonnet-20241022	81.28
Logic-LM Pan *et al.* [2023]	78.92
Logic-LM Han *et al.* [2022]	78.1
Polymath gpt-4o	57.64
Polymath Meta-Llama-3.1-70B-Instruct	56.65

Table 4: FOLIO Accuracy Results

We further evaluated our secondary Z3 back-end on the FOLIO benchmark using the same three models. As illustratd in Tab. 4, Claude 3.5 benefited the most from *polymath* support, and again compares favorably to prior approaches.

7 Threats to Validity

While we believe we made a convincing case in Sec. 6 that auxiliary tools can significantly boost the performance of state-of-the-art LLMs on challenging tasks, we would like to address potential threats to the validity of our results. Firstly, we evaluated our engine thus far only against logic grid and first-order logic puzzles. Our technique may prove less impactful when applied to other search-based problems formalised by an LLM. This may be due to the fact that our *Logic.py* prototype is currently incomplete and does not yet support some common Python language and standard library features, or more fundamentally, that the formalisation of other search-based problems may not prove as tractable for state-of-the-art LLMs as logic grid puzzles.

Furthermore, while our approach of using a tailored DSL to allow the LLM to operate an auxiliary tool yielded promising results, further experiments will have to show whether this approach can be generalised to other categories of auxiliary tools and problem domains, such as numerical computational programming languages or algebraic modeling languages.

8 Conclusions and Future Work

In this paper, we presented a novel approach to solving search-based problems using large language models. By introducing the domain-specific language *Logic.py* and implementing an agentic solver engine *polymath*, we demonstrated significant improvements in performance on *ZebraLogicBench* and *FOLIO* benchmarks. Our results show that combining the strengths of LLMs with those of constraint solvers can lead to remarkable advancements in solving traditionally challenging tasks.

The success of this approach highlights the potential for further research in this area. Some potential directions for future work include:

1. **Fine-Tuning** for *Logic.py*: So far, we only used off-the-shelf models to power our *polymath* engine. We plan to fine-tune models to better use *Logic.py* and further increase accuracy.
2. **Extending** *Logic.py*: Developing a more comprehensive and expressive version of *Logic.py* could enable the formalisation of a wider range of search-based problems.
3. **Improving the Logic Agent**: Enhancing the agentic solver engine to better handle complex problem statements and iterate on its mistakes (e.g. syntax errors) could lead to further performance gains.
4. **Applying the approach to other domains**: Exploring the application of our method to other, such as optimisation or mathematical rasoning, could reveal new opportunities for improvement.
5. **Investigating the role of LLMs in problem formalisation**: Further research into the capabilities and limitations of LLMs in formalising search-based problems could provide valuable insights into the design of more effective problem-solving systems.
6. **Teaching neural nets to search like solvers**: Our translations leverage powerful aspects of constraint solvers: proof checking and search. It would be possible to decompose these strengths. Solvers like Z3 have evolved subtle heuristics for approaching computationally intractable problems that are NP-hard and more, and if we can train nets in a way that learns

https://doi.org/10.52202/085713-2093

these heuristics they might provide benefit more broadly than as the targets of translation-based work. In this context, work such as this could provide baselines or targets for what we would want out of the training.

7. **Reinforcement Learning using Logic.py constraints**: During our experimental evaluation, we noticed that interpreting even a single clue out of n incorrectly can lead to a result table where zero columns match the expected value, e.g. because the row order shifted. He have started a reinforcment learning project where we score model answers not using *Logic.py* constraints in to give the model *semantic* feedback (e.g. number of clues respected), rather than *syntactic* feedback (e.g. number of correct cells in result table). This already shows promising improvements in the model's reasoning and coding capabilities.

Building on this project, our broader research aims to further explore the potential of combining large language models with specialized reasoning tools. By pursuing these avenues of research, we believe that it is possible to develop even more powerful and efficient problem-solving systems.

9 Funding Transparency Statement

9.1 Funding

Funding in direct support of this work: GPU compute provided by Meta.

9.2 Competing Interests

Additional revenues related to this work: Employment with Meta.

References

Hadeel Al-Negheimish, Pranava Madhyastha, and Alessandra Russo. Augmenting large language models with symbolic rule learning for robust numerical reasoning. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*, 2023.

Shmuel Berman, Kathleen McKeown, and Baishakhi Ray. Solving zebra puzzles using constraint-guided multi-agent systems, 2024.

Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. JBMC: A bounded model checking tool for verifying Java bytecode. In *Computer Aided Verification (CAV)*, volume 10981 of *LNCS*, pages 183–190. Springer, 2018.

Cristina David, Pascal Kesseli, Daniel Kroening, and Matt Lewis. Program synthesis for program analysis. *ACM Transactions on Programming Languages and Systems*, 40:1–45, 05 2018.

Timofey Fedoseev, Dimitar Iliev Dimitrov, Timon Gehr, and Martin Vechev. LLM training data synthesis for more effective problem solving using satisfiability modulo theories. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, 2024.

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Luke Benson, Lucy Sun, Ekaterina Zubova, Yujie Qiao, Matthew Burtell, David Peng, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Shafiq Joty, Alexander R. Fabbri, Wojciech Kryscinski, Xi Victoria Lin, Caiming Xiong, and Dragomir Radev. Folio: Natural language reasoning with first-order logic. *arXiv preprint arXiv:2209.00840*, 2022.

Shima Imani, Liang Du, and Harsh Shrivastava. Mathprompter: Mathematical reasoning using large language models, 2023.

https://doi.org/10.52202/085713-2093

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Saldyt, and Anil Murthy. Llms can't plan, but can help planning in llm-modulo frameworks, 2024.

Bill Yuchen Lin, Ronan Le Bras, Peter Clark, and Yejin Choi. Zebralogic: Benchmarking the logical reasoning ability of language models, 2024.

Agnieszka Mensfelt, Kostas Stathis, and Vince Trencsenyi. Logic-enhanced language model agents for trustworthy social simulations, 2024.

Liangming Pan, Alon Albalak, Xinyi Wang, and William 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*, pages 3806–3824, Singapore, December 2023. Association for Computational Linguistics.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, and Jianfeng Gao. Check your facts and try again: Improving large language models with external knowledge and automated feedback, 2023.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Satlm: Satisfiability-aided language models using declarative prompting, 2023.

https://doi.org/10.52202/085713-2093

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The claims in the abstract are based on the experimental section. We since added an additional benchmark, FOLIO, but this does not affect the claims in the abstract.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We address limitations in the section "Threats to Validity". We also discuss experiment configurations we were not able to carry out due to budget and time constraints in the section "Experimental Evaluation".

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [NA]

Justification: The paper does not make theoretical claims and does not introduce theorems.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: This project was open sourced and was forked by users already. To guarantee anonimity, we provide a ZIP file of the repository. The paper contains instructions of how to attach polymath to a new inference service, since ours is bespoke to our company resources.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

https://doi.org/10.52202/085713-2093

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The project was open sourced and we provide a ZIP file of the repository for anonimity. The datasets are publicly available on HuggingFace, and the repository contains a script to download them.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [NA]

Justification: We did not fine-tune or train a model in this project. We only used test data sets to evaluate the approach.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: The evaluation suite in our experimental evaluation is provided by a third party, and we only provide the data as yielded by this suite. We repeated this evaluation multiple times before initial and camera-ready submission to confirm our results, but otherwise reported the results as output by this third-party suite.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We documented the hardware we used to run our experiments, apart from GPU resources used in external third party services.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: To the best of our knowledge, none of the concerns in the guidelines are violated in our work.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: The research performed does not directly expose such societal impacts.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.

https://doi.org/10.52202/085713-2093

- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: We did not release a model. The paper only concerns itself with improving the performance of existing models on logic tasks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All licenses are respected and all data is properly cited.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: All newly created artifacts are documented in detail.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: No crowdsourcing or research with human subjects was involved.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

https://doi.org/10.52202/085713-2093

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: The core method development in this research did not involve LLMs.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.

A Zebra Puzzle examples

There are 4 houses, numbered 1 to 4 from left to right, as seen from across the street. Each house is occupied by a different person. Each house has a unique attribute for each of the following characteristics:

- Each person has a unique name: Alice, Eric, Arnold, Peter
- Each person has an occupation: artist, engineer, teacher, doctor
- People have unique favorite book genres: fantasy, science fiction, mystery, romance
- People use unique phone models: google pixel 6, iphone 13, oneplus 9, samsung galaxy s21

Clues:

1. The person who is an engineer is directly left of the person who uses a Samsung Galaxy S21.
2. The person who loves fantasy books is in the second house.
3. Alice is not in the second house.
4. Eric is the person who is a teacher.
5. The person who uses a Samsung Galaxy S21 is the person who loves fantasy books.
6. The person who uses an iPhone 13 is the person who loves science fiction books.
7. The person who loves science fiction books is somewhere left of the person who uses a OnePlus 9.
8. The person who uses a OnePlus 9 is Arnold.
9. The person who is a doctor is the person who loves mystery books.
10. The person who uses an iPhone 13 is the person who is a teacher.

Figure 6: Example Zebra logic puzzle

House	Name	Occupation	BookGenre	PhoneModel
1	Alice	Engineer	Romance	Pixel 6
2	Peter	Artist	Fantasy	Galaxy S21
3	Eric	Teacher	SciFi	iPhone 13
4	Arnold	Doctor	Mystery	OnePlus 9

Table 5: Zebra Puzzle Example Solution

B CBMC C constraint examples

In this section we provide ANSI-C constraint examples for our running example in the paper suitable for consumption by the CBMC constraint solver.

 https://doi.org/10.52202/085713-2093

```c
struct House {
  int house_number;
  const char * name;
  const char * smoothie;
  // ...
};

static int House_house_number[] =
  {1, 2, 3, 4, 5, 6};
static bool House_house_number_used[6];
static const char * House_name[] =
  {"Alice", "Eric", "Peter", ...};
static bool House_name_used[6];
// ...

#define __CPROVER_unique_domain( \
  field, field_domain_array) \
{ \
  size_t index; \
  __CPROVER_assume(index < \
    (sizeof(field_domain_array) / \
     sizeof(field_domain_array[0]))); \
  __CPROVER_assume( \
    !field_domain_array##_used[index]); \
  field_domain_array##_used[index] = \
    true; \
  field = field_domain_array[index]; \
}
// ...

static void init_House(
  struct House * instance) {

  __CPROVER_unique_domain(
    instance->house_number,
    House_house_number
  );
  __CPROVER_unique_domain(
    instance->name,
    House_name
  );
  // ...
}
```

Figure 7: CBMC Data Structure Example

```c
static void validate(
  struct PuzzleSolution solution) {

  bob = __CPROVER_nondet_element(
    solution.houses);
  __CPROVER_assume(bob.name == "Bob");
  __CPROVER_assume(bob.phone ==
    "xiaomi␣mi␣11");
  // ...
}

// ...

int main(void) {
  struct PuzzleSolution solution;
  init_PuzzleSolution(&solution);
  validate(solution);

  __CPROVER_output("solution",
    solution);
  __CPROVER_assert(false, "");
}
```

Figure 8: CBMC Search Constraint Example

https://doi.org/10.52202/085713-2093