AUTO-FORMALIZING NEURO-SYMBOLIC PREDICTORS

Anonymous authors
Paper under double-blind review

ABSTRACT

Neuro-Symbolic (NeSy) predictors incorporate prior knowledge into the prediction process of neural networks, ensuring that outputs satisfy specified constraints, making them particularly suitable for high-stakes applications where compliance with domain knowledge is essential. A key bottleneck in this paradigm is the acquisition of symbolic constraints: encoding domain knowledge into logical formulas remains a manual and expert-intensive process. In this work, we investigate the extent to which auto-formalization via LLMs can systematically translate textual knowledge into symbolic knowledge that can be plugged into NeSy predictors. To this end, we introduce `auto-nesy-bench`, a new benchmark for evaluating constraint formalization and its impact on downstream accuracy of NeSy predictors. Through an extensive evaluation across several domains, we find that LLMs can formalize constraints to a meaningful extent, generating formulas that are often similar to those provided by human experts. Moreover, when the generated formulas are syntactically valid, they can lead to high-quality downstream predictions.

1 INTRODUCTION

Neuro-Symbolic (NeSy) AI combines the learning capabilities of neural networks with the reasoning capabilities of symbolic systems, making it a promising paradigm for building reliable AI models (De Raedt et al., 2021; Garcez et al., 2022). A prominent and well-studied class of NeSy models is that of *NeSy predictors* (Manhaeve et al., 2018; Badreddine et al., 2022; Chen et al., 2026b), which incorporate prior knowledge into the prediction process to ensure that outputs comply with domain-specific constraints (Giunchiglia & Lukasiewicz, 2020; Giunchiglia et al., 2024; Chen et al., 2026a), making them suitable for high-stakes applications (Giunchiglia et al., 2022; 2025; Dash et al., 2022).

Such models typically assume that the *constraints are provided beforehand*, requiring a domain expert to identify the relevant knowledge and formalize it into a representation that can be incorporated into the predictor. This requirement persists even in recent frameworks such as `DeepLog` (Derkinderen et al., 2026) and `ULLER` (Van Krieken et al., 2024), which aim to make NeSy predictors more accessible to practitioners, as the formalization of logical constraints still falls on the end user. This raises a natural question: *what if an end user wants to deploy a NeSy predictor but has no expert available to formalize the required knowledge?* We address this gap by leveraging Large Language Models (LLMs) as domain experts to formalize natural language into symbolic constraints.

Recent work has shown that LLMs can generate constraint programs directly from natural language descriptions (Tsouros et al., 2023; Michailidis et al., 2024). Existing benchmarks target either Boolean satisfiability (SAT), such as `satbench` (Wei et al., 2025), or CP, such as `dcpbench` (Michailidis et al., 2025b), but do not evaluate how generated constraints affect learning and downstream NeSy performance. To address this gap, we introduce `auto-nesy-bench`, a challenging NeSy benchmark that extends the well-established `rsbench` (Bortolotti et al., 2024) with prompts, data instances, output annotations, and evaluation metrics in which the task is to formalize a logical constraint in conjunctive normal form (CNF) out of a textual description (see Fig. 1).

We therefore evaluate both the quality of constraints generated by modern and frontier LLMs on `satbench`, `dcpbench`, and `auto-nesy-bench`, as well as their downstream impact when integrated into state-of-the-art NeSy predictors, such as *semantic probabilistic layers* (Ahmed et al., 2022, `SPL`), or employed as soft constraints using *semantic loss* (Xu et al., 2018, `SL`) on `auto-nesy-bench`. To instantiate this evaluation, the LLM formalizes the constraints in propositional logic (PL) in one of several formalisms ranging from a textual representation of a CNF such as

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

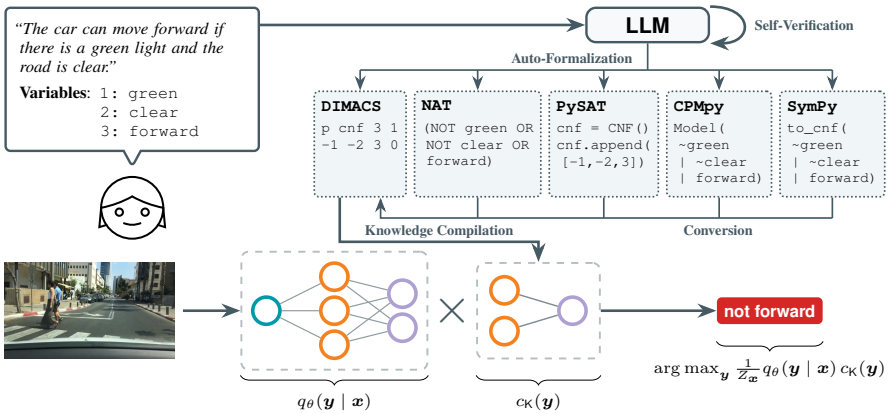


Figure 1: **Illustration of our `auto-nesy-bench` auto-formalization framework**: given a natural language description of a constraint and the corresponding variables, an LLM generates `DIMACS`, `PySAT`, `CPMpy`, `SymPy`, or natural language (`NAT`) formalizations. Non-`DIMACS` outputs are converted to `DIMACS` and later compiled, via knowledge compilation, into circuits that can be used in NeSy predictors (`SPL`), thus ensuring that the output satisfies the specified constraints by design, following Eq. (1). The five boxes show equivalent encodings of $(\texttt{green} \wedge \texttt{clear}) \Rightarrow \texttt{forward}$.

`DIMACS` (Johnson & Trick, 1996) or natural language (`NAT`) to Python programs using `PySAT` (Ignatiev et al., 2018), `CPMpy` (Guns, 2019) and `SymPy` (Meurer et al., 2017), each one coming with different challenges (see Section 3.1). Our results show that high-performing LLMs can formalize NeSy tasks, but performance depends on the level of detail in the constraint description. When the generated formulas are syntactically valid, formalization errors can have little impact on downstream predictions.

Our contributions: We introduce `auto-nesy-bench`, a benchmark for evaluating auto-formalization in NeSy tasks. We investigate the following research questions: (**RQ1**) Can LLMs formalize logical constraints from natural language descriptions? and (**RQ2**) How do NeSy models perform when provided with LLM-generated constraints? To address **RQ1**, we analyze the problem from multiple perspectives, examining which types of (a) LLMs, (b) input formats, and (c) output formats lead to the best results, as well as which configurations fail. For **RQ2**, we evaluate two representative state-of-the-art NeSy approaches, a layer and a loss, using LLM-generated constraints.

2 NeSy Predictors

Among NeSy approaches (De Raedt et al., 2021; Garcez et al., 2022), here we focus on *NeSy predictors*, classifiers designed for reliability (Giunchiglia et al., 2022; Chen & Zhao, 2025; Chen et al., 2026a). Given an input $\boldsymbol{x} \in \mathbb{R}^n$, these models predict a (multi-)label $\boldsymbol{y} \in \{0,1\}^m$ while leveraging *prior knowledge* K, which is usually expressed in the form of structural (Di Liello et al., 2020) or safety requirements (Hoernle et al., 2022). The model then produces a predictive distribution $p_\theta(\boldsymbol{y} \mid \boldsymbol{x}; \mathsf{K})$ that assigns lower (Xu et al., 2018; van Krieken et al., 2023), or even provably zero (Manhaeve et al., 2018; Ahmed et al., 2022), probability to outputs $\boldsymbol{y}$ that violate the constraint K.

As an example, consider Fig. 1, where a NeSy predictor determines whether a car can move forward from input observation $\boldsymbol{x}$, such as an image of the road. Let `green`, `clear`, and `forward` denote Boolean variables indicating whether the traffic light is green, the road is clear, and the car moves forward, respectively. We impose the domain constraint $\mathsf{K} : (\texttt{green} \wedge \texttt{clear}) \Rightarrow \texttt{forward}$, stating that the car can move forward when the traffic light is green and the road is clear. While the neural network may assign nonzero probability to output combinations that violate K, a NeSy predictor incorporates K into the prediction process. It *assigns zero probability to invalid combinations* and renormalizes the remaining probability mass. For example, $\texttt{green} = 1$, $\texttt{clear} = 1$, and $\texttt{forward} = 0$ is ruled out because it violates K, while all combinations satisfying K remain possible.

Existing NeSy predictors differ along two main dimensions. First, they vary in the underlying logical formalism they support, i.e., propositional logic (Ahmed et al., 2022; Giunchiglia et al., 2024), first-order logic (Manhaeve et al., 2018; Zhou, 2019; Badreddine et al., 2022; Pojer et al., 2024; 2025), and

satisfiability modulo theory over linear real arithmetic (Kurscheidt et al., 2025; 2026). Second, they differ in how they enforce constraint satisfaction, relying on paradigms such as probabilistic logic (Lippi & Frasconi, 2009; Marra & Kuželka, 2021; Pryor et al., 2023; Skryagin et al., 2022; Maene & Raedt, 2023) or fuzzy logic (Diligenti et al., 2017; Donadello et al., 2017; Badreddine et al., 2022).

In this work, we focus primarily on *semantic probabilistic layers* (SPL) (Ahmed et al., 2022), a representative state-of-the-art NeSy predictor for PL. SPL is a predictive layer that can be placed on top of a neural network to enforce logical constraints. It combines the unconstrained predictive distribution $q_\theta(\boldsymbol{y} \mid \boldsymbol{x})$ produced by the neural network with a constraint distribution $c_{\mathsf{K}}(\boldsymbol{y})$, represented as a differentiable probabilistic circuit (Darwiche, 2001; Vergari et al., 2019; Choi et al., 2020) obtained through knowledge compilation (Darwiche & Marquis, 2002). The resulting predictive distribution is given by the tractable product of experts:

$$p_\theta(\boldsymbol{y} \mid \boldsymbol{x}; \mathsf{K}) = \tfrac{1}{Z_{\boldsymbol{x}}} q_\theta(\boldsymbol{y} \mid \boldsymbol{x})\, c_{\mathsf{K}}(\boldsymbol{y}), \qquad Z_{\boldsymbol{x}} := \textstyle\sum_{\boldsymbol{y}} q_\theta(\boldsymbol{y} \mid \boldsymbol{x})\, c_{\mathsf{K}}(\boldsymbol{y}), \tag{1}$$

where $Z_{\boldsymbol{x}}$ is the partition function, i.e., the weighted model count induced by K (Chavira & Darwiche, 2008), which can be computed exactly with circuits (Vergari et al., 2021). By doing this, SPL *provably allocates zero probability mass to invalid output combinations*, making it reliable in high-stakes scenarios. However, to work SPL requires that the constraint K is given as an input as a well-formed CNF. As this might not always be possible or easy, we investigate next how to *auto-formalize* K out of a textual description that we assume to be more easily accessible. An alternative to SPL is to use *soft constraints*. A representative NeSy approach is the *semantic loss* (SL) (Xu et al., 2018), which minimizes the negative log-partition function in Eq. (1), see Section I.2.3.

3 Auto-formalizing Neuro-Symbolic Predictors

Our goal is to use an LLM as an expert that translates natural language domain knowledge into symbolic constraints that can be later used by a NeSy predictor. The resulting pipeline, shown in Fig. 1, consists of two stages: first, the LLM formalizes the constraints; then, the generated formula is incorporated into the NeSy predictor and used during learning and inference.

LLMs as constraint experts. Recently, reasoning models have been developed to improve performance on complex tasks by generating intermediate reasoning steps before producing a final answer (Wei et al., 2022; Guo et al., 2025). Following Michailidis et al. (2025b), we treat an LLM as a black-box function $h : \Lambda^* \to \Lambda^*$, where Λ is the model vocabulary and Λ^* denotes the set of finite token sequences. As shown in Fig. 1, given a prompt, the LLM produces an output sequence $\boldsymbol{s}$ without requiring access to its weights, logits, hidden states, or training data.

3.1 Constraint Formalization

Learning symbolic constraints from data has been extensively studied as *constraint acquisition* (De Raedt et al., 2018; Tsouros et al., 2024; 2025). Existing approaches typically learn constraints from labeled satisfying and violating assignments or through interaction with an oracle or solver (De Raedt et al., 2016; Bessiere et al., 2017; Freuder & Wallace, 1992). In our setting, instead of learning from a collection of examples, the LLM receives a single natural language description of the domain knowledge and acts as the constraint-acquisition mechanism (Michailidis et al., 2024).

We focus on PL and use CNF as the target language for constraints. A CNF can be expressed through several equivalent representations, including high-level modeling languages and libraries such as CPMpy, SymPy, and PySAT. For example, as depicted in Fig. 1, the constraint $(\texttt{green} \wedge \texttt{clear}) \Rightarrow \texttt{forward}$ can be expressed in any of these representations, despite representing the same logical relation. We consider these different representations because they reflect interfaces practitioners actually use to specify constraints, from low-level solver formats to high-level modeling libraries, allowing us to test whether LLMs auto-formalization ability depends on the target syntax.

Among these, Python-based libraries (PySAT, CPMpy, SymPy) are of particular interest, as LLMs have repeatedly been shown to be especially capable of generating correct Python code (Chen et al., 2021; Austin et al., 2021; Li et al., 2022), making it a natural interface to test for auto-formalization. We also consider direct generation of Boolean formulas in natural language (NAT) and DIMACS CNF, i.e., the common standard for SAT solvers (Johnson & Trick, 1996). As shown in Fig. 1, all these representations are translated into DIMACS, to unify evaluation (see Section 5.1).

Prompting. Each prompt frames the LLM as an expert in SAT solving and constraint modeling, specifies the syntax of the target representation (`DIMACS`, `NAT`, `PySAT`, `CPMpy`, or `SymPy`), provides the grounded variables, and requires the generated formula to be contained within a `<cnf>...</cnf>` block. Unlike Michailidis et al. (2025b), we also investigate how performance varies with the level of detail provided in the prompt (Section 5.1), thereby assessing the extent to which auto-formalization depends on the user's expertise. Complete prompts are provided in Section F.

Symbol grounding. A key challenge in translating natural language descriptions into logical constraints is identifying relevant symbols and consistently mapping them to logical variables, a problem known as *symbol grounding* (Harnad, 1990). Recent work has shown that widely used open weight LLMs can struggle to correctly identify symbols in data (Wüst et al., 2025; Debole et al., 2026). Furthermore, comparing formulas that use different symbols requires an additional alignment step (Bortolotti et al., 2025), making evaluation substantially more challenging. Since our goal is to study constraint formalization rather than symbol identification, we provide the LLM with the *complete set of primary variables for each task*. Each variable is specified by its name and meaning.[1] As illustrated in Fig. 1, this grounded set of variables serves as the interface between the natural language description and the target representation, allowing us to evaluate whether the LLM correctly captures the intended constraint independently of symbol identification. The LLM may still introduce auxiliary variables, provided that they are explicitly defined in terms of the primary variables.

Self-verification. Following Michailidis et al. (2025b), we allow the LLM to iteratively correct its output over multiple stages. When a syntax error is detected, we provide a correction prompt containing the erroneous formula and corresponding error message. The LLM has to return either `[[OK]]` if no correction is needed, or `[[FIXED]]` followed by the corrected formula in a `<cnf>...</cnf>` block. Before and after each correction step, we automatically fix formatting issues that do not require LLM reasoning, such as incorrect `DIMACS` headers or missing Python imports (see Section C).

Formalization evaluation. Existing benchmarks, such as `satbench` and `dcpbench`, primarily rely on satisfiability results to evaluate the quality of the generated formulas. In `satbench`, this amounts to checking whether the generated formula has the same satisfiability status as the ground-truth formula, i.e., whether both are SAT or UNSAT. In `dcpbench`, the evaluation instead tests whether a generated CP solution is feasible and satisfies the ground-truth constraint. These metrics are not suitable for the full auto-formalization pipeline, as we are interested not only in whether a formula is SAT, but also in *which solutions it admits*. This is because NeSy predictors such as `SPL` assign probability mass only to satisfiable assignments, preventing the final distribution from assigning mass to UNSAT assignments. To capture this, we evaluate the agreement between the valid assignments of the generated formula and those of the ground-truth formula using model counting (MC) (Chavira & Darwiche, 2008). We obtain the entries of the confusion matrix by counting the valid assignments satisfying both the ground-truth ϕ^* and the generated formula $\hat{\phi}$:

$$\text{TP} = \text{MC}(\phi^* \wedge \hat{\phi}),\ \text{FP} = \text{MC}(\neg\phi^* \wedge \hat{\phi}),\ \text{FN} = \text{MC}(\phi^* \wedge \neg\hat{\phi}),\ \text{TN} = \text{MC}(\neg\phi^* \wedge \neg\hat{\phi}), \quad (2)$$

i.e., true positive, false positive, false negative and true negative, respectively. We compute these quantities exactly for tasks with at most 20 variables and use `ApproxMC` (Chakraborty et al., 2016) for larger instances (see Section B for details), then derive F1, precision, and recall. Recall is particularly important in our setting as `SPL` uses the constraints to restrict the space of possible outputs. If a generated constraint incorrectly excludes a valid assignment, the predictor cannot assign probability mass to that assignment, making the corresponding configuration impossible to predict (Grivas et al., 2022; 2024). Finally, we report the proportion of generated formulas containing syntax errors.

3.2 From Auto-Formalization To NeSy Predictors

Once generated, each CNF formula is embedded into an `SPL` (Ahmed et al., 2022) and, as an alternative, used as a `SL` (Xu et al., 2018). CNFs are compiled into sentential decision diagrams (SDDs) (Darwiche, 2011) via the `PySDD` Python library. For `SPL`, the SDD is loaded as a differentiable probabilistic circuit using `cirkit` (The april Lab, 2024), while `SL` uses `KLay` (Maene et al., 2025).

Training and inference. `SPL` is trained by minimizing the negative log-likelihood of the target label $\boldsymbol{y}$ using maximum likelihood: $\mathcal{L} = -\mathbb{E}\left[\log p_\theta(\boldsymbol{y} \mid \boldsymbol{x}; \mathsf{K})\right]$. In contrast, `SL` uses a standard classifier and incorporates K through a penalty loss: $\mathcal{L}_{\text{SL}} = -\log \sum_{\boldsymbol{y} \models \mathsf{K}} \prod_i p(Y_i \mid \boldsymbol{x})$, i.e., the

[1] Different formats require additional information, e.g., `DIMACS` index. Details are given in Section F.

negative log-probability that the network predictions satisfy K. At test time, we perform maximum a posteriori inference over the predictive distribution, i.e., $\hat{\boldsymbol{y}} = \arg\max_{\boldsymbol{y}} p_\theta(\boldsymbol{y} \mid \boldsymbol{x}; \mathsf{K})$ for `SPL` and $\hat{\boldsymbol{y}} = \arg\max_{\boldsymbol{y}} p_\theta(\boldsymbol{y} \mid \boldsymbol{x})$ for `SL`. The quality of the LLM-generated constraints can affect the final predictor in two ways: incorrect constraints may *rule out valid predictions*, while incomplete or weak constraints may *allow invalid predictions*. These effects are hard in `SPL`, where the formula determines the feasible predictions, but act as a training bias in `SL`. Thus, we evaluate not only whether LLMs generate valid constraints, but also whether those constraints yield correct predictions.

Metrics. In addition to F1, accuracy, and recall, we introduce three metrics tailored to our setting: formula relationship (REL), consistency (CON), and model-count ratio (MC-R). REL characterizes the relationship between the generated formula $\hat{\phi}$ and the ground-truth formula ϕ^*, distinguishing four cases: `equal` ($\hat{\phi} = \phi^*$), `permissive` ($\hat{\phi} \Leftarrow \phi^*$), `strict` ($\hat{\phi} \Rightarrow \phi^*$), and `incomparable` ($\hat{\phi} \parallel \phi^*$). A `permissive` formula admits all valid assignments but also some invalid ones, while a `strict` formula admits only valid assignments, but excludes some valid ones. `Incomparable` formulas exhibit both types of errors. CON measures the proportion of assignments consistent with the ground-truth constraint, while MC-R measures the relative number of satisfying assignments admitted by the generated formula. This is useful for `strict` and `permissive` cases, as it quantifies the extent to which the set of satisfying assignments admitted by the generated formula differs from that of the ground-truth formula, e.g., if a `permissive` formula has MC-R $= 1.75$, then $\hat{\phi}$ admits 1.75 times as many satisfying assignments as the ground-truth. Further details are reported in Section B.

4 AUTO-NESY-BENCH

The proposed pipeline raises two questions: ***(RQ1)*** *can an LLM correctly formalize the intended domain knowledge?* and ***(RQ2)*** *how do LLM-generated constraints affect the behavior of the resulting NeSy predictor?* Existing constraint-generation benchmarks address the first question in standalone CP settings, such as `dcpbench`, or SAT settings, such as `satbench`, while NeSy benchmarks typically assume pre-existing background knowledge. Thus, neither evaluates the full pipeline from natural language descriptions to formal constraints and their downstream impact on NeSy predictors. To address this gap, we introduce `auto-nesy-bench`, a benchmark designed to evaluate LLM-based auto-formalization and its downstream impact on NeSy predictors. We first describe the existing benchmarks and then introduce `auto-nesy-bench`. Additional details are in Section D.

SATBench. `satbench` (Wei et al., 2025) evaluates LLMs logical reasoning through natural language description of puzzles generated from CNF formulas. The original benchmark contains 2,100 instances, including both satisfiable and unsatisfiable problems. We repurpose its paired natural language descriptions and reference CNF formulas to evaluate auto-formalization. We exclude UNSAT instances because, when equipped with `SPL`, they assign zero probability to every possible configuration, resulting in an uninformative downstream predictor. After excluding these instances and applying the consistency checks described in Section D.1, we retain 1,041 SAT instances.

DCPBench-Open. We repurpose `dcpbench` (Michailidis et al., 2025a; 2026) for auto-formalization, creating a new version of the original benchmark consisting of 164 instances containing natural language descriptions of combinatorial problems collected from multiple sources. These problems are designed for CP, where the goal is to construct a constraint-based model rather than enumerate the set of solutions. Consequently, the benchmark includes problems involving both discrete and continuous variables, as well as optimization problems. Optimization problems are not suitable for our NeSy predictor setting, as they seek a single optimal solution rather than a set of valid solutions. Similarly, instances involving continuous variables fall outside the scope of PL. We therefore discard such instances during preprocessing. We also remove constraints that lead to excessive circuit sizes during knowledge compilation, as well as instances whose formulas fail to convert into CNF. Additionally, our formulation differs from the original use of the dataset in that, for instance-specific problems, we explicitly ground the relevant instance in the constraint formalization. After this preprocessing, we obtain 85 instances, which we use to evaluate the auto-formalization capabilities of the LLMs.

Auto-NeSy-Bench. `auto-nesy-bench` is an extension of `rsbench` (Bortolotti et al., 2024) which originally contained 8 datasets spanning mathematical reasoning and high-stakes applications, and was designed to study the quality of concepts in NeSy predictors (Marconato et al., 2023). We expand `rsbench` to 18 representative NeSy problems adapted from established NeSy datasets,

Table 1: **Model choice substantially affects auto-formalization performance across benchmarks.** We compare various models on all benchmarks, reporting F1, precision, recall, and syntax error rate (mean ± standard deviation). LLM coverage varies slightly due to resource constraints (see Section I for the full results). Best results are in bold, and second-best results are underlined. Ordered by size.

DATASET	MODEL	F1 (↑)	PRECISION (↑)	RECALL (↑)	SYNTAX ER (↓)
auto-nesy-bench	qwen3-8b	0.372 ± 0.202	0.402 ± 0.192	0.423 ± 0.200	$30.6\% \pm 25.5\%$
	mistral-nemo	0.053 ± 0.074	0.051 ± 0.073	0.154 ± 0.093	$62.8\% \pm 22.9\%$
	phi4-reasoning-plus	0.423 ± 0.177	0.438 ± 0.167	0.440 ± 0.191	$49.4\% \pm 16.9\%$
	gpt-oss-20b	0.505 ± 0.163	0.564 ± 0.124	0.556 ± 0.149	$30.0\% \pm 10.9\%$
	gemma3-27b	0.040 ± 0.040	0.048 ± 0.053	0.164 ± 0.134	$55.6\% \pm 29.7\%$
	gemma4-31b	$\underline{0.706 \pm 0.233}$	$\underline{0.732 \pm 0.203}$	$\underline{0.782 \pm 0.197}$	$\mathbf{3.9\% \pm 2.5\%}$
	olmo3-32b-think	0.569 ± 0.126	0.609 ± 0.105	0.677 ± 0.110	$15.0\% \pm 9.0\%$
	qwen3-32b	0.655 ± 0.138	0.678 ± 0.127	0.712 ± 0.132	$14.4\% \pm 11.2\%$
	deepseek-r1-llama-70b	0.412 ± 0.148	0.435 ± 0.164	0.527 ± 0.159	$28.3\% \pm 22.7\%$
	qwen3-coder-next	0.478 ± 0.192	0.507 ± 0.218	0.576 ± 0.147	$20.0\% \pm 13.2\%$
	gpt-oss-120b	$\mathbf{0.786 \pm 0.137}$	$\mathbf{0.796 \pm 0.137}$	$\mathbf{0.838 \pm 0.101}$	$\underline{7.8\% \pm 7.1\%}$
dcpbench	gpt-oss-20b	$\underline{0.193 \pm 0.061}$	$\underline{0.196 \pm 0.060}$	0.245 ± 0.053	$43.1\% \pm 14.4\%$
	deepseek-r1-llama-70b	0.078 ± 0.037	0.077 ± 0.037	0.202 ± 0.077	$39.5\% \pm 16.2\%$
	qwen3-coder-next	0.139 ± 0.083	0.142 ± 0.088	$\underline{0.269 \pm 0.147}$	$\underline{36.8\% \pm 23.0\%}$
	gpt-oss-120b	$\mathbf{0.324 \pm 0.098}$	$\mathbf{0.340 \pm 0.092}$	$\mathbf{0.400 \pm 0.094}$	$\mathbf{18.0\% \pm 17.4\%}$
satbench	gemma4-31b	$\underline{0.936 \pm 0.136}$	$\underline{0.934 \pm 0.138}$	$\underline{0.944 \pm 0.126}$	$\underline{0.1\% \pm 0.3\%}$
	qwen3-coder-next	0.648 ± 0.188	0.642 ± 0.188	0.682 ± 0.192	$13.5\% \pm 11.6\%$
	gpt-oss-120b	$\mathbf{0.963 \pm 0.008}$	$\mathbf{0.959 \pm 0.008}$	$\mathbf{0.981 \pm 0.006}$	$\mathbf{0.0\% \pm 0.1\%}$

including `sudoku` (Augustine et al., 2022), `road-r` (Giunchiglia et al., 2023), `sushi` (Kamishima, 2003), and `warcraft` (Pogančić et al., 2020). Unlike existing SAT and CP benchmarks, which typically provide a single fixed problem description, `auto-nesy-bench` pairs each task with natural language descriptions at multiple levels of granularity, together with annotated examples, so that it can be used to evaluate every stage of the auto-formalization pipeline, from constraint extraction through downstream NeSy prediction. Section D details how each problem was adapted and describes the granularity levels of the natural language descriptions and the corresponding dataset statistics.

5 EXPERIMENTAL RESULTS

We try to answer our **RQ1** and **RQ2** by evaluating several open-weight LLMs families: Qwen(-Coder) (Yang et al., 2025; Cao et al., 2026), Gemma-3/4 (Team et al., 2025; 2026), Mistral-Nemo (Jiang et al., 2023), DeepSeek-R1-Distill (Liu et al., 2024), Olmo (Groeneveld et al., 2024), Phi (Abdin et al., 2025), and GPT-OSS (OpenAI, 2025). Sizes vary from 8.2B to 117B parameters. Every LLM has been evaluated *unquantized*, except for GPT-OSS that comes natively quantized.

5.1 RQ1: AUTO-FORMALIZATION OF LOGICAL CONSTRAINTS

We structure our analysis along three main axes. First, we compare different LLMs to assess their *auto-formalization* capability. Second, we investigate how *prompt* configuration affects performance. Finally, we compare different *output formats* to identify the most effective for formalizing constraints.

Settings. For each benchmark, we consider two prompting strategies, i.e., zero-shot (ZS) and zero-shot chain-of-thought (ZS-CoT), and five output formats, i.e., `DIMACS`, `NAT`, `PySAT`, `CPMpy`, and `SymPy`, yielding ten configurations per LLM and benchmark. For `auto-nesy-bench`, we additionally evaluate two prompt variants with different levels of detail. Due to computational constraints, we evaluate a broader range of LLMs on `auto-nesy-bench`, our novel benchmark, while evaluating a subset of LLMs on `satbench` and `dcpbench`. This allows us to thoroughly evaluate our new benchmark while also providing validation across existing benchmarks. Due to space constraints, we defer additional experiments and `SPL` ablations using LLM-generated formulas from less detailed prompts to Section I, along with an analysis of `SL` with soft constraints in Section I.2.3.

Can LLMs formalize logical constraints? We average each LLM contribution across all output formats and prompting strategies. To make the comparison fair, we consider *only* the detailed prompt for `auto-nesy-bench`. Table 1 shows that model choice substantially affects auto-formalization performance. On `auto-nesy-bench`, `gpt-oss-120b` performs best, achieving an F1 of 0.786 (precision 0.796, recall 0.838), followed by `gemma4-31b` with an F1 of 0.706. These two LLMs

Table 2: **Detailed descriptions and CoT prompts improve auto-formalization performance across all benchmarks.** We compare detailed and natural descriptions on `auto-nesy-bench` and zero-shot (ZS) and zero-shot-chain-of-thought (ZS-CoT) prompts on `auto-nesy-bench`, `dcpbench`, and `satbench`, reporting F1, precision, recall, and syntax error rate, with mean ± standard deviation. Best results are highlighted in bold. See Section I for the complete results.

Dataset	Variant	Config.	F1 (↑)	Precision (↑)	Recall (↑)	Syntax ER (↓)
auto-nesy-bench	Description	Non-Detailed	0.330 ± 0.206	0.357 ± 0.218	0.464 ± 0.221	$\mathbf{25.3\%} \pm \mathbf{24.9\%}$
		Detailed	$\mathbf{0.455 \pm 0.277}$	$\mathbf{0.478 \pm 0.279}$	$\mathbf{0.532 \pm 0.263}$	$28.9\% \pm 25.7\%$
	Prompting	ZS	0.437 ± 0.262	0.463 ± 0.268	0.515 ± 0.243	$\mathbf{28.2\%} \pm \mathbf{24.7\%}$
		ZS-CoT	$\mathbf{0.472 \pm 0.290}$	$\mathbf{0.494 \pm 0.289}$	$\mathbf{0.548 \pm 0.281}$	$29.6\% \pm 26.6\%$
dcpbench	Prompting	ZS	0.176 ± 0.119	0.181 ± 0.124	$\mathbf{0.285 \pm 0.132}$	$\mathbf{33.7\%} \pm \mathbf{20.6\%}$
		ZS-CoT	$\mathbf{0.191 \pm 0.113}$	$\mathbf{0.197 \pm 0.118}$	0.272 ± 0.113	$35.0\% \pm 20.3\%$
satbench	Prompting	ZS	0.805 ± 0.240	0.801 ± 0.241	0.826 ± 0.233	$5.6\% \pm 10.9\%$
		ZS-CoT	$\mathbf{0.893 \pm 0.124}$	$\mathbf{0.889 \pm 0.125}$	$\mathbf{0.912 \pm 0.112}$	$\mathbf{3.5\%} \pm \mathbf{7.0\%}$

also exhibit the lowest syntax error rates, at 7.8% and 3.9%, respectively. In contrast, `gemma-3-27b` performs the worst in F1, achieving only 0.040 and a syntax error rate of 55.6%, indicating that a large fraction of its generated formalizations cannot be successfully parsed. No LLM reaches perfect performance, indicating that `auto-nesy-bench` remains a non-trivial challenge for these popular open-weight LLMs. On `satbench`, `gpt-oss-120b` achieves near-optimal performance, with an F1 of 0.963, followed by `gemma4-31b` with an F1 of 0.936. In contrast, performance drops substantially on `dcpbench`, where `gpt-oss-120b` achieves an F1 of only 0.324. Thus, `auto-nesy-bench` offers a challenging yet substantially more tractable setting than `dcpbench`. Across datasets and models, recall is generally higher than precision, suggesting that generated formulas tend to be permissive, admitting some invalid assignments rather than excluding valid ones.

How impactful is the prompt? To answer this question, we evaluate two possibilities: (i) the granularity of the prompt, and (ii) the introduction of CoT. To do that, we averaged each model and output modality. We evaluate (i) on `auto-nesy-bench`, as it is the *only* benchmark providing this level of granularity. As shown in Table 2 (Description), detailed descriptions substantially improve auto-formalization performance: increasing F1 from 0.330 to 0.455, precision from 0.357 to 0.478 and recall from 0.464 to 0.532. This improvement is unlikely to be attributed to easier parsing, as detailed prompts exhibit a higher syntax error rate, i.e., 28.9% vs 25.3% for non detailed descriptions. To evaluate (ii), we compare ZS and ZS-CoT prompting across all three benchmarks. As shown in Table 2 (Prompting), CoT improves F1: from 0.437 to 0.472 on `auto-nesy-bench`, 0.176 to 0.191 on `dcpbench`, and 0.805 to 0.893 on `satbench`. However, its impact on recall and syntax errors varies across datasets: recall decreases slightly on `dcpbench`, while syntax errors increase on `auto-nesy-bench` and `dcpbench` but decrease on `satbench`. Overall, CoT consistently improves F1, but its benefits are dataset dependent. Full prompts are reported in Section F.

How important is the LLM output format? We average each model-prompting configuration, using *only* the detailed `auto-nesy-bench` prompt for a fair comparison. As shown in Table 3, code-based formats (`PySAT`, `CPMpy`, `SymPy`) outperform direct `DIMACS` and `NAT` generation on `auto-nesy-bench` and `dcpbench`. On `auto-nesy-bench`, `CPMpy` achieves the highest F1 (0.575), precision (0.586), and recall (0.622), while `SymPy` ranks second in all three metrics. Both outperform `DIMACS` and `NAT` in F1 (0.364 and 0.331), with `PySAT` at 0.453. On `dcpbench`, `CPMpy` has again the highest F1 (0.247) and precision (0.251), while `DIMACS` has the highest recall (0.358); all code-based formats outperform `NAT` and `DIMACS` in F1. In contrast, `NAT` performs best on `satbench`, achieving an F1 of 0.920, followed by `SymPy` at 0.878. Performance does not correlate with syntax errors: `DIMACS` achieves the lowest error (20.5%, 21.3%, and 0.4%).

5.2 RQ2: Downstream Impact of LLM-Generated Constraints

The generated formula alone does not reveal its impact on a downstream NeSy predictor: a formula may differ substantially from the ground truth yet have little effect on prediction, or achieve a high F1 score while remaining unusable in practice. To address this, we embed LLM-generated formulas into an `SPL` and evaluate the resulting models, closing the loop between auto-formalization and downstream NeSy predictors. We performed this evaluation on `auto-nesy-bench` using

Table 3: **Code-based output formats generally achieve higher auto-formalization performance on `auto-nesy-bench` and `dcpbench`, whereas the `NAT` performs best on `satbench`.** We compare five output formats reporting F1, precision, recall, and syntax error, with mean ± standard deviation. Best results are in bold, and second-best results are underlined (see Section I for full results).

DATASET	FORMAT	F1 (↑)	PRECISION (↑)	RECALL (↑)	SYNTAX ER (↓)
auto-nesy-bench	DIMACS	0.364 ± 0.223	0.394 ± 0.231	0.467 ± 0.227	$\mathbf{20.5\% \pm 18.8\%}$
	NAT	0.331 ± 0.173	0.361 ± 0.178	0.457 ± 0.199	$31.8\% \pm 22.5\%$
	PySAT	0.453 ± 0.300	0.500 ± 0.310	0.498 ± 0.284	$31.1\% \pm 26.3\%$
	CPMpy	$\mathbf{0.575 \pm 0.299}$	$\mathbf{0.586 \pm 0.303}$	$\mathbf{0.622 \pm 0.280}$	$\underline{30.3\% \pm 28.2\%}$
	SymPy	$\underline{0.549 \pm 0.280}$	$\underline{0.550 \pm 0.281}$	$\underline{0.614 \pm 0.265}$	$30.8\% \pm 29.4\%$
dcpbench	DIMACS	0.147 ± 0.094	0.157 ± 0.106	$\mathbf{0.358 \pm 0.114}$	$\mathbf{21.3\% \pm 16.6\%}$
	NAT	0.109 ± 0.045	0.114 ± 0.051	0.218 ± 0.051	$45.4\% \pm 14.9\%$
	PySAT	$\underline{0.207 \pm 0.135}$	$\underline{0.212 \pm 0.139}$	0.251 ± 0.133	$\underline{28.4\% \pm 21.2\%}$
	CPMpy	$\mathbf{0.247 \pm 0.120}$	$\mathbf{0.251 \pm 0.123}$	$\underline{0.295 \pm 0.128}$	$34.9\% \pm 18.4\%$
	SymPy	0.206 ± 0.112	0.209 ± 0.117	0.272 ± 0.124	$41.8\% \pm 20.7\%$
satbench	DIMACS	0.699 ± 0.311	0.693 ± 0.313	0.726 ± 0.300	$\mathbf{0.4\% \pm 0.7\%}$
	NAT	$\mathbf{0.920 \pm 0.085}$	$\mathbf{0.914 \pm 0.089}$	$\mathbf{0.944 \pm 0.068}$	$\underline{1.0\% \pm 1.4\%}$
	PySAT	0.873 ± 0.138	0.869 ± 0.139	0.888 ± 0.135	$7.0\% \pm 11.6\%$
	CPMpy	0.875 ± 0.149	0.872 ± 0.149	0.886 ± 0.148	$9.4\% \pm 13.5\%$
	SymPy	$\underline{0.878 \pm 0.129}$	$\underline{0.875 \pm 0.130}$	$\underline{0.901 \pm 0.118}$	$4.9\% \pm 6.6\%$

Table 4: **Generated formulas, when available, achieve comparable downstream `SPL` performance to expert-provided formulas on `auto-nesy-bench`**. We report mean Δ F1/Δ Recall (REC) relative to the ground-truth; standard deviations across 5 seeds are omitted (see Section I). REL denotes the relation between predicted and ground-truth formulas: `equal` (=), `strict` (⇒), `permissive` (⇐), or `incomparable` (‖), indicating whether the predicted formula satisfying assignments are equal to, a subset of, a superset of, or neither. MC-R measures the ratio of satisfying assignments of the predicted formula to that of the ground-truth formula. Missing cells indicate: † degenerate formula (e.g., UNSAT), ⋆ no formula generated (no CNF), or § run failure (crash) (see Table 24).

	G-T		gpt-oss-120b				olmo3-32b-think				qwen3-8b			
TASK	F1 (↑)	REC (↑)	ΔF1	ΔREC	REL	MC-R	ΔF1	ΔREC	REL	MC-R	ΔF1	ΔREC	REL	MC-R
bdd-oia	0.58	0.57	+0.01	+0.01	=	1.00	†	†	†	†	+0.00	+0.01	=	1.00
bdd-oia-2	0.74	0.74	+0.01	+0.01	=	1.00	+0.01	+0.01	=	1.00	⋆	⋆	⋆	⋆
cebab	0.47	0.46	+0.00	+0.01	=	1.00	−0.01	−0.01	=	1.00	⋆	⋆	⋆	⋆
chx	0.54	0.52	−0.06	−0.04	=	1.00	§	§	§	§	†	†	†	†
cifar10	0.87	0.87	−0.07	−0.07	=	1.00	−0.06	−0.06	=	1.00	−0.06	−0.06	=	1.00
cifar100	0.35	0.41	−0.08	−0.09	=	1.00	−0.09	−0.10	=	1.00	−0.08	−0.09	=	1.00
cle4evr	1.00	1.00	−0.00	−0.00	=	1.00	−0.00	−0.00	=	1.00	−0.00	−0.00	=	1.00
fashion	0.92	0.92	−0.00	+0.00	=	1.00	−0.00	−0.00	=	1.00	−0.00	+0.00	=	1.00
kand-logic	0.70	0.70	−0.09	−0.09	=	1.00	⋆	⋆	⋆	⋆	†	†	†	†
kand-logic-2	0.81	0.81	−0.05	−0.05	=	1.00	−0.04	−0.04	=	1.00	−0.11	−0.11	⇒	0.02
mn-add-bin	0.98	0.98	−0.00	+0.00	=	1.00	†	†	†	†	+0.00	+0.00	=	1.00
mn-add	0.98	0.98	−0.01	−0.01	=	1.00	−0.00	+0.00	=	1.00	−0.00	−0.00	=	1.00
mn-mul-bin	0.98	0.98	−0.00	−0.00	=	1.00	−0.00	−0.00	=	1.00	§	§	§	§
mn-mul	0.98	0.98	−0.00	−0.00	=	1.00	−0.01	−0.01	=	1.00	−0.00	−0.01	=	1.00
road-r	0.31	0.30	−0.09	−0.07	‖	1.03	§	§	§	§	†	†	†	†
sudoku	0.79	0.79	−0.02	−0.02	=	1.00	⋆	⋆	⋆	⋆	+0.05	+0.05	=	1.00
sushi	0.29	0.30	−0.02	+0.00	=	1.00	+0.01	+0.02	=	1.00	+0.01	+0.01	=	1.00
warcraft	0.87	0.86	−0.03	−0.02	⇐	1.75	−0.01	−0.01	⇐	1.75	§	§	§	§

`gpt-oss-120b`, `olmo3-32b-think` and `qwen3-8b`, representing large, medium, and small models, respectively, all of which perform well on `auto-nesy-bench` (see Table 1). For each LLM, we use its best-performing configuration: `CPMpy` output with a detailed CoT prompt (see Table 2 and Table 3). For each (LLM, task) pair, we train an `SPL` using the LLM-generated formula as prior knowledge and compare it with the same backbone trained using the ground-truth constraint.

Discussion. Across models, Table 4 shows that LLM-generated formulas are, in most cases, both *usable*, i.e., the auto-formalization pipeline produces a syntactically valid and executable formula, and *exactly correct* (=), i.e., they yield the same set of valid assignments. Empirically, an `SPL` equipped with LLM-generated constraints achieves performance comparable to that obtained with the ground-truth formula, with average differences in Δ F1 and Δ recall below 0.03 and 0.02, respectively. The main exception is `kand-logic-2` with `qwen3-8b`, whose generated formula captures only 2% of valid assignments (MC-R = 0.02). In contrast, over-approximations (⇐), such as `warcraft` , have little effect on downstream predictions. This suggests that, in-distribution, the negative effect of

overly permissive formulas can be mitigated during training, as the model can learn to assign lower probability to output combinations that violate the ground-truth constraints. All formulas in Table 4 achieve perfect consistency (see Section I). Failures are also unevenly distributed across models. Table 4 shows that `gpt-oss-120b` produces a valid formula on every task ($0/18$ failures), while `olmo3-32b-think` and `qwen3-8b` fail on $6/18$ and $7/18$ tasks respectively, roughly evenly split between degenerate (†), unproduced ($\star$), and run-failure (§) formulas (see Table 24 for details).

6 Related Work

Constraint Modeling in LLMs. Recent research has explored LLMs as a tool to translate natural language descriptions into formal constraint, logic, or CP models (Tsouros et al., 2023; Michailidis et al., 2024; 2025b). This direction has also been explored through in-context and retrieval-augmented strategies for example selection (Michailidis et al., 2024), active-learning (Mechqrane et al., 2024; Mechqrane & Bessiere, 2026), fine-tuning (Shi et al., 2025), SAT solving (Hazra et al., 2025), and anomaly detection (Zhang & Jain, 2026). A parallel line uses LLMs as natural-language-to-symbolic translators paired with external solvers or provers, targeting propositional and first-order logic (Liu et al., 2026; Pan et al., 2025b;a), ontologies (Strader et al., 2024), or executable programs (Austin et al., 2021). Orthogonal approaches generate answer set programming (ASP) rules (Kareem et al., 2024), synthesize inductive logic programming benchmarks (Helff et al., 2026), and ground symbols from natural language descriptions (Stein et al., 2025). However, these works evaluate end-task accuracy or formalization quality, and none targets auto-formalization for NeSy predictors.

Knowledge Acquisition in NeSy. Knowledge acquisition is a well-studied problem in NeSy AI and statistical relational learning (De Raedt et al., 2021). Existing approaches fall into two settings: (i) learning explicit knowledge from data, and (ii) learning models that acquire explicit and interpretable knowledge. The former aims to recover symbolic knowledge from training examples using a predefined vocabulary, as in inductive logic programming and its probabilistic extensions (Muggleton & De Raedt, 1994; De Raedt & Kersting, 2008). The latter includes a growing body of NeSy models (Daniele et al., 2023; Tang & Ellis, 2023; Shindo et al., 2023; 2024; Wüst et al., 2024; Debot et al., 2024), which may be unreliable due to reasoning shortcuts (Bortolotti et al., 2025). Both settings assume that the predicate vocabulary is specified by a human expert or that the knowledge is learned from data. Instead, we study whether LLMs can formalize it from natural language descriptions.

7 Conclusion

To the best of our knowledge, ours is the first benchmark and end-to-end pipeline for LLM-based auto-formalization of NeSy predictors, using constraints either as a layer or training loss. We investigate its opportunities and limitations, showing that the best-performing LLMs can generate high-quality formulas that perform comparably to manually constructed ones in downstream NeSy predictors, and that their performance depends on the quality of the constraint specification. Our results further highlight which LLMs, prompting strategies, and output formats are most suitable for auto-formalization, while showing that smaller models can struggle substantially. More broadly, we see this as a step toward promoting NeSy AI adoption: our goal is not to replace domain experts, but to support them, since generated formulas remain interpretable and can be inspected and corrected.

Future Work. Our evaluation is restricted to propositional logic formulas with `SPL` and `SL` as representative NeSy approaches. We focus on formulas compatible with current NeSy pipelines, as handling substantially more complex constraints remains a general challenge in NeSy AI (Huang et al., 2021). We leave the effects of automatic symbol extraction (Stein et al., 2025) and their downstream consequences (Marconato et al., 2026) for future work. Additional future directions include evaluating auto-formalization with a broader range of NeSy predictors, including architectures that support first order logic, such as `DeepProbLog` (Manhaeve et al., 2018), `LogicTensorNetworks` (Rinaldi & Serafini, 2026; Badreddine et al., 2022), and ASP-based models (Aspis et al., 2022; Yang et al., 2020; Skryagin et al., 2023; Cunnington et al., 2024). Another direction is to study auto-formalization with more complex constraint formalisms, such as satisfiability modulo theory over linear real arithmetic (Kolb et al., 2018; De Smet et al., 2023; Kurscheidt et al., 2025; 2026), as well as uncertainty estimation (Marconato et al., 2024; Ruggieri & Pugnana, 2025; Bortolotti et al., 2026) and fine-tuning (Calanzone et al., 2025).

AI USE STATEMENT

In this work, we used generative AI tools to polish the manuscript text, debug, and support us in coding features that we discussed and subsequently validated. We did not use generative AI tools for the idea, experiment planning, metric definition, literature review, or other required disclosure tasks. We have reviewed all AI-assisted work. In particular, we checked LLM-generated code for potential bugs and misalignment between our ideas and its implementation, and tested it for correctness with the involvement of two authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

The authors believe that the manuscript does not raise any ethical concerns beyond those associated with prompting an LLM, using a NeSy predictor, or using any of the pre-existing datasets (Section D).

REPRODUCIBILITY STATEMENT

The full auto-formalization pipeline is described in Section 3, including the main processing steps and design choices, with the evaluation metrics defined in Section B and the error taxonomy used for qualitative analysis provided in Section C. All datasets (`satbench`, `dcpbench`, `auto-nesy-bench`) are described in detail in Section D, including their construction and relevant statistics, while the corresponding per-task problem statements, variable lists, and prompts are provided in Sections D.3.2, D.3.3 and F. Dataset and source licensing information is summarized in Table 9. The evaluated LLMs, model checkpoints, and decoding hyperparameters are listed in Sections E and G and Table 5 to make the experimental setup fully specified. For the experiments, the implementation details, hyperparameter search procedure, and backbone architectures are described in Sections 5.2 and H.1. Failed, incomplete, or otherwise unavailable runs are explicitly documented in Table 24 and Table 29. We report the full per-model, per-configuration results, additional experiments and ablations in Section I. Finally, the codebase is included in the supplementary materials and will be publicly released upon acceptance.

REFERENCES

Marah Abdin, Sahaj Agarwal, Ahmed Awadallah, Vidhisha Balachandran, Harkirat Behl, Lingjiao Chen, Gustavo de Rosa, Suriya Gunasekar, Mojan Javaheripi, Neel Joshi, et al. Phi-4-reasoning technical report. *arXiv preprint arXiv:2504.21318*, 2025.

Eldar D Abraham, Karel D'Oosterlinck, Amir Feder, Yair Gat, Atticus Geiger, Christopher Potts, Roi Reichart, and Zhengxuan Wu. Cebab: Estimating the causal effects of real-world concepts on nlp model behavior. *Advances in Neural Information Processing Systems*, 35:17582–17596, 2022.

Kareem Ahmed, Stefano Teso, Kai-Wei Chang, Guy Van den Broeck, and Antonio Vergari. Semantic Probabilistic Layers for Neuro-Symbolic Learning. In *NeurIPS*, 2022.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2623–2631, 2019.

Yaniv Aspis, Krysia Broda, Jorge Lobo, and Alessandra Russo. Embed2Sym - Scalable Neuro-Symbolic Reasoning via Clustered Embeddings. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*, pp. 421–431, 8 2022. doi: 10.24963/kr.2022/44. URL https://doi.org/10.24963/kr.2022/44.

Eriq Augustine, Connor Pryor, Charles Dickens, Jay Pujara, William Wang, and Lise Getoor. Visual sudoku puzzle classification: A suite of collective neuro-symbolic tasks. In *International Workshop on Neural-Symbolic Learning and Reasoning (NeSy)*, 2022.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini, and Michael Spranger. Logic tensor networks. *Artificial Intelligence*, 303:103649, 2022.

Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.

Samuele Bortolotti, Emanuele Marconato, Tommaso Carraro, Paolo Morettin, Emile Van Krieken, Antonio Vergari, Stefano Teso, and Andrea Passerini. A neuro-symbolic benchmark suite for concept quality and reasoning shortcuts. *Advances in neural information processing systems*, 37: 115861–115905, 2024.

Samuele Bortolotti, Emanuele Marconato, Paolo Morettin, Andrea Passerini, and Stefano Teso. Shortcuts and identifiability in concept-based models from a neuro-symbolic lens. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 173204–173243. Curran Associates, Inc., 2025. doi: 10.52202/085713-5763. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/fcdccd419c4dc471fa3b73ec97b53789-Paper-Conference.pdf.

Samuele Bortolotti, Emanuele Marconato, Andrea Pugnana, Andrea Passerini, and Stefano Teso. Concise and logically consistent conformal sets for neuro-symbolic concept-based models. *arXiv preprint arXiv:2605.18202*, 2026.

Diego Calanzone, Stefano Teso, and Antonio Vergari. Logically consistent language models via neuro-symbolic integration. In *International Conference on Learning Representations*, volume 2025, pp. 54495–54522, 2025.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. Qwen3-coder-next technical report, 2026. URL https://arxiv.org/abs/2603.00729.

Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls. In *IJCAI*, 2016.

Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence.*, 172(6-7):772–799, 2008.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Weixin Chen and Han Zhao. Understanding and improving adversarial robustness of neural probabilistic circuits. In *NeurIPS*, 2025.

Weixin Chen, Antonio Vergari, and Han Zhao. Can VLMs reason robustly? a neuro-symbolic investigation. *Transactions on Machine Learning Research*, 2026a. ISSN 2835-8856. URL https://openreview.net/forum?id=4y6jiE6Q60.

Weixin Chen, Simon Yu, Huajie Shao, Lui Sha, and Han Zhao. Neural probabilistic circuits: Enabling compositional and interpretable predictions through logical reasoning. *Machine Learning*, 115(9): 207, 2026b.

YooJung Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic modeling. Technical report, University of California, Los Angeles (UCLA), 2020.

Joseph Paul Cohen, Joseph D. Viviano, Paul Bertin, Paul Morrison, Parsa Torabian, Matteo Guarrera, Matthew P Lungren, Akshay Chaudhari, Rupert Brooks, Mohammad Hashir, and Hadrien Bertrand. Torchxrayvision: A library of chest x-ray datasets and models. In Ender Konukoglu, Bjoern Menze, Archana Venkataraman, Christian Baumgartner, Qi Dou, and Shadi Albarqouni (eds.), *Proceedings of The 5th International Conference on Medical Imaging with Deep Learning*, volume 172 of *Proceedings of Machine Learning Research*, pp. 231–249. PMLR, 06–08 Jul 2022. URL https://proceedings.mlr.press/v172/cohen22a.html.

Daniel Cunnington, Mark Law, Jorge Lobo, and Alessandra Russo. The role of foundation models in neuro-symbolic learning and reasoning. *arXiv preprint arXiv:2402.01889*, 2024.

Alessandro Daniele, Tommaso Campari, Sagar Malhotra, and Luciano Serafini. Deep symbolic learning: discovering symbols and rules from perceptions. In *IJCAI*, 2023.

Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM (JACM)*, 48:608–647, 2001.

Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

Tirtharaj Dash, Sharad Chitlangia, Aditya Ahuja, and Ashwin Srinivasan. A review of some techniques for inclusion of domain-knowledge into deep neural networks. *Scientific Reports*, 2022.

Luc De Raedt and Kristian Kersting. *Probabilistic Inductive Logic Programming*, pp. 1–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-78652-8. doi: 10.1007/978-3-540-78652-8_1. URL https://doi.org/10.1007/978-3-540-78652-8_1.

Luc De Raedt, Anton Dries, Tias Guns, and Christian Bessiere. Learning constraint satisfaction problems: An ilp perspective. In *Data Mining and Constraint Programming: Foundations of a Cross-Disciplinary Approach*, pp. 96–112. Springer, 2016.

Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. In *AAAI*, 2018.

Luc De Raedt, Sebastijan Dumančić, Robin Manhaeve, and Giuseppe Marra. From statistical relational to neural-symbolic artificial intelligence. In *IJCAI*, 2021.

Lennert De Smet, Pedro Zuidberg Dos Martires, Robin Manhaeve, Giuseppe Marra, Angelika Kimmig, and Luc De Raedt. Neural probabilistic logic programming in discrete-continuous domains. In *Uncertainty in Artificial Intelligence*, pp. 529–538. PMLR, 2023.

Nicola Debole, Pietro Barbiero, Francesco Giannini, Andrea Passerini, Stefano Teso, and Emanuele Marconato. If concept bottlenecks are the question, are foundation models the answer? *Machine Learning*, 115(5):97, 2026.

David Debot, Pietro Barbiero, Francesco Giannini, Gabriele Ciravegna, Michelangelo Diligenti, and Giuseppe Marra. Interpretable concept-based memory reasoning. In *NeurIPS*, 2024.

Vincent Derkinderen, Robin Manhaeve, Rik Adriaensen, Lucas Van Praet, Lennert De Smet, Giuseppe Marra, and Luc De Raedt. The deeplog neurosymbolic machine, 2026. URL https://arxiv.org/abs/2508.13697.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pp. 4171–4186, 2019.

Luca Di Liello, Pierfrancesco Ardino, Jacopo Gobbi, Paolo Morettin, Stefano Teso, and Andrea Passerini. Efficient generation of structured objects with constrained adversarial networks. *Advances in neural information processing systems*, 33:14663–14674, 2020.

Michelangelo Diligenti, Marco Gori, and Claudio Sacca. Semantic-based regularization for learning and inference. *Artificial Intelligence*, 2017.

Ivan Donadello, Luciano Serafini, and Artur D'Avila Garcez. Logic tensor networks for semantic image interpretation. In *IJCAI*, 2017.

Christoph Feichtenhofer, Haoqi Fan, Jitendra Malik, and Kaiming He. Slowfast networks for video recognition. In *2019 IEEE/CVF international conference on computer vision (ICCV)*, pp. 6201–6210. IEEE, 2019.

Eugene C Freuder and Richard J Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58 (1-3):21–70, 1992.

Artur d'Avila Garcez, Sebastian Bader, Howard Bowman, Luis C Lamb, Leo de Penning, BV Illuminoo, Hoifung Poon, and COPPE Gerson Zaverucha. Neural-symbolic learning and reasoning: A survey and interpretation. *Neuro-Symbolic Artificial Intelligence: The State of the Art*, 342:1, 2022.

Eleonora Giunchiglia and Thomas Lukasiewicz. Coherent hierarchical multi-label classification networks. *NeurIPS*, 2020.

Eleonora Giunchiglia, Mihaela Catalina Stoian, and Thomas Lukasiewicz. Deep learning with logical constraints. pp. 5478–5485, 2022. doi: 10.24963/IJCAI.2022/767. URL https://doi.org/10.24963/ijcai.2022/767.

Eleonora Giunchiglia, Mihaela Cătălina Stoian, Salman Khan, Fabio Cuzzolin, and Thomas Lukasiewicz. Road-r: the autonomous driving dataset with logical requirements. *Machine Learning*, 112(9):3261–3291, 2023.

Eleonora Giunchiglia, Alex Tatomir, Mihaela Cătălina Stoian, and Thomas Lukasiewicz. Ccn+: A neuro-symbolic framework for deep learning with requirements. *International Journal of Approximate Reasoning*, pp. 109124, 2024.

Eleonora Giunchiglia, Fergus Imrie, Mihaela van der Schaar, and Thomas Lukasiewicz. Machine learning with requirements: A manifesto. *Neurosymbolic Artificial Intelligence*, 1:NAI–240767, 2025. doi: 10.3233/NAI-240767. URL https://doi.org/10.3233/NAI-240767.

Andreas Grivas, Nikolay Bogoychev, and Adam Lopez. Low-rank softmax can have unargmaxable classes in theory but rarely in practice. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6738–6758, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.465. URL https://aclanthology.org/2022.acl-long.465/.

Andreas Grivas, Antonio Vergari, and Adam Lopez. Taming the sigmoid bottleneck: Provably argmaxable sparse multi-label classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 12208–12216, 2024.

Dirk Groeneveld, Iz Beltagy, Evan Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Raghavi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, Will Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah A. Smith, and Hannaneh Hajishirzi. Olmo: Accelerating the science of language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 15789–15809. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.841. URL https://doi.org/10.18653/v1/2024.acl-long.841.

Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Stevan Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1–3): 335–346, June 1990. ISSN 0167-2789. doi: 10.1016/0167-2789(90)90087-6.

Rishi Hazra, Gabriele Venturato, Pedro Zuidberg Dos Martires, and Luc De Raedt. Have large language models learned to reason? a characterization via 3-sat phase transition. *arXiv preprint arXiv:2504.03930*, 2025.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Lukas Helff, Ahmad Omar, Felix Friedrich, Antonia Wüst, Hikaru Shindo, Rupert Mitchell, Tim Woydt, Patrick Schramowski, Wolfgang Stammer, and Kristian Kersting. SLR: Automated synthesis for scalable logical reasoning. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 402–426, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.16. URL https://aclanthology.org/2026.acl-long.16/.

Nick Hoernle, Rafael Michael Karampatsis, Vaishak Belle, and Kobi Gal. Multiplexnet: Towards fully satisfied logical constraints in neural networks. In *AAAI*, 2022.

Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. Scallop: From probabilistic deductive databases to scalable differentiable reasoning. *NeurIPS*, 2021.

Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pp. 428–437, 2018. doi: 10.1007/978-3-319-94144-8_26. URL https://doi.org/10.1007/978-3-319-94144-8_26.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

David S Johnson and Michael A Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *CVPR*, 2017.

Toshihiro Kamishima. Nantonac collaborative filtering: recommendation based on order responses. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 583–588, 2003.

Irfan Kareem, Katie Gallagher, Manuel Borroto, Francesco Ricca, and Alessandra Russo. Using learning from answer sets for robust question answering with llm. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pp. 112–125. Springer, 2024.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations (ICLR)*, 2015.

Samuel Kolb, Stefano Teso, Andrea Passerini, and Luc De Raedt. Learning SMT(LRA) constraints using SMT solvers. In *IJCAI*, 2018.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Leander Kurscheidt, Paolo Morettin, Roberto Sebastiani, Andrea Passerini, and Antonio Vergari. A probabilistic neuro-symbolic layer for algebraic constraint satisfaction. In Silvia Chiappa and Sara Magliacane (eds.), *Proceedings of the Forty-first Conference on Uncertainty in Artificial Intelligence*, volume 286 of *Proceedings of Machine Learning Research*, pp. 2431–2471. PMLR, 21–25 Jul 2025. URL https://proceedings.mlr.press/v286/kurscheidt25a.html.

Leander Kurscheidt, Gabriele Masina, Roberto Sebastiani, and Antonio Vergari. The theory and practice of MAP inference over non-convex constraints. *CoRR*, abs/2602.08681, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pp. 740–755. Springer, 2014.

Marco Lippi and Paolo Frasconi. Prediction of protein β-residue contacts by markov logic networks with grounding-specific weights. *Bioinformatics*, 2009.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Xuan Liu, Dheeraj Kodakandla, Kushagra Srivastva, and Mahfuza Farooque. Veritrans: Fine-tuned llm-assisted nl-to-pl translation via a deterministic neuro-symbolic pipeline. *arXiv preprint arXiv:2604.10341*, 2026.

Jaron Maene and Luc De Raedt. Soft-unification in deep probabilistic logic. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Jaron Maene, Vincent Derkinderen, and Pedro Zuidberg Dos Martires. Klay: Accelerating arithmetic circuits for neurosymbolic ai. In *ICLR*, 2025.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. DeepProbLog: Neural Probabilistic Logic Programming. *NeurIPS*, 2018.

Emanuele Marconato, Stefano Teso, Antonio Vergari, and Andrea Passerini. Not all neuro-symbolic concepts are created equal: Analysis and mitigation of reasoning shortcuts. In *NeurIPS*, 2023.

Emanuele Marconato, Samuele Bortolotti, Emile van Krieken, Antonio Vergari, Andrea Passerini, and Stefano Teso. Bears make neuro-symbolic models aware of their reasoning shortcuts. In *Uncertainty in Artificial Intelligence (UAI)*, 2024.

Emanuele Marconato, Samuele Bortolotti, Emile van Krieken, Paolo Morettin, Elena Umili, Antonio Vergari, Efthymia Tsamoura, Andrea Passerini, and Stefano Teso. Symbol grounding in neuro-symbolic ai: A gentle introduction to reasoning shortcuts. *Journal of Artificial Intelligence Research*, 86, 2026. doi: 10.1613/jair.1.22389.

Giuseppe Marra and Ondřej Kuželka. Neural markov logic networks. In *Uncertainty in Artificial Intelligence*, 2021.

Younes Mechqrane and Christian Bessiere. Active constraint acquisition using large language models. *J. Artif. Intell. Res.*, 85, 2026. doi: 10.1613/JAIR.1.19277. URL https://doi.org/10.1613/jair.1.19277.

Younes Mechqrane, Christian Bessiere, and Ismail Elabbassi. Using Large Language Models to Improve Query-based Constraint Acquisition. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pp. 1916–1925, Jeju, South Korea, August 2024. doi: 10.24963/ijcai.2024/212. URL https://hal.science/hal-04740486.

Xiaoxia Meng, Xiaowei Wang, Shoulin Yin, and Hang Li. Few-shot image classification algorithm based on attention mechanism and weight fusion. *Journal of Engineering and Applied Science*, 70 (1):14, 2023.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL https://doi.org/10.7717/peerj-cs.103.

K. Michailidis, D. Tsouros, and T. Guns. Dcp-bench-open, 2025a. URL https://doi.org/10.5281/zenodo.17800138.

Kostis Michailidis, Dimos Tsouros, and Tias Guns. Constraint modelling with llms using in-context learning. In *CP*, volume 307 of *LIPIcs*, pp. 20:1–20:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

Kostis Michailidis, Dimos Tsouros, and Tias Guns. Cp-bench: Evaluating large language models for constraint modelling. In *28th European Conference on Artificial Intelligence (ECAI)*, volume 413 of *Frontiers in Artificial Intelligence and Applications*, pp. 861–868. IOS Press, 2025b.

Kostis Michailidis, Dimos Tsouros, and Tias Guns. Dcp-bench-open: Evaluating llms for constraint modelling of discrete combinatorial problems, 2026. URL https://arxiv.org/abs/2506.06052.

Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

Heimo Müller and Andreas Holzinger. Kandinsky patterns. *Artificial Intelligence*, 300:103546, 2021. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2021.103546. URL https://www.sciencedirect.com/science/article/pii/S0004370221000977.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Muyu Pan, Dheeraj Kodakandla, and Mahfuza Farooque. Fine-tuned large language models for logical translation: Reducing hallucinations with lang2logic. In *2025 International Symposium on Networks, Computers and Communications (ISNCC)*, pp. 1–4. IEEE, 2025a.

Muyu Pan, Matthew Walter, Dheeraj Kodakandla, and Mahfuza Farooque. Langsat: A novel framework combining nlp and reinforcement learning for sat solving. *arXiv preprint arXiv:2512.04374*, 2025b.

Marin Vlastelica Pogančić, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BkevoJSYPB.

Raffaele Pojer, Andrea Passerini, and Manfred Jaeger. Generalized reasoning with graph neural networks by relational bayesian network encodings. In Soledad Villar and Benjamin Chamberlain (eds.), *Proceedings of the Second Learning on Graphs Conference*, volume 231 of *Proceedings of Machine Learning Research*, pp. 16:1–16:12. PMLR, 27–30 Nov 2024. URL https://proceedings.mlr.press/v231/pojer24a.html.

Raffaele Pojer, Andrea Passerini, Kim G Larsen, and Manfred Jaeger. A neuro-symbolic approach for probabilistic reasoning on graph data. *arXiv preprint arXiv:2507.21873*, 2025.

Connor Pryor, Charles Dickens, Eriq Augustine, Alon Albalak, William Yang Wang, and Lise Getoor. Neupsl: Neural probabilistic soft logic. In Edith Elkind (ed.), *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, pp. 4145–4153. International Joint Conferences on Artificial Intelligence Organization, 8 2023. doi: 10.24963/ijcai.2023/461. URL https://doi.org/10.24963/ijcai.2023/461. Main Track.

Andrea Pugnana, Giovanni De Toni, Cesare Barbera, Roberto Pellungrini, Bruno Lepri, and Andrea Passerini. To ask or not to ask: Learning to require human feedback. *arXiv preprint arXiv:2510.08314*, 2025.

Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(6):1137–1149, 2017.

Davide Rinaldi and Luciano Serafini. sltn: Structural logic tensor networks, 2026. URL https://arxiv.org/abs/2608.11136.

Salvatore Ruggieri and Andrea Pugnana. Things machine learning models know that they don't know. In *AAAI*, pp. 28684–28693. AAAI Press, 2025.

Weichun Shi, Minghao Liu, Wanting Zhang, Langchen Shi, Fuqi Jia, Feifei Ma, and Jian Zhang. Constraintllm: A neuro-symbolic framework for industrial-level constraint programming. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 16010–16030, 2025.

Hikaru Shindo, Viktor Pfanschilling, Devendra Singh Dhami, and Kristian Kersting. (alpha)ilp: thinking visual scenes as differentiable logic programs. *Machine Learning*, 112(5):1465–1497, 2023.

Hikaru Shindo, Viktor Pfanschilling, Devendra Singh Dhami, and Kristian Kersting. Learning differentiable logic programs for abstract visual reasoning. *Machine Learning*, 113(11):8533–8584, 2024.

Arseny Skryagin, Wolfgang Stammer, Daniel Ochs, Devendra Singh Dhami, and Kristian Kersting. Neural-probabilistic answer set programming. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 19, pp. 463–473, 2022.

Arseny Skryagin, Daniel Ochs, Devendra Singh Dhami, and Kristian Kersting. Scalable neural-probabilistic answer set programming. *Journal of Artificial Intelligence Research*, 78:579–617, 2023.

Adam Stein, Aaditya Naik, Neelay Velingker, Mayur Naik, and Eric Wong. The road to generalizable neuro-symbolic learning should be paved with foundation models. *arXiv preprint arXiv:2505.24874*, 2025.

Jared Strader, Nathan Hughes, William Chen, Alberto Speranzon, and Luca Carlone. Indoor and outdoor 3d scene graph generation via language-enabled spatial ontologies. *IEEE Robotics and Automation Letters*, 9(6):4886–4893, 2024.

Hao Tang and Kevin Ellis. From perception to programs: regularize, overparameterize, and amortize. In *ICML*, 2023.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

Stefano Teso and Kristian Kersting. Explanatory interactive machine learning. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, pp. 239–245, 2019.

The april Lab. cirkit: a python framework to build, learn and reason about probabilistic circuits and tensor networks, October 2024. URL https://github.com/april-tools/cirkit.

Dimos Tsouros, Hélène Verhaeghe, Serdar Kadıoğlu, and Tias Guns. Holy grail 2.0: From natural language to constraint models. *arXiv preprint arXiv:2308.01589*, 2023.

Dimos Tsouros, Senne Berden, Steven Prestwich, and Tias Guns. Generalizing constraint models in constraint acquisition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 11362–11371, 2025.

Dimosthenis Tsouros, Senne Berden, and Tias Guns. Learning to learn in interactive constraint acquisition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 8154–8162, 2024.

Emile van Krieken, Thiviyan Thanapalasingam, Jakub Tomczak, Frank van Harmelen, and Annette Ten Teije. A-nesi: A scalable approximate method for probabilistic neurosymbolic inference. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 24586–24609. Curran Associates, Inc., 2023. doi: 10.52202/075280-1069. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/4d9944ab3330fe6af8efb9260aa9f307-Paper-Conference.pdf.

Emile Van Krieken, Samy Badreddine, Robin Manhaeve, and Eleonora Giunchiglia. Uller: A unified language for learning and reasoning. In *International Conference on Neural-Symbolic Learning and Reasoning*, pp. 219–239. Springer, 2024.

Antonio Vergari, Nicola Di Mauro, and Guy Van den Broeck. Tractable probabilistic models: Representations, algorithms, learning, and applications. *Tutorial at the 35th Conference on Uncertainty in Artificial Intelligence (UAI)*, 2019.

Antonio Vergari, YooJung Choi, Anji Liu, Stefano Teso, and Guy Van den Broeck. A compositional atlas of tractable circuit operations for probabilistic inference. *NeurIPS*, 2021.

Shuhei Watanabe. Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance. *arXiv preprint arXiv:2304.11127*, 2023.

Anjiang Wei, Yuheng Wu, Yingjia Wan, Tarun Suresh, Huanmi Tan, Zhanke Zhou, Sanmi Koyejo, Ke Wang, and Alex Aiken. SATBench: Benchmarking LLMs' logical reasoning via automated puzzle generation from SAT formulas. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 33832–33849. Association for Computational Linguistics, 2025.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Antonia Wüst, Wolfgang Stammer, Quentin Delfosse, Devendra Singh Dhami, and Kristian Kersting. Pix2code: Learning to compose neural visual concepts as programs. In Negar Kiyavash and Joris M. Mooij (eds.), *Proceedings of the Fortieth Conference on Uncertainty in Artificial Intelligence*, volume 244 of *Proceedings of Machine Learning Research*, pp. 3829–3852. PMLR, 15–19 Jul 2024. URL https://proceedings.mlr.press/v244/wust24a.html.

Antonia Wüst, Tim Tobiasch, Lukas Helff, Inga Ibs, Wolfgang Stammer, Devendra Singh Dhami, Constantin A Rothkopf, and Kristian Kersting. Bongard in wonderland: Visual puzzles that still make ai go mad? In *International Conference on Machine Learning*, pp. 68118–68142. PMLR, 2025.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *ArXiv*, abs/1708.07747, 2017.

Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Van den Broeck. A semantic loss function for deep learning with symbolic knowledge. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5498–5507. PMLR, 2018.

Yiran Xu, Xiaoyin Yang, Lihang Gong, Hsuan-Chu Lin, Tz-Ying Wu, Yunsheng Li, and Nuno Vasconcelos. Explainable object-induced action decision for autonomous vehicles. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Zhun Yang, Adam Ishay, and Joohyung Lee. Neurasp: Embracing neural networks into answer set programming. In *IJCAI 2020*, 2020.

Haoting Zhang and Shekhar Jain. Llm-assisted logic rule learning: Scaling human expertise for time series anomaly detection. *arXiv preprint arXiv:2601.19255*, 2026.

Zhi-Hua Zhou. Abductive learning: towards bridging machine learning and logical reasoning. *Science China. Information Sciences*, 62(7):76101, 2019.

A IMPLEMENTATION

All the experiments are implemented using Python 3.12.4 with PyTorch 2.11. LLM evaluation was performed using `vLLm` (Kwon et al., 2023) on different clusters:

Table 5: LLM evaluation with the vLLM backend across the clusters used in our experiments.

MODEL	LINK	CUDA	VLLM	GPUS

`qwen3-32b`
`mistral-nemo`
`phi4-reasoning-plus`
	`Qwen/Qwen3-32B`
	`mistralai/Mistral-Nemo-Instruct-2407`
	`microsoft/Phi-4-reasoning-plus`
		12.5.0	0.23.0	2× A100
				80 GB

`gemma3-27b` `gemma4-31b`	`google/gemma-3-27b-it` `google/gemma-4-31B-it`	12.6	0.23.0	1–2× RTX PRO 6000 Blackwell 96 GB (sm_120)
`olmo3-32b-think`	`allenai/Olmo-3-32B-Think`	12.6	0.23.0	4× A100
`qwen3-8b`	`Qwen/Qwen3-8B`	12.6	0.19.0	2× RTX 4090 24 GB
`gpt-oss-120b` `gpt-oss-20b`	`openai/gpt-oss-120b` `openai/gpt-oss-20b`	12.6	0.23.0	1× H200 NVL 141 GB
`qwen3-coder-next`	`Qwen/Qwen3-Coder-Next`	12.6	0.23.0	2× H200 NVL 141 GB
`deepseek-r1-llama-70b`	`deepseek-ai/DeepSeek-R1-Distill-Llama-70B`	12.6	0.23.0	4× RTX A6000 48 GB

`SPL` and `SL` experiments were done using two RTX 4090s. The implementations of `SPL` were taken verbatim from `cirkit` (The april Lab, 2024). Datasets were either taken from their original repository or generated. All the details are provided in Section D. All the tested models are open-weight, directly downloaded from HuggingFace. Hyperparameters are in Section G and Section H.

B METRICS

In this section, we go through all the metrics we used in our experiments.

B.1 AUTO-FORMALIZATION

Let $\hat{\phi}$ be the LLM-generated formula and ϕ^* the ground-truth formula over N propositional variables. The syntax error rate is computed (after the self-verification step, if present) as

$$\text{SyntaxErr} = \frac{|\{i : \hat{\phi}_i \text{ fails to parse}\}|}{n}. \tag{3}$$

Confusion-matrix counts are obtained via model counting (MC) (Chavira & Darwiche, 2008): exactly when the task has $\leq$ 20 variables, and with `ApproxMC` (Chakraborty et al., 2016) otherwise. Here, TP, FP, FN, and TN denote true positives, false positives, false negatives, and true negatives, respectively. For formulas without auxiliary variables, all four cells are counted independently:

$$\text{TP} = \text{MC}(\phi^* \wedge \hat{\phi}),\ \text{FP} = \text{MC}(\neg\phi^* \wedge \hat{\phi}),\ \text{FN} = \text{MC}(\phi^* \wedge \neg\hat{\phi}),\ \text{TN} = \text{MC}(\neg\phi^* \wedge \neg\hat{\phi}), \tag{4}$$

where $\neg\phi$ is encoded by `PySAT` Tseitin negation with auxiliary variables kept disjoint between the two negated formulas.

Formats such as `CPMpy` and `SymPy` may introduce Tseitin auxiliary variables A during conversion. Let $\hat{\phi}(X, A)$ denote the predicted encoding over primary variables X and auxiliary variables A, and let $\hat{\phi}_X \equiv \exists A., \hat{\phi}$ denote its projection onto X. Negating the encoding clause-by-clause is *not* equivalent to negating this projection: in general, $\neg\exists A., \hat{\phi} \not\equiv \exists A., \neg\hat{\phi}$, and the latter may admit spurious assignments. We therefore *never negate an encoding containing auxiliary variables*. Instead,

all reported quantities are computed from $\hat{\phi}_X$, with the projection performed *before* any negation or conjunction.

We compute only

$$T = \text{MC}_X(\phi^*), \quad P = \text{MC}_X(\hat{\phi}), \quad \text{TP} = \text{MC}_X(\phi^* \wedge \hat{\phi}), \tag{5}$$

and derive $\text{FP} = P - \text{TP}$, $\text{FN} = T - \text{TP}$, $\text{TN} = 2^N - T - P + \text{TP}$. When `ApproxMC` is used, each of the three calls is run at confidence $\delta/3$, so all T, P and TP counts hold jointly with probability $\geq 1 - \delta$ (union bound).

B.2 Downstream NeSy predictor

During our `SPL` and `SL` evaluation pipeline, we rely on standard supervised learning metrics such as F1 score, accuracy, precision and recall. Here, we describe additional metrics we used for a qualitative analysis of the downstream NeSy predictor. All the metrics are computed on a hold-out test set.

Consistency (Con). This metric measures whether the final maximum a posteriori prediction of the NeSy predictor complies with the ground-truth formula.

$$\text{Con} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}\{\hat{\boldsymbol{y}}_i \models \phi^*\}, \tag{6}$$

If the consistency is not 1.00, it indicates that some predicted assignments do not satisfy the constraints. Such predictions can be harmful to the downstream application.

Ground-truth vs predicted formula relationship (Rel). For each (model, task) pair, we classify the LLM-generated formula $\hat{\phi}$ against the task ground-truth formula ϕ^* into one of four exact relations. We test the two subset relations exactly:

$$\phi^* \subseteq \hat{\phi} \iff \phi^* \wedge \neg\hat{\phi} \equiv \bot, \qquad \hat{\phi} \subseteq \phi^* \iff \hat{\phi} \wedge \neg\phi^* \equiv \bot. \tag{7}$$

The pair $(\hat{\phi}, \phi^*)$ is then assigned one of four relations:

$$\text{rel}(\hat{\phi}, \phi^*) = \begin{cases} \texttt{equal} & \text{if } \phi^* \subseteq \hat{\phi} \text{ and } \hat{\phi} \subseteq \phi^* \quad (\hat{\phi} = \phi^*) \\ \texttt{permissive} & \text{if } \phi^* \subseteq \hat{\phi} \text{ and } \hat{\phi} \not\subseteq \phi^* \quad (\hat{\phi} \Leftarrow \phi^*) \\ \texttt{strict} & \text{if } \hat{\phi} \subseteq \phi^* \text{ and } \phi^* \not\subseteq \hat{\phi} \quad (\hat{\phi} \Rightarrow \phi^*) \\ \texttt{incomparable} & \text{otherwise} \quad (\hat{\phi} \parallel \phi^*) \end{cases} \tag{8}$$

`permissive` means the LLM formula admits every valid assignment plus some invalid ones (false positives only); `strict` means it excludes some valid assignments (false negatives, which permanently remove support to valid assignments from the predictive distribution); `incomparable` means both kinds of error occur simultaneously.

Model-count ratio (MC-R). We additionally report the ratio of the two formulas model counts

$$\text{MC-R}(\hat{\phi}, \phi^*) \;=\; \frac{\text{MC}(\hat{\phi})}{\max(\text{MC}(\phi^*),\, 1)}, \tag{9}$$

which quantifies *how much* more (or less) permissive $\hat{\phi}$ is even when the qualitative relation is not `equal`, e.g., $\text{mc_ratio} = 1.75$ for a `permissive` formula means it admits 75% more satisfying assignments than the ground-truth.

C Error Taxonomy

When evaluating auto-formalization, we use error not only as a means of analysis, but also as feedback for self-correction (Teso & Kersting, 2019). We partition errors into two categories: *general*, comprising errors inherent to the resulting `DIMACS` representation regardless of the source format, and *generation*, comprising errors introduced by the particular process used to generate the `DIMACS`.

General (applies to `DIMACS` directly, and to `NAT`, and to `PySAT`/ `CPMpy`/`SymPy` executed output):

CODE	TRIGGER
`no_cnf_tags`	Output missing `<cnf>...</cnf>` tags
`invalid_header`	`p cnf N M` line absent or malformed
`clause_no_zero`	Clause does not terminate with `0`
`empty_clause`	A clause contains only `0`
`invalid_content`	Non-integer content in a clause body
`missing_num_vars`	Header present but N missing or zero
`clause_count_mismatch`	Declared clause count M disagrees with body
`literal_out_of_range`	A literal exceeds $[-N, N] \setminus \{0\}$
`formula_too_complex`	`NAT` to CNF conversion blows up
`zero_literal_in_clause`	`DIMACS` 0 delimiter not at the end of the clause

Generation (format-specific, precedes or replaces the `DIMACS` layer):

FORMAT	CODE	TRIGGER
`PySAT`	`execution_error`	Program raised an error when executed
	`timeout_error`	Program execution exceeded the time limit
	`encoder_aux_collision`	PySAT encoder auxiliary-variable index collision risk
`CPMpy`	`execution_error`	Program raised an error when executed
	`timeout_error`	Program execution exceeded the time limit
	`unknown_variable`	Referenced variable outside the declared mapping
	`non_boolean_variable`	Declared variable is not Boolean
`SymPy`	`execution_error`	Program raised an error when executed
	`timeout_error`	Program execution exceeded the time limit
	`unknown_variable`	Referenced variable outside the declared mapping
`NAT`	`no_cnf_tags`	Output missing `<cnf>...</cnf>` tags
	`empty_formula`	Nothing between the tags
	`parse_error`	Expression fails to parse
	`invalid_operator`	Operator other than AND/OR/NOT (any case)
	`empty_clause`	Empty clause `()` in the expression
	`unknown_variable`	A formula symbol outside the expected mapping

A single output may trigger more than one code. The *preamble auto-fix* step repairs `missing_num_vars` and `clause_count_mismatch` by recomputing the header from the clause bodies directly, without an additional LLM query; it applies wherever the `DIMACS`- syntax layer applies (`DIMACS` and the compiled output of `PySAT`/`CPMpy`/ `SymPy`), not to `NAT`. Such *triggers* will be contained in the self-verification step for guiding the LLM in the correction.

D DATASETS

In this section, we describe all the datasets used during our evaluation. A comparison about dataset statistics, specically, clause and variable counts is provided in Table 6.

Table 6: **`auto-nesy-bench` is NeSy-ready, while existing benchmarks are not**. `satbench` contains many small problems, whereas `dcpbench` contains fewer but substantially larger problems. `auto-nesy-bench` offers a balanced mix and, importantly, includes input-output annotations for use with NeSy predictors. Variables and clauses are reported as MIN / MAX / MEAN.

DATASET	PROBLEMS	VARIABLES			CLAUSES			NESY-READY
		MIN	MAX	MEAN	MIN	MAX	MEAN	
`satbench` (Wei et al., 2025)	1,041	5	90	36	4	50	20	✗
`dcpbench` (Michailidis et al., 2025b)	85	8	1,102,080	58,673	20	7,095,776	394,204	✗
`auto-nesy-bench` (ours)	18	9	102	32	17	129,648	7,966	✓

D.1 SATBENCH

`satbench` (Wei et al., 2025) is a benchmark for evaluating the logical reasoning capabilities of LLMs through natural language descriptions of problems generated from CNF formulas. Each

instance pairs a Boolean formula with a natural language scenario and descriptions of its variables, allowing models to reason about the underlying logical constraints expressed in natural language and determine whether the constraints are satisfiable.

For our auto-formalization setting, we repurpose these paired natural language descriptions and reference CNF formulas to evaluate whether models can recover the underlying logical constraints from natural language. We use the LLM4Code/SATBench Hugging Face dataset pinned at revision `186740e`. The original dataset contains 2,100 instances, evenly split between SAT (1,050) and UNSAT (1,050) formulas. We discard the latter because an UNSAT formula has no valid use for a NeSy predictor. Additionally, we remove 9 SAT malformed instances (mismatch between the declared variable/clause count). This leaves 1,041 instances for evaluation.

Across these instances, the number of clauses ranges from 4 to 50, and the number of variables ranges from 5 to 90. Each instance specifies Boolean variables indexed by coordinates (e.g., `x_0_0`), together with a natural language problem description and variable descriptions. An example is provided in Table 7.

Table 7: **Example from `satbench`.** Each `satbench` instance pairs a natural language scenario and variable descriptions with a reference Boolean formula. Here, five community-garden volunteers, Ana, Ben, Carla, Dave, and Emma, decide whether to sign up for weekend watering duty, where x_i indicates whether volunteer i signs up, in the order just given. Each row shows one natural-language constraint and its corresponding CNF clause and DIMACS encoding. The reference formula is the conjunction of all clauses. In our auto-formalization setting, the natural language scenario and constraints serve as input, and the conjunction of all constraints serves as the reference formalization.

NATURAL-LANGUAGE CONSTRAINT	CNF CLAUSE	DIMACS ENCODING
Ana signs up for watering duty.	x_0	`1 0`
Either Ben does not sign up, or Ana does not sign up.	$\neg x_1 \vee \neg x_0$	`-2 -1 0`
Either Ana does not sign up, or Emma signs up.	$\neg x_0 \vee x_4$	`-1 5 0`
Either Carla does not sign up, or Ben signs up.	$\neg x_2 \vee x_1$	`-3 2 0`

D.2 DCPBENCH-OPEN

DCP-Bench-Open (Michailidis et al., 2025b;a) contains 164 CP problems, including puzzles, scheduling tasks, and CSPLib instances. An input instance consists of a natural language description, a CP instance, a reference `CPMpy` model, and an example solution. Unlike `satbench`, these reference models can contain arithmetic constraints.

To make this benchmark suitable for an auto-formalization task, we instantiate the reference `CPMpy` models with their instance data and convert the constraints into CNF using the `to_cnf`[2] `CPMpy` function, encoding the variables as one-hot. As they cannot be used by a NeSy predictor, we exclude optimization problems, constraints that result in an error during CNF conversion, and conversions that exceed a 90 s timeout due to scalability issues (Huang et al., 2021). The experiments reported in the main paper use a subset of 85 problems with usable `DIMACS` targets. To avoid introducing additional bias, we remove the problem name from the natural language description. To make the instances compatible with `satbench` and `auto-nesy-bench`, we explicitly specify the Boolean variables by enumerating them and stating their value ranges, e.g., `1-101: men: integer, domain [0, 100].`

Example (`room_assignment`). The natural language description presents a room assignment problem. Specifically, we are asked to assign four room requests to five rooms, numbered 0 to 4, with one room per request and no overlapping requests assigned to the same room. The concrete instance has start dates May 1, 2, 3, and 4, and end dates May 5, 6, 7, and 8, respectively, all in 2024. The first request is preassigned to room 3. We write r_i to denote the room assigned to request i, the CP constraints are:

$$r_i \in \{0, \ldots, 4\} \quad (i = 0, \ldots, 3), \qquad r_0 = 3, \qquad \text{AllDifferent}(r_0, r_1, r_2, r_3). \tag{10}$$

The actual variable declaration supplied in the prompt is:

[2] https://cpmpy.readthedocs.io/en/latest/api/transformations/to_cnf.html

```
1-5: room_assignments_0: integer, domain [0, 4]
6-10: room_assignments_1: integer, domain [0, 4]
11-15: room_assignments_2: integer, domain [0, 4]
16-20: room_assignments_3: integer, domain [0, 4]
```

Let $b_{i,v}$ mean $r_i = v$, an equivalent CNF over these 20 variables is

$$\bigwedge_{i=0}^{3}\left[\left(\bigvee_{v=0}^{4} b_{i,v}\right) \wedge \bigwedge_{0\le v<w\le 4} (\neg b_{i,v} \vee \neg b_{i,w})\right]$$
$$\wedge\, b_{0,3} \wedge \bigwedge_{0\le i<j\le 3} \bigwedge_{v=0}^{4} (\neg b_{i,v} \vee \neg b_{j,v}).$$

The first line enforces exactly one room per request; the second enforces the pre-assignment and prevents room sharing.

D.3 AUTO-NESY-BENCH

`auto-nesy-bench` consists of a set of 18 NeSy tasks. These are established NeSy visual-reasoning benchmarks and vary from single-digit to hundred-plus-variable formulas. Table 8 lists every task with its ground-truth formula size.

Table 8: NeSy tasks and their ground-truth `DIMACS` sizes.

TASK	VARIABLES	CLAUSES
`chx` (Cohen et al., 2022)	9	27
`fashion` (Xiao et al., 2017)	10	46
`bdd-oia-2` (Xu et al., 2020)	11	17
`cle4evr` (Johnson et al., 2017)	12	28
`cifar10` (Krizhevsky et al., 2009)	15	157
`mn-add-bin` (Manhaeve et al., 2018)	13	512
`mn-mul-bin` (Bortolotti et al., 2024)	15	712
`sushi` (Kamishima, 2003)	16	56
`kand-logic-2` (Müller & Holzinger, 2021)	24	1328
`warcraft` (Ahmed et al., 2022)	24	289
`bdd-oia` (Xu et al., 2020)	25	31
`cebab` (Bortolotti et al., 2026)	25	1070
`kand-logic` (Müller & Holzinger, 2021)	36	129648
`mn-add` (Manhaeve et al., 2018)	39	364
`road-r` (Giunchiglia et al., 2023)	41	243
`sudoku` (Augustine et al., 2022)	64	400
`cifar100` (Krizhevsky et al., 2009)	100	4951
`mn-mul` (Bortolotti et al., 2024)	102	3514

`chx`. We use `ChestX-Rays` (Cohen et al., 2022), following the preprocessing of Pugnana et al. (2025). The dataset consists of images of X-ray scans, annotated with four binary symptoms: `Fracture`, `Pneumothorax`, `Airspace opacity`, and `Nodule/mass`, forming the concept vector $\boldsymbol{c} \in \{0,1\}^4$. We adopt the logical formalization introduced in Bortolotti et al. (2026), where the target label is determined solely by the total number of present symptoms, i.e., $\sum_{i=1}^{4} \boldsymbol{c}_i$, mapping to five severity levels (`healthy`, `green_code`, `yellow_code`, `orange_code`, and `red_code`). The resulting dataset contains 3,150 training, 350 validation, and 876 test samples.

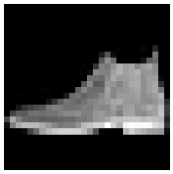


Figure 2: A sample from the `chx` dataset.

`fashion` & `cifar100`. Following (Xu et al., 2018), we formalized `FashionMNIST` (Xiao et al., 2017) and `CIFAR100` (Krizhevsky et al., 2009) as one-hot prediction task. Formally,

$$\boldsymbol{y}_k = \mathbb{1}\{\boldsymbol{k} = \boldsymbol{y}\}. \tag{11}$$

`FashionMNIST` consists of 70,000 28×28 grayscale images of garments in 10 classes (60,000 train / 10,000 test); `CIFAR100` consists of 60,000 32×32 colored images evenly distributed over 100 fine-grained classes (600 images per class, standard 50,000/10,000 train/test split). In both cases, we hold out 10,000 training images for validation.



(a) `fashion`



(b) `cifar100`

Figure 3: Samples from `fashion` and `cifar100` datasets.

`cifar10`. We use `CIFAR-10` (Krizhevsky et al., 2009), following the preprocessing and logical formalization of Bortolotti et al. (2026). The original dataset contains $60,000$ images from 10 classes; we use $35,000$ training, $15,000$ validation, and $10,000$ test samples. Each image is associated with a binary concept vector $\mathbf{c} \in \{0,1\}^7$, encoding the semantic attributes `ani`, `wng`, `met`, `whl`, `hai`, `hrn`, and `snt`. Class labels are deterministically inferred from these concepts through the logical rules of Bortolotti et al. (2026). The concept-label mapping is intentionally non-injective, making the pairs `automobile`/`truck` and `dog`/`horse` indistinguishable from concepts alone.

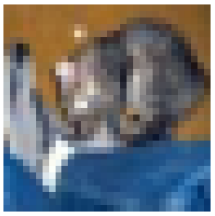


Figure 4: A sample from the `cifar10` dataset.

`bdd-oia` & `bdd-oia-2`. `BDD-OIA` is an autonomous driving task consisting of frames retrieved from driving scene videos (Xu et al., 2020; Bortolotti et al., 2024). We provide two versions of this dataset in order to encode different levels of granularity. Originally, each frame comes with four binary labels (`move_forward`, `stop`, `turn_left`, and `turn_right`) and 21 binary concepts that provide an explanation for the selected actions. The first version adopts a similar prior knowledge defined in Marconato et al. (2023), which we briefly summarize here for completeness. For the

move_forward and stop actions, the following logical rules are used:

$$\begin{aligned} O &:= \texttt{car} \vee \texttt{person} \vee \texttt{rider} \vee \texttt{other_obstacle}, \\ F &:= \texttt{green_light} \vee \texttt{follow} \vee \texttt{road_clear}, \\ S &:= \texttt{red_light} \vee \texttt{traffic_sign} \vee O, \\ \texttt{move_forward} &\iff F \wedge \neg S, \\ \texttt{stop} &\iff S. \end{aligned} \tag{12}$$

For the turning actions, the following rules define the conditions for executing a left turn:

$$\begin{aligned} L^+ &:= \texttt{left_lane} \vee \texttt{left_green_light} \vee \texttt{left_follow}, \\ L^- &:= \texttt{no_left_lane} \vee \texttt{left_obstacle} \vee \texttt{left_solid_line}, \\ \texttt{turn_left} &\iff L^+ \wedge \neg L^-. \end{aligned} \tag{13}$$

The rules for turn_right are defined analogously by replacing the left-related concepts with their corresponding right-related counterparts. The symbols O, F, S, L^+, and L^- are abbreviations, not additional benchmark variables. The second version of the dataset instead simply filters the concepts and action maintaining only those related to the move_forward and stop actions. The dataset itself consists of 16k fully annotated training frames, along with 2k and 4.5k labeled frames for validation and testing, respectively. Instead of processing the full frames, we rely on the embeddings of a pretrained Faster-RCNN (Ren et al., 2017) provided by Bortolotti et al. (2024).

mn-add & mn-add-bin. MN-Add, originally introduced in Manhaeve et al. (2018), is a canonical benchmark for NeSy AI. Each sample consists of two MNIST (LeCun, 1998) digit images $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ representing digits $(\boldsymbol{c}_1, \boldsymbol{c}_2) \in [0, 9]^2$, with the target defined as their arithmetic sum:

$$\boldsymbol{y} = \boldsymbol{c}_1 + \boldsymbol{c}_2. \tag{14}$$

The dataset contains 54k training, 6k validation, and 10k test examples. We provide two versions of this task, one in which the digits are encoded as one-hot, and another in which the digits are encoded as a single binary value.



Figure 5: A sample from the mn-add/mn-mul dataset.

mn-mul & mn-mul-bin. MN-Mul is a variant of MN-Add (Manhaeve et al., 2018). Each sample consists of two MNIST (LeCun, 1998) digit images $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ representing digits $(\boldsymbol{c}_1, \boldsymbol{c}_2) \in [0, 9]^2$, with the target defined as their arithmetic product:

$$\boldsymbol{y} = \boldsymbol{c}_1 * \boldsymbol{c}_2. \tag{15}$$

The dataset contains 54k training, 6k validation, and 10k test examples. We provide two versions of this task, one in which the digits are encoded as one-hot, and another in which the digits are encoded as a single binary value.

sushi. We use the SUSHI3 tabular preference dataset (Kamishima, 2003), in which 5,000 respondents each provide a total order over 10 sushi varieties, and adopt the ranking formalization of Xu et al. (2018). Each order is split into a 6-item context sub-ranking (conditioning input) and a 4-item label sub-ranking. The label is encoded as a 4×4 permutation matrix: variable $\boldsymbol{x}_{ij}$ is true iff item i is assigned rank j, for $i, j \in \{0, 1, 2, 3\}$ (16 Boolean variables). A valid assignment is exactly a permutation matrix, i.e., every item receives exactly one rank and every rank is taken by exactly one item. We split the 5,000 respondents into 4,000/500/500 for train/validation/test.

kand-logic & kand-logic-2. Kand-Logic is a two-image relational variant of the Kandinsky-pattern benchmark (Müller & Holzinger, 2021), provided through rsbench (Bortolotti et al., 2024). Each sample consists of two images, each containing three geometric primitives; every primitive carries a shape (square, triangle, circle) and a color (red, blue, yellow),

each being an one-hot vector, i.e., 6 variables per primitive, 18 per image, 36 in total. The pair is positive iff both images realize the same *pattern category* on shape or on color, where a category is one of `same` (all three primitives equal), `pair` (exactly two equal), or `diff` (all three distinct). We also include `kand-logic-2`, which uses two primitives per image (24 variables): with only two primitives, the `pair` category is undefined and `same`/`diff` become complementary. Training data is produced by running `rsbench SyntheticKandinskyGenerator` to emit single-figure images, pairing images that share a pattern category, and verifying every sample against the ground-truth formula before writing it; we generate 800/100/100 train/validation/test samples per variant.



(a) `kand-logic-2` (b) `kand-logic`

Figure 6: Samples from `kand-logic-2`/`kand-logic` datasets.

`sudoku`. We use a 4×4 `VisualSudoku` task in the spirit of Augustine et al. (2022): each cell of a 4×4 grid is rendered with an `MNIST` image (1 to 4), and the model must decide whether the depicted grid is a valid solution. The one-hot encoding uses variable $\boldsymbol{x}_{ijv}$ for "cell (i, j) holds digit v" with $i, j \in \{0, \ldots, 3\}$ and $v \in \{1, \ldots, 4\}$ (64 Boolean variables). An assignment is valid iff it is a complete solution: exactly one digit per cell and each digit appearing exactly once in every row, every column, and every 2×2 box. We use 4,000/500/500 train/validation/test grids drawn from the 288 valid boards (with random digit relabelings).



Figure 7: A sample from the `sudoku` dataset.

`road-r`. `ROAD-R` (Giunchiglia et al., 2023) is an extension of the ROAD autonomous driving dataset with a set of hand-written logical requirements over the label space. Each frame is annotated with an agent class (one of 10), a traffic-light color (`red`/`amber`/`green`, at most one), a set of actions (`16`), and a set of locations (`12`), which result in 41 Boolean variables (1-10 agent, 11-13 color, 14-29 action, 30-41 location). Unlike the other tasks, the target CNF is not compiled from a Python program: it is the requirements are provided by the authors (41 variables, 243 clauses). Samples are 8-frame clips ending at each annotated key-frame. It consists of $\sim$47k samples, split 80/10/10 by source video.

`warcraft`. We use a 4×4 `Warcraft` shortest-path task in the spirit of Pogančić et al. (2020). For simplicity, maps are procedurally generated: each of the 16 grid cells is assigned one of 5 terrain classes i.i.d. uniformly at random, then rendered with a real `Warcraft` II terrain tile of the matching class. The model must decide whether the depicted set of grid edges is the minimum-cost simple path from the top-left cell $(0, 0)$ to the bottom-right cell $(3, 3)$. The encoding uses one Boolean variable per grid edge, i.e., 24 Boolean variables in total. An assignment is valid iff it is a complete solution: the source and sink each have exactly one selected incident edge and they trace a single simple path between the two corners. Each of the 5 terrain classes $k \in \{0, \ldots, 4\}$ carries a per-cell traversal

cost $c_k = 4^k$ (i.e., $1, 4, 16, 64, 256$). The ground-truth optimal path is computed in advance. We use 8,000/1,000/1,000 train/validation/test grids, each rendered as a 64×64 image.

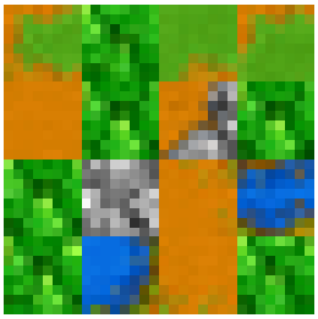


Figure 8: A sample from the `warcraft` dataset.

`cebab`. We use `CEBaB` (Abraham et al., 2022), a dataset of restaurant reviews annotated via human-mediated counterfactual edits along four aspects: `food`, `service`, `noise`, and `ambiance`, each labeled with a majority sentiment (`Positive`, `Negative`, `Neutral`, or `unknown`), forming four one-hot arrays. We adopt the logical formalization of Bortolotti et al. (2026), which adds a fifth concept, `review_majority`, obtained from the original 1 to 5 star rating into the same four values, giving $\boldsymbol{c} \in \{0,1\}^{20}$. The target label is a one-hot vote over the five concepts (`positive`, `negative`, `neutral`, `unknown`, `conflict`): the majority class among the five wins, higher-priority classes break ties in their favor, and a `positive`/`negative` tie at the overall maximum yields `conflict`. We use the CEBaB v1.1 observational split: 1,755 training, 1,673 validation, and 1,689 test reviews.

Sample from `cebab`.

```
Review text:
    "Limoncello has excellent Italian food. So good,
    that it tastes homemade. The only drawback is that
    it is so loud in the restaurant."

GROUND-TRUTH ANNOTATION:
    food     aspect majority = Positive
    service  aspect majority = unknown
    noise    aspect majority = Negative
    ambiance aspect majority = unknown
    review   majority        = Positive   (raw star rating: "4")
```

`cle4evr`. `CLE4EVR` is a two-object relational variant of the `CLEVR` visual reasoning benchmark (Johnson et al., 2017), provided through `rsbench` (Bortolotti et al., 2024). Each image depicts exactly two 3D-rendered objects; every object carries a shape (`cube`, `sphere`, `cylinder`) and a color (`red`, `blue`, `green`), each a 3-way one-hot vector with 6 variables per object, 12 in total. An assignment is valid iff the two objects share the same shape *and* the same color. Scenes are rendered via Blender/Cycles, with both objects constructed to match by construction so every sample satisfies the ground-truth formula by design. We generate 5,000/2,000/5,000 train/validation/test images.

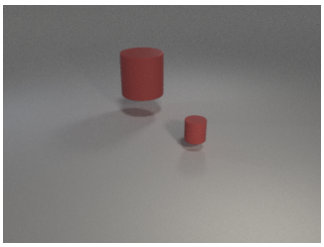


Figure 9: A sample from the `cle4evr` dataset.

Construction of the target `DIMACS`. We use `PySAT` (Ignatiev et al., 2018) to compile each task validity predicate into an equivalent `DIMACS` CNF over the variable layout described above. Every compiled formula is cross-checked against a Python oracle, exhaustively when the variable count permits ($\leq 2^{20}$ assignments).[3] `road-r` is the only exception: its CNF is taken unchanged from its dataset rather than compiled.

Constructing the task descriptions. Each task is presented to the LLM using a fixed prompt scaffold (Section F), instantiated with two hand-written components: a *constraint description* and a *variable list*. We provide the constraint description in two variants with different levels of granularity. The *detailed* description presents each rule and explicitly lists the background facts that should *not* be encoded. The *not detailed* description details the same intended task more concisely in ordinary prose, leaving more of the rule decomposition to the model and, in some cases, leaving constraints or modeling conventions implicit (e.g., "the car may move forward iff it is allowed to proceed and the road ahead is clear"). Full prompts are shown in Section D.3.2. Both descriptions, as well as the per-variable descriptions, were independently written by two of the authors, who first impersonated an expert and then a non-expert who at least knew the task they wanted to solve. The authors then compared their versions clause by clause and reconciled all discrepancies until reaching full agreement; the released texts are the resulting consensus versions. Thus, the constraint-description explicitness is the only factor that differs between the natural and formalized conditions: the surrounding prompt template, variable numbering, and injected variable list are byte-identical across conditions.

D.3.1 Licensing and Redistribution

We plan to release the benchmark under a two-license split, following common practice for mixed-provenance NeSy and vision benchmarks: one license for our own code, and one for the data artifacts we redistribute. We plan to distribute the code under the `BSD-3-Clause`[4] license.

Redistributed data artifacts. We plan to release labels, CNF targets, and any bundled images or features under `CC BY-NC-SA 4.0`[5]. This is the most restrictive license among the datasets in `auto-nesy-bench` and is inherited from `ROAD-R` (Giunchiglia et al., 2023), which is released under `CC BY-NC-SA 4.0`. All other bundled datasets are distributed under licenses that are equally or more permissive (`MIT`[6], `BSD-3-Clause`, `CC BY 4.0`, or unrestricted). The datasets `mn-add`, `mn-mul` (and their `-bin` variants), `kand-logic`, `kand-logic-2`, `bdd-oia`, and `bdd-oia-2` are sourced from `rsbench`[7] through its companion codebase, `rsbench-code`[8] (Bortolotti et al., 2024). Detailed licensing information for all datasets is provided in Table 9.

Datasets not redistributed. For the datasets marked ✗ above, the released codebase cannot bundle the data. Instead, it will provide a download script that retrieves each dataset directly from its original source when first needed.

[3] All formulas were independently written and consistently checked by two of the authors.
[4] https://opensource.org/license/bsd-3-clause
[5] https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en
[6] https://opensource.org/license/mit
[7] https://unitn-sml.github.io/rsbench
[8] https://github.com/unitn-sml/rsbench-code

Table 9: **License and redistribution status of every `auto-nesy-bench` datasets**. "Redistributed" indicates whether the data itself can be shipped in a new benchmark, as opposed to being fetched from the original source.

DATASET	TASK(S)	LICENSE	REDISTRIBUTED
MNIST (LeCun, 1998)	`mn-add(mul)(-bin)`	Unrestricted (NIST-derived)	✓
Fashion-MNIST (Xiao et al., 2017)	`fashion`	MIT	✓
CIFAR-based (Krizhevsky et al., 2009)	`cifar10(100)`	No explicit license	✗
BDD-OIA (Xu et al., 2020)	`bdd-oia(-2)`	See (Bortolotti et al., 2024)	✓
ROAD-R (Giunchiglia et al., 2023)	`road-r`	CC BY-NC-SA 4.0	✓
SUSHI3 (Kamishima, 2003)	`sushi`	Research use permitted; redistribution forbidden.	✗
CEBaB (Abraham et al., 2022)	`cebab`	CC BY 4.0	✓
ChestX-Rays (Cohen et al., 2022)	`chx`	Unrestricted (NIH), attribution requested	✓
Warcraft tiles (Pogančić et al., 2020)	`warcraft`	MIT	✓
Kandinsky/CLEVR (Bortolotti et al., 2024)	`kand-logic(-2),cle4evr`	BSD-3-Clause	✓

`SUSHI3`. This dataset is not redistributed with the benchmark and must be downloaded directly from Kamishima's archive (https://www.kamishima.net/sushi/). Its license explicitly states that "You must not redistribute." While research use is permitted, redistribution of the dataset is expressly prohibited.

`CIFAR-10 / CIFAR-100`. These datasets cannot be redistributed with a new benchmark and can be instead downloaded on demand via `torchvision` (https://www.cs.toronto.edu/~kriz/cifar.html). We follow this approach because the datasets do not provide an explicit license or redistribution grant from their creators.

D.3.2 CONSTRAINT DESCRIPTIONS (DETAILED VS. NON-DETAILED)

For each task, we give the `{problem}` text in both phrasings. Here we list all the prompts used for `auto-nesy-bench`.

`mn-add-bin`.

```
[DETAILED]
We want to formalize the single-digit addition task (full range:
↪  digits 0-9).

Logical constraints:
- The first digit is binary encoded (value between 0 and 9).
- The second digit is binary encoded (value between 0 and 9).
- The sum is binary encoded (value between 0 and 18).
- The sum is equal to the first digit plus the second digit in
↪  binary.

[NOT DETAILED]
We want to formalize the single-digit addition task (full range:
↪  digits 0-9).

Logical constraints:
- The sum is equal to the first digit plus the second digit in
↪  binary.
```

`mn-add`.

```
[DETAILED]
We want to formalize the single-digit addition task (full range:
↪  digits 0-9).

Logical constraints:
```

```
- The first digit must have exactly one value from 0 to 9.
- The second digit must have exactly one value from 0 to 9.
- The sum must have exactly one value from 0 to 18.
- The sum must equal the first digit plus the second digit.

[NOT DETAILED]
We want to formalize the single-digit addition task (full range:
↪  digits 0-9).

Logical constraints:
- The sum must equal the first digit plus the second digit.
```

mn-mul-bin.

```
[DETAILED]
We want to formalize the single-digit multiplication task (full
↪  range: digits 0-9).

Logical constraints:
- The first digit is binary encoded (value between 0 and 9).
- The second digit is binary encoded (value between 0 and 9).
- The product is binary encoded (value between 0 and 81).
- The product is equal to the first digit multiplied by the second
↪  digit in binary.

[NOT DETAILED]
We want to formalize the single-digit multiplication task (full
↪  range: digits 0-9).

Logical constraints:
- The product is equal to the first digit multiplied by the second
↪  digit in binary.
```

mn-mul.

```
[DETAILED]
We want to formalize the single-digit multiplication task (full
↪  range: digits 0-9).

Logical constraints:
- The first digit must have exactly one value from 0 to 9.
- The second digit must have exactly one value from 0 to 9.
- The product must have exactly one value from 0 to 81.
- The product is equal to the first digit multiplied by the second
↪  digit.

[NOT DETAILED]
We want to formalize the single-digit multiplication task (full
↪  range: digits 0-9).

Logical constraints:
- The product is equal to the first digit multiplied by the second
↪  digit.
```

chx.

```
[DETAILED]
We want to formalize a clinical severity-scoring rule for chest
↪  X-ray findings.

Logical constraints:
```

```
The severity code is determined by the number of findings that are
↪  present:
- no findings correspond to healthy
- one finding to green_code
- two findings to yellow_code
- three findings to orange_code
- four findings to red_code.

Exactly one of the emergency codes must be true.

[NOT DETAILED]
We want to formalize a clinical severity-scoring rule for chest
↪  X-ray findings.

Logical constraints:
- The severity code is determined by the number of findings that
↪  are present
  (from 0 - healthy, to 4 - red code).
- Exactly one of the emergency codes must be true.
```

fashion.

```
[DETAILED]
We want to formalize a constraint on a dress classification task:
↪  every clothing image
belongs to exactly one of the classes.

Logical constraints:
- The item must belong to exactly one class.
- No two classes can be assigned to the same item.

[NOT DETAILED]
We want to formalize a constraint on a dress classification task:
↪  every clothing image
belongs to exactly one of the classes.

Logical constraints:
- The item must belong to exactly one class.
```

cifar10.

```
[DETAILED]
We want to formalize an object classification task.

Logical constraints:
- The object is an airplane if it has wings and is metallic, but
↪  has no wheels and is not an
  animal, hairy, horned, or snouted.
- The object is either a car or truck if it has wheels and is
↪  metallic, but has no wings and
  is not an animal, hairy, horned, or snouted.
- The object is a bird if it is an animal with wings, but has no
↪  wheels and is not metallic,
  hairy, horned, or snouted.
- The object is a frog if it is an animal with no wheels, wings,
↪  metallic surface, hair,
  horns, or snout.
- The object is a deer if it is an animal that is hairy and has
↪  horns and a snout, but has no
  wheels, wings, or metallic surface.
```

```
- The object is a cat if it is an animal that is hairy, but has no
↪  wheels, wings, metallic
  surface, horns, or snout.
- The object is either a dog or horse if it is an animal that is
↪  hairy and has a snout, but
  has no wheels, wings, metallic surface, or horns.
- The object is a ship if it is metallic, but has no wheels or
↪  wings and is not an animal,
  hairy, horned, or snouted.
- Exactly one of the object class must be true.

If the object does not exactly match any of these descriptions, it
↪  cannot be classified as
any of the above.

[NOT DETAILED]
We want to formalize an object classification task.

Logical constraints:
- An airplane has wings and is metallic.
- A car or truck has wheels and is metallic.
- A bird is an animal with wings.
- A frog is an animal with none of the listed physical features.
- A deer is an animal with hair, horns, and a snout.
- A cat is an animal with hair.
- A dog or horse is an animal with hair and a snout.
- A ship is metallic but has no animal features.
- Exactly one class must be true.

If the object does not exactly match any of these descriptions, it
↪  cannot be classified as
any of the above.
```

cifar100.

```
[DETAILED]
We want to formalize a constraint on a general object
↪  classification task: every image
belongs to exactly one of the classes.

Logical constraints:
- The object belongs to exactly one class.
- No two classes can be assigned to the same object.

[NOT DETAILED]
We want to formalize a constraint on a general object
↪  classification task: every image
belongs to exactly one of the classes.

Logical constraints:
- The object belongs to exactly one class.
```

sushi.

```
[DETAILED]
We want to formalize the validity of a ranking of sushi types
↪  across preference ranks.

Logical constraints:
- Each sushi item must be assigned to exactly one preference rank.
- Every rank must be assigned to exactly one sushi item.
```

```
- Each sushi item cannot be assigned to more than one rank.
- No two sushi items can share the same rank.

[NOT DETAILED]
We want to formalize the validity of a ranking of sushi types
↪  across preference ranks.

Logical constraints:
- Encode a valid permutation matrix representing a complete
↪  ranking of the four sushi items.
```

sudoku.

```
[DETAILED]
We want to formalize a grid validity task (4x4 grid).

Logical constraints:
- Each cell must contain exactly one digit.
- Every row must contain each digit exactly once.
- Every column must contain each digit exactly once.
- Every 2x2 box must contain each digit exactly once.
- The four 2x2 boxes are:
    - Rows 0-1 and Columns 0-1
    - Rows 0-1 and Columns 2-3
    - Rows 2-3 and Columns 0-1
    - Rows 2-3 and Columns 2-3
- The constraints apply simultaneously to all 16 cells of the grid

[NOT DETAILED]
We want to formalize a grid validity task (4x4 grid).

Logical constraints:
- Each cell must contain exactly one digit.
- Each digit 1-4 appears exactly once in each row, each column,
↪  and every 2x2 box.
```

warcraft.

```
[DETAILED]
We want to formalize a grid-pathfinding validity task on a 4x4
↪  grid.

Logical constraints:
- The selected variables represent the edges belonging to a path
↪  on the 4x4 grid.
- The path must start at the top-left cell (0,0).
- Exactly one of the two edges incident to the start cell (right
↪  or down) must be selected.
- The path must end at the bottom-right cell (3,3).
- Exactly one of the two edges incident to the end cell (left or
↪  up) must be selected.
- Every other cell in the grid must have either:
    - no incident selected edges (the path does not pass through
    ↪  the cell), or
    - exactly two incident selected edges (one edge enters the
    ↪  cell and one edge leaves it).
- No intermediate cell may have exactly one or more than two
↪  incident selected edges.
- The selected edges must form a continuous path connecting the
↪  start cell to the end cell.
- The selected edges must not contain any closed cycle.
```

```
- There must not be any disconnected loop or additional cycle
 ↪  elsewhere in the grid.
- The selected edges must form exactly one simple path from cell
 ↪  (0,0) to cell (3,3).

[NOT DETAILED]
We want to formalize a grid-pathfinding validity task on a 4x4
 ↪  grid.

Logical constraints:
- The selected edges should form one simple path on the 4x4 grid
 ↪  starting from cell (0,0)
  to cell (3,3).
- No loops, cycles, or disconnected paths are allowed.
```

kand-logic.

```
[DETAILED]
We want to formalize a pattern-recognition task: whether two
 ↪  images share a common
shape/color pattern.

Logical constraints:
- Each primitive has exactly one shape and one color.
- The two images must share either the same color pattern or the
 ↪  same shape pattern:
    - a color pattern is defined by whether the three primitives
     ↪  in an image all have the
      same color, exactly two share the same color, or all three
       ↪  have different colors.
    - a shape pattern is defined by whether the three primitives
     ↪  all have the same shape,
      exactly two share the same shape, or all three have
       ↪  different shapes.

[NOT DETAILED]
We want to formalize a pattern-recognition task: whether two
 ↪  images share a common
shape/color pattern.

Logical constraints:
- The two images must share a pattern. This happens if the same
 ↪  color/shape pattern applies
  to all three objects, exactly two objects, or none in both
   ↪  images.
```

kand-logic-2.

```
[DETAILED]
We want to formalize a pattern-recognition task: whether two
 ↪  images share a common
shape/color pattern.

Logical constraints:
- Each primitive has exactly one shape and one color.
- The two images must share either the same color pattern or the
 ↪  same shape pattern:
    - a color pattern is defined by whether the two primitives in
     ↪  an image have the same
      color or different colors.
```

```
    - a shape pattern is defined by whether the two primitives
     ↪  have the same shape or
      different shapes.

[NOT DETAILED]
We want to formalize a pattern-recognition task: whether two
 ↪  images share a common
shape/color pattern.

Logical constraints:
- The two images must share a pattern. This happens if the same
 ↪  color/shape pattern applies
  to both objects or to neither in both images.
```

bdd-oia.

```
[DETAILED]
We want to formalize a decision task: choosing driving actions
 ↪  from a set of binary
road-condition variables.

Domain notes (background only, do NOT encode these as CNF clauses;
 ↪  the concept variables
above are free inputs, not constrained against each other):
  - red_light and green_light are not physically both true at
   ↪  once.
  - road_clear is true exactly when none of
   ↪  car/person/rider/other_obstacle hold.

Logical constraints:
- The car can move forward if and only if there is a green light,
 ↪  it can follow the vehicle
  ahead, or the road is clear, provided that there is no red light
   ↪  and nothing is blocking
  the way, such as a traffic sign, a vehicle blocking the passage,
   ↪  a pedestrian, a cyclist,
  or any other obstacle.
- The car must stop if and only if there is a red light indicating
 ↪  that it must stop, a
  traffic sign requiring it to stop, or something is blocking the
   ↪  way, such as a vehicle
  blocking the passage, a pedestrian, a cyclist, or any other
   ↪  obstacle.
- The car can turn left if and only if there is a left lane
 ↪  available, the left traffic light
  is green, or it can follow another vehicle turning left,
   ↪  provided that there is no
  indication that no left lane is available, no obstacle is
   ↪  blocking the left turn, and there
  is no solid line preventing the maneuver.
- The car can turn right if and only if there is a right lane
 ↪  available, the right traffic
  light is green, or it can follow another vehicle turning right,
   ↪  provided that there is no
  indication that no right lane is available, no obstacle is
   ↪  blocking the right turn, and
  there is no solid line preventing the maneuver.

[NOT DETAILED]
```

```
We want to formalize a decision task: choosing driving actions
 ↪  from a set of binary
road-condition variables.

Logical constraints:
- The car can move forward if and only if it is allowed to proceed
 ↪  and the road ahead is
   clear.
- The car must stop if and only if a traffic signal, sign, or
 ↪  obstacle requires it to stop.
- The car can turn left if and only if the maneuver is permitted
 ↪  and the left lane is clear.
- The car can turn right if and only if the maneuver is permitted
 ↪  and the right lane is
   clear.

Note: red_light and green_light are not physically both true at
 ↪  once, and road_clear is true
exactly when none of car/person/rider/other_obstacle hold, but
 ↪  these are background facts
about how the inputs behave in practice, not additional rules to
 ↪  encode.
```

bdd-oia-2.

```
[DETAILED]
We want to formalize a decision task: choosing driving actions
 ↪  from a set of binary
road-condition variables.

Domain notes (background only, do NOT encode these as CNF clauses;
 ↪  the concept variables
above are free inputs, not constrained against each other):
   - red_light and green_light are not physically both true at
    ↪  once.
   - road_clear is true exactly when none of
    ↪  car/person/rider/other_obstacle hold.

Logical constraints:
- The car can move forward if and only if there is a green light,
 ↪  it can follow the vehicle
   ahead, or the road is clear, provided that there is no red light
    ↪  and nothing is blocking
   the way, such as a traffic sign, a vehicle blocking the passage,
    ↪  a pedestrian, a cyclist,
   or any other obstacle.
- The car must stop if and only if there is a red light indicating
 ↪  that it must stop, a
   traffic sign requiring it to stop, or something is blocking the
    ↪  way, such as a vehicle
   blocking the passage, a pedestrian, a cyclist, or any other
    ↪  obstacle.

[NOT DETAILED]
We want to formalize a decision task: choosing driving actions
 ↪  from a set of binary
road-condition variables.

Logical constraints:
```

```
- The car can move forward if and only if it is allowed to proceed
 ↪  and the road ahead is
  clear.
- The car must stop if and only if a traffic signal, sign, or
 ↪  obstacle requires it to stop.

Note: red_light and green_light are not physically both true at
 ↪  once, and road_clear is true
exactly when none of car/person/rider/other_obstacle hold, but
 ↪  these are background facts
about how the inputs behave in practice, not additional rules to
 ↪  encode.
```

cle4evr.

```
[DETAILED]
We want to formalize a co-occurrence matching task between two
 ↪  objects in a scene.

Logical constraints:
- Both objects must have the same shape and the same color.
- Each object must have exactly one shape and one color.

[NOT DETAILED]
We want to formalize a co-occurrence matching task between two
 ↪  objects in a scene.

Logical constraints:
- Both objects must have the same shape and the same color.
```

cebab.

```
[DETAILED]
We want to formalize an overall-sentiment aggregation rule for
 ↪  restaurant reviews.

Logical constraints:
- Exactly one output label must be true.
- The label is positive if and only if positive sentiment is the
 ↪  most common sentiment
  across the five sentiments, including cases where it ties with
   ↪  neutral or unknown, but not
  with negative.
- The label is negative if and only if negative sentiment is the
 ↪  most common sentiment
  across the five sentiments, including cases where it ties with
   ↪  neutral or unknown, but not
  with positive.
- The label is neutral if and only if neutral sentiment is
 ↪  strictly more common than both
  positive and negative sentiment, including cases where it ties
   ↪  with unknown.
- The label is unknown if and only if unknown is strictly more
 ↪  common than positive,
  negative, and neutral sentiment.
- The label is conflict if and only if positive and negative
 ↪  sentiment are tied as the most
  common sentiments.

[NOT DETAILED]
```

```
We want to formalize an overall-sentiment aggregation rule for
↪  restaurant reviews.

Logical constraints:
- The output label is determined by a majority vote over the five
↪  input sentiments:
  positive, negative, neutral, and unknown.
- In case of a tie, use the priority order: Positive > Negative >
↪  Neutral > Unknown.
- If positive and negative are tied for the highest count, the
↪  output is Conflict.
```

road-r.

```
[DETAILED]
We want to formalize a consistency-checking task from autonomous
↪  driving. Picture a single
frame of dash-cam footage from a self-driving car, with a box
↪  drawn around one detected road
user or traffic light. An annotator has labeled that one detection
↪  with a whole set of tags
at once, i.e., what kind of agent it is, what it's doing, where it
↪  is, and, if it's a
traffic light, what color it's showing.

Logical constraints:
- Exactly one agent class must be true: pedestrian, car, cyclist,
↪  motorbike, medium vehicle,
  large vehicle, bus, emergency vehicle, traffic light, or other
  ↪  traffic light.
- At least one location must be true among vehicle lane, outgoing
↪  lane, outgoing cycle lane,
  incoming lane, incoming cycle lane, pavement, left pavement,
  ↪  right pavement, junction,
  crossing location, bus stop, or parking area. Traffic-light
  ↪  locations are also permitted
  by the stated requirement.
- If the detection is pushing an object, it must be a pedestrian.
- A pedestrian must have at least one of: pushing an object,
↪  moving away, moving towards,
  moving, stopped, turning left, turning right, waiting to cross,
  ↪  crossing from the left,
  crossing from the right, or crossing.
- Crossing, crossing from the left, crossing from the right,
↪  moving, moving towards, moving
  away, or stopped requires the detection to be a road user:
  ↪  pedestrian, car, cyclist,
  motorbike, medium vehicle, large vehicle, bus, or emergency
  ↪  vehicle.
- Waiting to cross requires the detection to be a pedestrian or
↪  cyclist.
- An emergency vehicle, bus, large vehicle, motorbike, or car must
↪  have at least one of the
  specified driving actions: overtaking, moving away, moving
  ↪  towards, moving, braking,
  stopped, indicating left, indicating right, hazard lights on,
  ↪  turning left, turning right,
  crossing from the left, crossing from the right, or crossing.
- A cyclist must have at least one action from crossing, moving
↪  away, moving towards, moving,
```

```
  braking, stopped, indicating left, indicating right, turning
   ↪  left, turning right,
  overtaking, waiting to cross, crossing from the left, or
   ↪  crossing from the right.
- Hazard lights require the detection to be an emergency vehicle,
 ↪  car, medium vehicle, large
  vehicle, bus, or motorbike.
- Overtaking, indicating left, indicating right, or braking
 ↪  requires the detection to be a
  cyclist, medium vehicle, large vehicle, bus, motorbike,
   ↪  emergency vehicle, or car.
- Turning left or turning right requires the detection to be a
 ↪  car, cyclist, motorbike,
  medium vehicle, large vehicle, bus, or emergency vehicle.
- If crossing is true, the detection must be a pedestrian or
 ↪  cyclist.
- The following agent classes are mutually exclusive:
    - car with cyclist, motorbike, medium vehicle, large vehicle,
     ↪  bus, emergency vehicle,
      traffic light, other traffic light, or pedestrian
    - cyclist with motorbike, medium vehicle, large vehicle, bus,
     ↪  emergency vehicle, traffic
      light, other traffic light, or pedestrian
    - motorbike with medium vehicle, large vehicle, bus, emergency
     ↪  vehicle, traffic light,
      other traffic light, or pedestrian
    - medium vehicle with large vehicle, bus, emergency vehicle,
     ↪  traffic light, other traffic
      light, pedestrian
    - large vehicle with bus, emergency vehicle, traffic light,
     ↪  other traffic light,
      pedestrian
    - bus with emergency vehicle, traffic light, other traffic
     ↪  light, or pedestrian
    - emergency vehicle with traffic light, other traffic light,
     ↪  or pedestrian
    - traffic light with other traffic light
- Traffic-light colours are mutually exclusive: red, amber, and
 ↪  green cannot occur together.
- Red, amber, or green requires the detection to be a traffic
 ↪  light or another traffic light.
- Traffic-light colours are incompatible with the road-user
 ↪  classes and actions specified in
  the requirements. In particular, red/amber/green cannot occur
   ↪  together with pedestrian,
  car, cyclist, motorbike, medium vehicle, large vehicle, bus, or
   ↪  emergency vehicle, and
  cannot occur with the prohibited movement/action combinations
   ↪  listed in the requirements.
- Traffic lights and other traffic lights cannot have the
 ↪  prohibited actions such as moving
  away, moving towards, moving, braking, stopped, indicating
   ↪  left/right, hazard lights,
  turning left/right, overtaking, waiting to cross, crossing from
   ↪  either side, crossing, or
  pushing an object.
- Traffic lights and other traffic lights are also incompatible
 ↪  with pedestrian.
- The following action pairs cannot occur together:
    - moving away and moving
```

```
    - moving towards and moving
    - moving and stopped
    - indicating left and indicating right
    - braking and waiting to cross
    - braking and pushing an object
    - crossing and braking
    - waiting to cross and indicating left
    - waiting to cross and indicating right
    - waiting to cross and overtaking
    - waiting to cross and crossing from the left
    - waiting to cross and crossing from the right
    - waiting to cross and crossing
    - pedestrian and overtaking
    - pedestrian and indicating left
    - pedestrian and indicating right
    - pedestrian and hazard lights
    - pedestrian and braking
    - pedestrian and pushing an object
- The following location pairs cannot occur together:
    - outgoing lane and outgoing cycle lane
    - vehicle lane and incoming cycle lane
    - outgoing lane and incoming cycle lane
    - incoming lane and outgoing cycle lane
    - outgoing lane and pavement
    - incoming cycle lane and outgoing cycle lane
    - outgoing lane and right pavement
    - incoming cycle lane and pavement
    - crossing location and outgoing lane
    - vehicle lane and parking area
    - bus stop and outgoing lane
    - outgoing lane and parking area
    - bus stop and outgoing cycle lane
    - parking area and outgoing cycle lane
    - crossing location and incoming cycle lane
    - left pavement and right pavement
    - incoming lane and parking area
    - incoming cycle lane and bus stop
    - incoming cycle lane and parking area
    - pavement and parking area
    - left pavement and parking area
    - right pavement and parking area
    - junction and parking area
    - crossing location and bus stop
    - crossing location and parking area
    - bus stop and parking area
- All of the above constraints apply simultaneously to the same
 ↪  detection.

[NOT DETAILED]
We want to formalize a consistency-checking task from autonomous
 ↪  driving. Picture a single
frame of dash-cam footage from a self-driving car, with a box
 ↪  drawn around one detected road
user or traffic light. An annotator has labeled that one detection
 ↪  with a whole set of tags
at once, i.e., what kind of agent it is, what it's doing, where it
 ↪  is, and, if it's a
traffic light, what color it's showing.

Logical constraints:
```

```
- Exactly one agent class must be true.
- At least one location must be true.
- Each agent class can only have its valid actions.
- Each action can only apply to valid agent classes.
- Each agent class can only occur in valid locations.
- Traffic lights can only have a valid traffic-light colour and no
 ↪  movement actions.
- Red, amber, and green are mutually exclusive.
- Pushing an object is only allowed for pedestrians.
- Crossing and waiting-to-cross are only allowed for valid road
 ↪  users.
- Invalid agent-class combinations are not allowed.
- Invalid agent-action combinations are not allowed.
- Invalid agent-location combinations are not allowed.
- Invalid action-action combinations are not allowed.
- Invalid location-location combinations are not allowed.
- All constraints apply to the same detection.
```

D.3.3 VARIABLE LISTS

The block below is substituted verbatim for `{variables}`. Each line is "`<index>: <name>: <description>`".

`chx` (9).

```
  1: fracture: a bone fracture is visible
  2: pneumothorax: a pneumothorax (collapsed lung) is visible
  3: airspace_opacity: airspace opacity (e.g. consolidation or
   ↪  infiltrate) is visible
  4: nodule_mass: a nodule or mass is visible
  5: healthy: no findings are present
  6: green_code: green emergency code
  7: yellow_code: yellow emergency code
  8: orange_code: orange emergency code
  9: red_code: red emergency code
```

`fashion` (10).

```
  1: class_0_tshirt_top: the clothing item is a t-shirt
  2: class_1_trouser: the clothing item is a pair of trousers
  3: class_2_pullover: the clothing item is a pullover
  4: class_3_dress: the clothing item is a dress
  5: class_4_coat: the clothing item is a coat
  6: class_5_sandal: the footwear is a sandal
  7: class_6_shirt: the clothing item is a shirt
  8: class_7_sneaker: the footwear is a sneaker
  9: class_8_bag: the item is a bag
 10: class_9_ankle_boot: the footwear is an ankle boot
```

`bdd-oia-2` (11).

```
  1: green_light: the traffic light ahead is green
  2: follow: a lead vehicle is present and can be followed
  3: road_clear: the road ahead is clear (no obstacles)
  4: red_light: the traffic light ahead is red
  5: traffic_sign: a stop/yield sign is visible
  6: car: there is a car blocking the way
  7: person: there is a pedestrian crossing the street
  8: rider: there is a rider blocking the way
  9: other_obstacle: there is an obstacle blocking the way
 10: move_forward: proceed straight ahead
 11: stop_action: come to a stop
```

cle4evr (12).

```
 1: obj1_shape_cube: the first object is a cube
 2: obj1_shape_sphere: the first object is a sphere
 3: obj1_shape_cylinder: the first object is a cylinder
 4: obj1_color_red: the first object is red
 5: obj1_color_blue: the first object is blue
 6: obj1_color_green: the first object is green
 7: obj2_shape_cube: the second object is a cube
 8: obj2_shape_sphere: the second object is a sphere
 9: obj2_shape_cylinder: the second object is a cylinder
10: obj2_color_red: the second object is red
11: obj2_color_blue: the second object is blue
12: obj2_color_green: the second object is green
```

cifar10 (15).

```
 1: whl: the object has wheels
 2: met: the object is metallic
 3: wng: the object has wings
 4: ani: the object is an animal
 5: hai: the object is hairy
 6: hrn: the object has horns
 7: snt: the object has a snout
 8: label_airplane: the object is an airplane
 9: label_car_or_truck: the object is either a car or a truck
10: label_bird: the object is a bird
11: label_frog: the object is a frog
12: label_deer: the object is a deer
13: label_cat: the object is a cat
14: label_dog_or_horse: the object is either a dog or a horse
15: label_ship: the object is a ship
```

mn-mul-bin (15).

```
 1: d1_bit0: the least significant bit of the first digit
 2: d1_bit1: the second bit of the first digit
 3: d1_bit2: the third bit of the first digit
 4: d1_bit3: the most significant bit of the first digit
 5: d2_bit0: the least significant bit of the second digit
 6: d2_bit1: the second bit of the second digit
 7: d2_bit2: the third bit of the second digit
 8: d2_bit3: the most significant bit of the second digit
 9: prod_bit0: the least significant bit of the product
10: prod_bit1: the second bit of the product
11: prod_bit2: the third bit of the product
12: prod_bit3: the fourth bit of the product
13: prod_bit4: the fifth bit of the product
14: prod_bit5: the sixth bit of the product
15: prod_bit6: the most significant bit of the product
```

sushi (16).

```
 1..4:   item_0_rank_{0..3}: sushi item 0 is assigned to rank
 ↪ {0..3}
 5..8:   item_1_rank_{0..3}: sushi item 1 is assigned to rank
 ↪ {0..3}
 9..12:  item_2_rank_{0..3}: sushi item 2 is assigned to rank
 ↪ {0..3}
13..16:  item_3_rank_{0..3}: sushi item 3 is assigned to rank
↪ {0..3}
```

```
(rendered as 16 explicit lines "item_i_rank_j: sushi item i is
↪  assigned to rank j")
```

mn-add-bin (13).

```
  1: d1_bit0: the least significant bit of the first digit
  2: d1_bit1: the second bit of the first digit
  3: d1_bit2: the third bit of the first digit
  4: d1_bit3: the most significant bit of the first digit
  5: d2_bit0: the least significant bit of the second digit
  6: d2_bit1: the second bit of the second digit
  7: d2_bit2: the third bit of the second digit
  8: d2_bit3: the most significant bit of the second digit
  9: sum_bit0: the least significant bit of the sum
 10: sum_bit1: the second bit of the sum
 11: sum_bit2: the third bit of the sum
 12: sum_bit3: the fourth bit of the sum
 13: sum_bit4: the most significant bit of the sum
```

kand-logic-2 (24).

```
For im in {1,2}, primitive p in {1,2}:
  im{im}_p{p}_shape_square   : primitive p in image im is a square
  im{im}_p{p}_shape_triangle : primitive p in image im is a
  ↪  triangle
  im{im}_p{p}_shape_circle   : primitive p in image im is a circle
  im{im}_p{p}_color_red      : primitive p in image im is red
  im{im}_p{p}_color_blue     : primitive p in image im is blue
  im{im}_p{p}_color_yellow   : primitive p in image im is yellow
Indices 1..24 in the order (im1 p1, im1 p2, im2 p1, im2 p2), 6
↪  bits each.
```

warcraft (24).

```
  1..12:  edge_right_r_c (r=0..3, c=0..2):
          the edge between cells (r,c) and (r,c+1) is part of the
          ↪  path
 13..24:  edge_down_r_c  (r=0..2, c=0..3):
          the edge between cells (r,c) and (r+1,c) is part of the
          ↪  path
```

bdd-oia (25).

```
  1: green_light: the traffic light ahead is green
  2: follow: a lead vehicle is present and can be followed
  3: road_clear: the road ahead is clear (no obstacles)
  4: red_light: the traffic light ahead is red
  5: traffic_sign: a stop/yield sign is visible
  6: car: there is a car blocking the way
  7: person: there is a pedestrian crossing the street
  8: rider: there is a rider blocking the way
  9: other_obstacle: there is an obstacle blocking the way
 10: left_lane: there is a left lane enabling a left turn
 11: left_green_light: there is a green light enabling a left turn
 12: left_follow: a lead vehicle on the left lane is present and
 ↪  can be followed
 13: no_left_lane: there is no left lane
 14: left_obstacle: there is an obstacle blocking the left lane
 15: left_solid_line: there is a solid line preventing a left turn
 16: right_lane: there is a right lane enabling a left turn
 17: right_green_light: there is a green light enabling a right
 ↪  turn
```

```
18: right_follow: a lead vehicle on the right lane is present and
 ↪  can be followed
19: no_right_lane: there is no right lane
20: right_obstacle: there is an obstacle blocking the right lane
21: right_solid_line: there is a solid line preventing a right
 ↪  turn
22: move_forward: proceed straight ahead
23: stop_action: come to a stop
24: turn_left: turn left
25: turn_right: turn right
```

cebab (25).

```
 1: food_aspect_majority_positive: positive sentiment toward the
 ↪  food
 2: food_aspect_majority_negative: negative sentiment toward the
 ↪  food
 3: food_aspect_majority_neutral: neutral sentiment toward the
 ↪  food
 4: food_aspect_majority_unknown: unknown sentiment toward the
 ↪  food
 5: service_aspect_majority_positive: positive sentiment toward
 ↪  the service
 6: service_aspect_majority_negative: negative sentiment toward
 ↪  the service
 7: service_aspect_majority_neutral: neutral sentiment toward the
 ↪  service
 8: service_aspect_majority_unknown: unknown sentiment toward the
 ↪  service
 9: noise_aspect_majority_positive: positive sentiment toward the
 ↪  noise
10: noise_aspect_majority_negative: negative sentiment toward the
 ↪  noise
11: noise_aspect_majority_neutral: neutral sentiment toward the
 ↪  noise
12: noise_aspect_majority_unknown: unknown sentiment toward the
 ↪  noise
13: ambiance_aspect_majority_positive: positive sentiment toward
 ↪  the ambiance
14: ambiance_aspect_majority_negative: negative sentiment toward
 ↪  the ambiance
15: ambiance_aspect_majority_neutral: neutral sentiment toward
 ↪  the ambiance
16: ambiance_aspect_majority_unknown: unknown sentiment toward
 ↪  the ambiance
17: review_majority_positive: positive overall review sentiment
18: review_majority_negative: negative overall review sentiment
19: review_majority_neutral: neutral overall review sentiment
20: review_majority_unknown: unknown overall review sentiment
21: label_positive: final assessment of the restaurant is
 ↪  positive
22: label_negative: final assessment of the restaurant is
 ↪  negative
23: label_neutral: final assessment of the restaurant is neutral
24: label_unknown: final assessment of the restaurant is unknown
25: label_conflict: final assessment of the restaurant is
 ↪  conflicting
```

kand-logic (36).

Same schema as kand-logic-2 but with 3 primitives per image:

```
For im in {1,2}, primitive p in {1,2,3}:
  im{im}_p{p}_shape_{square,triangle,circle},
   ↪  im{im}_p{p}_color_{red,blue,yellow}
Indices 1..36 in the order (im1 p1..p3, im2 p1..p3), 6 bits each.
```

mn-add (39).

```
  1..10:  d1_{0..9}: the first digit is {0..9}
 11..20:  d2_{0..9}: the second digit is {0..9}
 21..39:  s_{0..18}:  the sum is {0..18}
```

road-r (41).

```
Agent (one-hot):
  1: ped     the detection is a pedestrian
  2: car     the detection is a car
  3: cyc     the detection is a cyclist
  4: mobike  the detection is a motorbike
  5: med_veh the detection is a medium vehicle
  6: lar_veh the detection is a large vehicle
  7: bus     the detection is a bus
  8: em_veh  the detection is an emergency vehicle
  9: tl      the detection is a traffic light
 10: oth_tl  the detection is another traffic light
Traffic-light colour (at most one):
 11: red     the traffic light is red
 12: amber   the traffic light is amber
 13: green   the traffic light is green
Action:
 14: mov_away     the detected agent is moving away
 15: mov_tow      the detected agent is moving towards the ego
  ↪  vehicle
 16: mov          the detected agent is moving
 17: brake        the detected agent is braking
 18: stop         the detected agent is stopped
 19: incat_left   the agent is indicating left
 20: incat_rht    the agent is indicating right
 21: haz_lit      the agent has its hazard lights on
 22: tur_lft      the agent is turning left
 23: tur_rht      the agent is turning right
 24: ovtak        the agent is overtaking
 25: wait_2x      the agent is waiting to cross
 26: xing_fm_lft  the agent is crossing from the left
 27: xing_fm_rht  the agent is crossing from the right
 28: xing         the agent is crossing
 29: push_obj     the agent is pushing an object
Location:
 30: veh_lane        the detection is in a vehicle lane
 31: outgo_lane      the detection is in an outgoing lane
 32: outgo_cyc_lane  the detection is in an outgoing cycle lane
 33: incom_lane      the detection is in an incoming lane
 34: incom_cyc_lane  the detection is in an incoming cycle lane
 35: pav             the detection is on the pavement
 36: lft_pav         the detection is on the left pavement
 37: rht_pav         the detection is on the right pavement
 38: jun             the detection is at a junction
 39: xing_loc        the detection is at a crossing location
 40: bus_stop        the detection is at a bus stop
 41: parking         the detection is in a parking area
```

sudoku (64).

```
  index i*16 + j*4 + v  ->  cell_i_j_val_v (i,j in 0..3, v in
  ↪  1..4):
  "cell at row i, column j contains digit v"
```

`cifar100` (100).

```
  1..100: class_{0..99}: the image belongs to class {0..99}
```

`mn-mul` (102).

```
  1..10:  d1_{0..9}: the first digit is {0..9}
 11..20:  d2_{0..9}: the second digit is {0..9}
 21..102: p_{0..81}:  the product is {0..81}
```

E LARGE LANGUAGE MODELS

Backends span eight open-weight model families Qwen(-Coder) (Yang et al., 2025), Gemma (Team et al., 2025; 2026), Mistral-Nemo (Jiang et al., 2023), DeepSeek-R1-Distill (Guo et al., 2025; Liu et al., 2024), Olmo (Groeneveld et al., 2024), Phi (Abdin et al., 2025), GPT-OSS (OpenAI, 2025)) at parameter counts from 8.2B to 117B. Table 5 lists every checkpoint. We load each checkpoint *unquantized* in bfloat16. `gpt-oss-20b` and `gpt-oss-120b` are the only exceptions to this policy: both ship with their own native low-bit format (MXFP4 under `vLLM`) and are loaded as-is. We choose instruction-tuned models when possible.

F PROMPTS

Each (dataset, output format, prompting mode) combination has its own template (`fonesy/prompts/<dataset>_<format>[_cot].txt`); modes are ZS and ZS-CoT. We show the representative ZS `DIMACS` template for each of the three datasets below.

`auto-nesy-bench`.

```
You are an expert in SAT solving and constraint modeling.

Your task is to convert the following logical problem into a CNF
↪  formula in DIMACS format.

The Boolean variables are numbered as follows:
{variables}

DIMACS format rules:
- First line: p cnf <num_vars> <num_clauses>
- Each subsequent line is one clause: space-separated integers
↪  ending with 0
- Positive integer i = variable i is True; negative integer -i =
↪  variable i is False
- Variable numbers MUST match exactly the mapping above

Problem:
{problem}

Strict output rules:
- Start your answer with <cnf> and end with </cnf>
- Every clause line MUST end with 0
- NO markdown, NO code fences, NO backticks, NO explanation
↪  outside the tags
```

`satbench`.

```
{scenario}
{variable_mapping}

Use only the mapped variables; do not introduce auxiliary
↪ variables.

Logical constraints (encode EXACTLY these as CNF clauses, and
↪ nothing else):
{conditions}
```

dcpbench.

```
You are an expert in SAT solving and constraint modeling.

Your task is to convert the following logical problem into a CNF
↪ formula in DIMACS format.

This problem's declared variables are:
{variables}

DIMACS format rules:
- First line: p cnf <num_vars> <num_clauses>
- Each subsequent line is one clause: space-separated integers
↪ ending with 0
- Positive integer i = variable i is True; negative integer -i =
↪ variable i is False
- Each declared variable's numbered range above IS its one-hot
↪ expansion, in
  ascending domain-value order (e.g. "1-101: men: integer, domain
  ↪ [0, 100]"
  means variable 1 is true iff men=0, variable 2 iff men=1, ...,
  ↪ variable 101
  iff men=100); exactly one variable in each range must be true
- If you need additional Boolean variables for auxiliary
↪ sub-expressions,
  number them consecutively starting right after the highest
  ↪ number shown above

Problem:
{problem}

Strict output rules:
- Start your answer with <cnf> and end with </cnf>
- Every clause line MUST end with 0
- NO markdown, NO code fences, NO backticks, NO explanation
↪ outside the tags
```

The self-correction prompt (shared by all three datasets) supplies the original problem, the proposed CNF, and the detected error message, and requires the model to answer `[[OK]]` or `[[FIXED]]` followed by a corrected `<cnf>...</cnf>` block.

Other output formats. The templates described above correspond to the `DIMACS` variants. The remaining four formats, i.e., `PySAT`, `CPMpy`, `SymPy`, and `NAT`, retain the same role sentence, the `variables` and `problem` slots, and the `<cnf>...</cnf>` wrapper. They differ only in the format-specific instructions and in how the variable list is to be interpreted.

- `PySAT`: the model generates a Python program rather than raw clauses. The program imports `CNF` from `pysat.formula` and `sys`, appends clauses as lists of non-zero integers, and writes the resulting formula using `cnf.to_fp(sys.stdout)`. Variable indices must match the provided mapping. When using `CardEnc` or `PBEnc`, the model must specify

top_id=N or vpool=IDPool(start_from=N+1) to ensure that auxiliary variable IDs do not overlap with the mapped variables.

- CPMpy: the model generates a Python program that declares one cp.boolvar(name='<name>') for each listed variable. Variables are therefore identified by name rather than by integer index. The model combines these Boolean variables using &, |, and ~, and passes the resulting constraint list to the pre-injected _print_dimacs([...]) function, which must not be redefined. For dcpbench, variable names follow the convention <name>_eq<value>, with one Boolean variable for each value in the domain.
- SymPy: the model generates a Python program using Symbol('<name>') to represent variables and to_cnf(expr, simplify=False) to convert the resulting expression to CNF. The resulting formula is passed to the pre-injected _print_dimacs(cnf_expr) function. Variable naming follows the same convention as in CPMpy.
- **Natural-language operators** (nat): the model produces no code and does not use variable numbering. Variables are referred to by name, and the output must have the form (lit OR lit ...) AND (...). The operators AND, OR, and NOT are accepted in any letter case. XOR, NAND, NOR, XNOR, and the symbols &, |, and ~ are not permitted. Each clause must be enclosed in parentheses, and clauses must be joined by a top-level AND.

The output constraints are adapted accordingly. The requirement that every clause end with 0 applies only to the DIMACS formats, while the code-based formats require the generated program to print a valid DIMACS formula to standard output. In the ZS-CoT variants, the strict output constraints are replaced by an instruction to perform the reasoning inside <think></think> before emitting the final <cnf> block. Self-correction uses a dedicated template for each format while retaining the same [[OK]]/[[FIXED]] protocol.

G LLMs Hyperparameters and Model Selection

Each evaluation has a budget of 5 self-verification steps.

We do *not* tune any decoding hyperparameter. For every model, the sampling configuration, whether to sample at all, the temperature τ, the nucleus threshold ρ (top_p), κ (top_k), and any repetition penalty, is taken straight from the model HuggingFace model card / generation_config.json, i.e., the provider's own recommended inference settings. Almost all values in Table 10 come directly from that source; the few exceptions (e.g., repetition_penalty= 1.3 for the Gemma checkpoints, eager attention for the Qwen family) are documented mitigations for known degenerate-decoding or numerical issues.

The only decoding value we set ourselves is the generation-length cap. Because a target CNF can span thousands of clauses, we use 32,768 new tokens across all settings; this budget is identical for every model and every cluster. Reasoning-tuned checkpoints (qwen3-8b, qwen3-32b, the deepseek-r1-llama-70b, phi4-reasoning-plus, olmo3-32b-think, and the gpt-oss family) keep their internal "thinking" traces enabled; all other models follow their card greedy-vs.-sampling recommendation.

H SPL and SL Model Selection

All SPL and SL experiments use the Adam (Kingma & Ba, 2015) optimizer. Hyperparameters were selected through Bayesian optimization with Optuna (Akiba et al., 2019), using a TPE sampler (Watanabe, 2023) (all with seed 32) together with a median pruner that early-stops clearly bad trials from their intermediate per-epoch validation F1. Each study maximizes the best validation macro-F1 that a trial reaches over its own epochs. Studies ran 30 trials of 50 epochs each for mn-add-bin, mn-add, mn-mul-bin, mn-mul, sudoku, bdd-oia, bdd-oia-2, sushi, cle4evr and kand-logic; 20 trials of 30 epochs each for fashion, warcraft, cifar10, cifar100, chx and cebab; and 12 trials of 6 epochs each for road-r due to time constraints.

Table 10: **Decoding hyperparameters for the evaluated LLMs**. "sample" is `do_sample`; a dash means the parameter is unset (provider default); "think" marks models whose reasoning trace is always active. All values are the model-card / `generation_config.json` defaults unless noted.

MODEL	SAMPLE	τ	ρ	κ	REP. PEN.	THINK	PRECISION
`qwen3-32b`	✓	0.6	0.95	20	–	✓	bf16
`qwen3-8b`	✓	0.6	0.95	20	–	✓	bf16
`qwen3-coder-next`	✓	1.0	0.95	40	–	–	bf16
`mistral-nemo`	–	–	–	–	–	–	bf16
`phi4-reasoning-plus`	✓	0.8	0.95	50	–	✓	bf16
`gemma3-27b-it`	✓	0.7	0.95	64	1.3	–	bf16
`gemma4-31B-it`	✓	0.7	0.95	64	1.3	–	bf16
`olmo3-32b-think`	✓	0.6	0.95	–	–	✓	bf16
`deepseek-r1-llama-70b`	✓	0.6	0.95	–	–	✓	bf16
`gpt-oss-20b`	–	–	–	–	–	(native)	MXFP4
`gpt-oss-120b`	–	–	–	–	–	(native)	MXFP4

Only three quantities are tuned: the learning rate γ, searched log-uniformly over $[10^{-5}, 10^{-1}]$; the `SPL` head width h, over integers in $[32, 512]$; and the batch size ν, searched over a categorical grid that depends on the backbone and on the formula source. For the ground-truth formulas, ν spans over $\{16, 32, 64, 128, 256\}$ by default, restricted to $\{16, 32, 64\}$ for the `ResNet-18` (He et al., 2016) and `BERT` (Devlin et al., 2019) backbones for memory reasons. For the LLM-predicted formulas the ν grid is $\{16, 32, 64, 128, 256\}$ for all `Conv4`, `MLP`, `ResNet-18` and `BERT` tasks, with three exceptions: $\{8, 16, 32, 64\}$ for `chx` and `cebab` under `gpt-oss-120b`, and $\{16, 32, 64, 128\}$ for `cifar10`, `cifar100` and `chx` under `qwen3-8b`. For the `Faster-RCNN` (Ren et al., 2017) backbones (`cle4evr`, `kand-logic`, `kand-logic-2`) the grid is $\{2, 4, 8\}$ in all settings. For `road-r` with the `SlowFast` (Feichtenhofer et al., 2019) backbone the search space is narrower: $\gamma \in [10^{-5}, 3 \times 10^{-3}]$, $h \in [64, 512]$ and, for the ground-truth formula, $\nu \in \{32, 64, 96\}$; the `gpt-oss-120b` search used $\nu \in \{16, 32, 64\}$.

Final numbers are obtained by retraining the selected configuration for a *fixed* 50 epochs[9] with no early stopping; the reported test metrics use the *final-epoch* model. Every task is trained with the same 5 seeds for consistency, specifically $\{1011, 2021, 3031, 4041, 5051\}$; we report mean $\pm$ std.

The circuit has no trainable parameters of its own: it is compiled offline from each task CNF into a smooth structured-probabilistic layer whose sum layers are conditionally parameterized by a softmax gate function. A small `MLP` head $\texttt{Linear}(d_f, h) \to \texttt{ReLU} \to \texttt{Linear}(h, G)$ maps the backbone feature vector (dimension d_f) to the G gate-function weights with which is parametrized.

Even when two formulas are logically equivalent (REL is $=$), *they do not* necessarily yield the same circuit. The generated formula may use a different syntactic encoding of the same constraint; in particular, `CPMpy` and `SymPy` CNF lowering introduces auxiliary Tseitin variables. We existentially quantify these variables out during compilation, so the resulting circuit is defined over the same primary variables as the ground-truth circuit. However, the compiled structure, such as the vtree and SDD size, and consequently the number of sum-layer parameters produced by the network, depends on the specific CNF and *may therefore differ* from that of the ground-truth circuit.[10] As a result, optimal hyperparameters *do not* transfer across formulas, even when the formulas are logically equivalent. We therefore perform a separate hyperparameter search for each formula.

Below we list the configurations that performed best on each `auto-nesy-bench` dataset (γ / ν / h; optimizer `Adam`; backbone b) for ground-truth `SPL` formulas.

- `mn-add-bin` / `mn-add`: $\gamma = 6.4 \times 10^{-3}$ / 2.6×10^{-3}, $\nu = 32$ / 16, $h = 498$ / 188, $b =$ `Conv4`.
- `mn-mul-bin` / `mn-mul`: $\gamma = 1.6 \times 10^{-3}$ / 1.6×10^{-3}, $\nu = 256$ / 256, $h = 370$ / 370, $b =$ `Conv4`.
- `sudoku`: $\gamma = 2.9 \times 10^{-4}$, $\nu = 16$, $h = 455$, $b =$ `Conv4`.
- `fashion`: $\gamma = 4.7 \times 10^{-3}$, $\nu = 128$, $h = 52$, $b =$ `Conv4`.

[9] 10 for `road-r` due to time constraints.

[10] This affects only the `SPL` circuit and not the neural backbone.

- warcraft: $\gamma = 1.2\times10^{-2}$, $\nu = 64$, $h = 303$, $b =$ Conv4.
- cifar10: $\gamma = 1.0\times10^{-3}$, $\nu = 32$, $h = 223$, $b =$ ResNet-18.
- cifar100: $\gamma = 2.8\times10^{-4}$, $\nu = 32$, $h = 444$, $b =$ ResNet-18.
- chx: $\gamma = 9.5\times10^{-5}$, $\nu = 16$, $h = 462$, $b =$ ResNet-18.
- bdd-oia / bdd-oia-2: $\gamma = 1.4\times10^{-4}$ / 4.4×10^{-5}, $\nu = 16$ / 64, $h = 262$ / 441, $b =$ MLP (over 2048-d Faster-RCNN features).
- sushi: $\gamma = 1.6\times10^{-4}$, $\nu = 16$, $h = 254$, $b =$ MLP (over 36-d one-hot context).
- cle4evr: $\gamma = 8.4\times10^{-5}$, $\nu = 2$, $h = 512$, $b =$ Faster-RCNN.
- kand-logic / kand-logic-2: $\gamma = 2.5\times10^{-5}$, $\nu = 4$, $h = 327$, $b =$ Faster-RCNN.
- cebab: $\gamma = 1.0\times10^{-3}$, $\nu = 32$, $h = 223$, $b =$ BERT.
- road-r: $\gamma = 1.74\times10^{-4}$, $\nu = 64$, $h = 242$, $b =$ SlowFast-50.

Below, instad, we list the configurations that performed best on each auto-nesy-bench dataset (γ / ν / h; optimizer Adam; backbone b) for the LLM-predicted SPL formulas. We omit listing datasets for which the LLM did not produce a valid formula.

gpt-oss-120b.

- mn-add-bin: $\gamma = 2.6\times10^{-3}$, $\nu = 16$, $h = 188$, $b =$ Conv4.
- mn-add: $\gamma = 1.1\times10^{-2}$, $\nu = 64$, $h = 272$, $b =$ Conv4.
- mn-mul-bin: $\gamma = 2.3\times10^{-3}$, $\nu = 16$, $h = 190$, $b =$ Conv4.
- mn-mul: $\gamma = 7.8\times10^{-4}$, $\nu = 16$, $h = 162$, $b =$ Conv4.
- sudoku: $\gamma = 4.0\times10^{-4}$, $\nu = 16$, $h = 227$, $b =$ Conv4.
- fashion: $\gamma = 1.7\times10^{-3}$, $\nu = 128$, $h = 116$, $b =$ Conv4.
- warcraft: $\gamma = 3.4\times10^{-3}$, $\nu = 16$, $h = 185$, $b =$ Conv4.
- cifar10: $\gamma = 1.6\times10^{-3}$, $\nu = 256$, $h = 370$, $b =$ ResNet-18.
- cifar100: $\gamma = 2.8\times10^{-4}$, $\nu = 64$, $h = 190$, $b =$ ResNet-18.
- chx: $\gamma = 2.5\times10^{-5}$, $\nu = 8$, $h = 478$, $b =$ ResNet-18.
- bdd-oia: $\gamma = 9.0\times10^{-5}$, $\nu = 16$, $h = 356$, $b =$ MLP.
- bdd-oia-2: $\gamma = 4.5\times10^{-5}$, $\nu = 64$, $h = 34$, $b =$ MLP.
- sushi: $\gamma = 3.2\times10^{-3}$, $\nu = 256$, $h = 505$, $b =$ MLP.
- cle4evr: $\gamma = 9.0\times10^{-5}$, $\nu = 2$, $h = 356$, $b =$ Faster-RCNN.
- kand-logic: $\gamma = 1.4\times10^{-5}$, $\nu = 4$, $h = 289$, $b =$ Faster-RCNN.
- kand-logic-2: $\gamma = 3.1\times10^{-5}$, $\nu = 4$, $h = 375$, $b =$ Faster-RCNN.
- cebab: $\gamma = 1.8\times10^{-3}$, $\nu = 8$, $h = 141$, $b =$ BERT.
- road-r: $\gamma = 9.5\times10^{-5}$, $\nu = 16$, $h = 462$, $b =$ Slowfast-50.

olmo3-32b-think.

- mn-add: $\gamma = 3.0\times10^{-3}$, $\nu = 16$, $h = 59$, $b =$ Conv4.
- mn-mul-bin: $\gamma = 1.4\times10^{-3}$, $\nu = 16$, $h = 153$, $b =$ Conv4.
- mn-mul: $\gamma = 8.5\times10^{-4}$, $\nu = 16$, $h = 321$, $b =$ Conv4.
- fashion: $\gamma = 2.1\times10^{-3}$, $\nu = 64$, $h = 474$, $b =$ Conv4.
- warcraft: $\gamma = 1.2\times10^{-2}$, $\nu = 16$, $h = 185$, $b =$ Conv4.
- cifar10: $\gamma = 1.6\times10^{-3}$, $\nu = 256$, $h = 370$, $b =$ ResNet-18.
- cifar100: $\gamma = 9.5\times10^{-4}$, $\nu = 256$, $h = 60$, $b =$ ResNet-18.
- bdd-oia-2: $\gamma = 2.1\times10^{-5}$, $\nu = 16$, $h = 216$, $b =$ MLP.
- sushi: $\gamma = 1.1\times10^{-3}$, $\nu = 64$, $h = 164$, $b =$ MLP.

- `cle4evr`: $\gamma = 9.0\times10^{-5}$, $\nu = 2$, $h = 356$, $b =$ `Faster-RCNN`.
- `kand-logic-2`: $\gamma = 1.7\times10^{-5}$, $\nu = 2$, $h = 506$, $b =$ `Faster-RCNN`.
- `cebab`: $\gamma = 6.1\times10^{-4}$, $\nu = 16$, $h = 90$, $b =$ `BERT`.

`qwen3-8b`.

- `mn-add-bin`: $\gamma = 2.8\times10^{-3}$, $\nu = 16$, $h = 146$, $b =$ `Conv4`.
- `mn-add`: $\gamma = 2.6\times10^{-3}$, $\nu = 16$, $h = 254$, $b =$ `Conv4`.
- `mn-mul`: $\gamma = 2.6\times10^{-3}$, $\nu = 16$, $h = 188$, $b =$ `Conv4`.
- `sudoku`: $\gamma = 1.9\times10^{-3}$, $\nu = 16$, $h = 274$, $b =$ `Conv4`.
- `fashion`: $\gamma = 1.6\times10^{-3}$, $\nu = 16$, $h = 190$, $b =$ `Conv4`.
- `cifar10`: $\gamma = 4.4\times10^{-4}$, $\nu = 64$, $h = 135$, $b =$ `ResNet-18`.
- `cifar100`: $\gamma = 4.8\times10^{-4}$, $\nu = 64$, $h = 191$, $b =$ `ResNet-18`.
- `bdd-oia`: $\gamma = 2.2\times10^{-4}$, $\nu = 16$, $h = 300$, $b =$ `MLP`.
- `sushi`: $\gamma = 4.7\times10^{-4}$, $\nu = 32$, $h = 512$, $b =$ `MLP`.
- `cle4evr`: $\gamma = 9.0\times10^{-5}$, $\nu = 2$, $h = 356$, $b =$ `Faster-RCNN`.
- `kand-logic-2`: $\gamma = 1.8\times10^{-5}$, $\nu = 2$, $h = 499$, $b =$ `Faster-RCNN`.
- `chx`: $\gamma = 5.4\times10^{-5}$, $\nu = 16$, $h = 414$, $b =$ `ResNet-18`.

Due to time constraints, we reused the same hyperparameters for `SL` (Section I.2.3).

H.1 SPL Model Architectures

Six architectures f are used as backbones across the `SPL` and `SL` experiments. `Conv4` (Meng et al., 2023) and `ResNet-18` (He et al., 2016) are trained *from scratch* (random initialization); `Faster-RCNN` (Ren et al., 2017) is `COCO`-pretrained (Lin et al., 2014) and fine-tuned jointly with the head; `BERT` (Devlin et al., 2019) is frozen with a trainable head; `Slowfast-50` (Feichtenhofer et al., 2019) is trained from scratch. We assign: `Conv4` for the `MNIST`-based tasks, `sudoku`, `fashion` and `warcraft`; `ResNet-18` for `cifar10`, `cifar100` and `chx`; a 2-3 layer `MLP` over pre-computed feature vectors for `bdd-oia`, `bdd-oia-2` and `sushi`; `Faster-RCNN` for `cle4evr`, `kand-logic` and `kand-logic-2`; `bert-base-uncased` for `CEB`; and `Slowfast-50` for `road-r`. The `ResNet-18` and `BERT` stacks are the standard `torchvision`/`transformers` definitions (see Tables 12 and 14). The remaining three are detailed below.

`Conv4`. A four-block CNN used for the `MNIST`-based tasks, i.e., `sudoku`, `fashion` and `warcraft`. Table 11 details the layer configuration.

Table 11: **`Conv4` architecture** (`MNIST`-based tasks, `sudoku`, `fashion` and `warcraft`). C_{in} is 1 for grayscale and 3 for RGB inputs.

Input	Layer Type	Parameter	Activation
(C_{in}, H, W)	Conv2d + BatchNorm2d	depth= 64, kernel= 3, stride= 1, padding= 1	ReLU
$(64, H, W)$	MaxPool2d	kernel= 2, stride= 2	
$\times 3$ more identical (Conv2d + BN + ReLU + MaxPool2d) blocks, depth $64 \to 64$			
$(64, H', W')$	AdaptiveAvgPool2d	output= $(1, 1)$	
$(64, 1, 1)$	Flatten	dim= 64	
(64)	Linear	$64 \to$ `output_size`	

`ResNet-18`. The standard He et al. (2016) `ResNet-18`, randomly initialized (`weights=None`) and trained jointly with the `SPL` head from scratch, used for `cifar10`, `cifar100` and `chx`. The stem convolution is swapped only when the input is not 3-channel; every other layer is the unmodified `torchvision` definition. Table 12 details the layer configuration.

Table 12: **`ResNet-18` architecture** (`cifar10`, `cifar100`, `chx`). C_{in} is 3 for all three tasks; (H, W) is 32×32 for `cifar10`, `cifar100` and 224×224 for `chx`.

INPUT	LAYER TYPE	PARAMETER	ACTIVATION
(C_{in}, H, W)	Conv2d + BatchNorm2d	depth= 64, kernel= 7, stride= 2, padding= 3	ReLU
$(64, H/2, W/2)$	MaxPool2d	kernel= 3, stride= 2, padding= 1	
$(64, H/4, W/4)$	BasicBlock $\times$ 2 (`layer1`)	depth $64\to64$, stride= 1	ReLU
$(64, H/4, W/4)$	BasicBlock $\times$ 2 (`layer2`)	depth $64\to128$, stride= 2 (block 1)	ReLU
$(128, H/8, W/8)$	BasicBlock $\times$ 2 (`layer3`)	depth $128\to256$, stride= 2 (block 1)	ReLU
$(256, H/16, W/16)$	BasicBlock $\times$ 2 (`layer4`)	depth $256\to512$, stride= 2 (block 1)	ReLU
$(512, H/32, W/32)$	AdaptiveAvgPool2d	output= $(1, 1)$	
$(512, 1, 1)$	Flatten	dim= 512	
(512)	Linear	$512 \to$ `output_size`	

`MLP`. For tasks whose inputs are pre-extracted feature vectors rather than images, i.e., `bdd-oia` (2048-d `Faster-RCNN` embeddings) and Sushi (36-d one-hot context ranking), a small 2-3 layer `MLP` is trained from scratch. Table 13 details the configuration.

Table 13: **`MLP` backbone** (`bdd-oia`, `sushi`). d_{in} is 2048 for BDD-OIA and 36 for `sushi`.

INPUT	LAYER TYPE	PARAMETER	ACTIVATION
(d_{in})	Linear	$d_{\text{in}} \to h$	ReLU
(h)	Linear	$h \to h \quad (\times (L-1))$	ReLU
(h)	Linear	$h \to$ `output_size`	

`BERT`. For `cebab`, review text is tokenized (max length $n_{\text{tok}} = 128$) and encoded with a frozen `bert-base-uncased` (Devlin et al., 2019); only a small `Linear`→`ReLU` projection stack on top of the pooled `[CLS]` token is trained, sharing its h/L configuration with the `MLP` backbone (Table 13). Table 14 details the configuration.

Table 14: **`BERT` backbone** (`cebab`). $n_{\text{tok}} = 128$ is the fixed max token length; $L \in \{2, 3\}$ and h configure the trainable projection (same search parameters as Table 13). `BERT` itself is frozen; only the projection and `SPL` head receive gradients.

INPUT	LAYER TYPE	PARAMETER	ACTIVATION
$(n_{\text{tok}}, 2)$	tokenized input (`input_ids`, `attention_mask`)	max length $n_{\text{tok}} = 128$	
$(n_{\text{tok}}, 2)$	BertModel (`bert-base-uncased`, frozen)	12 Transformer encoder layers, hidden= 768, heads= 12, FFN= 3072	GELU (internal)
(768)	`[CLS]` pooled token	—	
(768)	Linear	$768 \to h$	ReLU
(h)	Linear $(\times (L-1))$	$h \to h$	ReLU
(h)	Linear	$h \to$ `output_size`	

`Slowfast-50`. For `road-r`, whose samples are short video clips, we adopt the `Slowfast-50` backbone (Feichtenhofer et al., 2019; two-pathway 3D ResNet, bottleneck block counts $[3, 4, 6, 3]$), vendored from the official `ROAD-R` codebase and trained from scratch. Each sample is an 8-frame box-crop clip of shape $(3, 8, 96, 96)$. The *fast* pathway processes all 8 frames at a reduced channel width ($\beta = 0.125$); the *slow* pathway processes every second frame ($\alpha = 2$) at full width; lateral connections fuse fast features into the slow pathway after each stage. We global-average-pool the final (`c5`) feature map over time and space and concatenate the two pathways (2048 slow + 256 fast = 2304 channels), which feeds the `SPL` head. Table 15 details the stage configuration.

Table 15: **`Slowfast-50` backbone** used for `road-r`, for an 8-frame $(3, 8, 96, 96)$ clip. Channel counts are given as slow / fast; the slow pathway runs at $T/\alpha = 4$ frames, the fast pathway at $T = 8$.

STAGE	SLOW PATHWAY	FAST PATHWAY	OUTPUT (SLOW / FAST CH.)
data	sample 4 frames ($\alpha = 2$)	all 8 frames	3 / 3
conv1	Conv3d k=$(1, 7, 7)$, s=$(1, 2, 2)$	Conv3d k=$(5, 7, 7)$, s=$(1, 2, 2)$	64 / 8
pool1	MaxPool3d k=$(1, 3, 3)$, s=$(1, 2, 2)$	MaxPool3d k=$(1, 3, 3)$, s=$(1, 2, 2)$	64 / 8
res2	3× bottleneck	3× bottleneck	256 / 32
res3	4× bottleneck, stride 2	4× bottleneck, stride 2	512 / 64
res4	6× bottleneck, stride 2	6× bottleneck, stride 2	1024 / 128
res5	3× bottleneck, dilation 2	3× bottleneck, dilation 2	2048 / 256
fusion	lateral Conv3d (s=$(\alpha, 1, 1)$) fast→slow after pool1 / res2 / res3 / res4		
head	global avg-pool over (T, H, W), concat slow $\|$ fast `Linear`$(2304, h) \to$ `ReLU` $\to$ `Linear`(h, G) (`SPL` head)		2304

I ADDITIONAL RESULTS

In this section we discuss additional results and ablation for the two stages of the auto-formalization pipeline.

I.1 AUTO-FORMALIZATION

In this section, we present additional results along the three axes of inputs, models, and outputs. We also provide an error analysis to better understand the errors of the generated formalizations. This investigation is divided into four components, i.e., input-level analysis, model-level analysis, output-level analysis, error analysis.

I.1.1 INPUT-LEVEL ANALYSIS

How does problem complexity influence auto-formalization performance? Beyond the prompt granularity studied in the main text (Section 5.1), auto-formalization performance may also depend on problem complexity. We consider two quantifiable proxies for complexity: the number of variables and clauses in the ground-truth `DIMACS` formula. Intuitively, larger formulas impose a greater formalization burden, as the model must track more variables and consistently encode more logical interactions, increasing the risk of omissions and inconsistencies. To conduct this analysis, for each benchmark, we average the F1 scores across model configurations for problems of different sizes, considering the number of variables and clauses separately. As formula sizes span several orders of magnitude, we use their base-10 logarithms and fit a separate univariate linear regression for each dataset and complexity measure. The resulting slope represents the expected change in mean F1 for a tenfold increase in the number of variables or clauses. We also report the R^2 metric, which measures the fraction of between-problem variation in F1 explained by the corresponding complexity measure. Figure 10 shows the problem-level observations, fitted regression lines, and 95% confidence intervals.

All six fitted slopes are negative: across all three datasets, mean F1 decreases as either the number of variables or clauses increases, suggesting that larger target formulas are generally more difficult to formalize. On `auto-nesy-bench`, a tenfold increase in the number of variables is associated with an F1 decrease of 0.1986 ($R^2 = 0.101$, $p = 0.198$), while a tenfold increase in the number of clauses corresponds to a decrease of 0.1391 ($R^2 = 0.452$, $p = 0.00223$). Thus, among the 18 `auto-nesy-bench` problems, clause count is more informative than variable count, although the small sample size warrants caution. On `dcpbench`, the slopes for variable and clause count are -0.0947 and -0.0845, respectively. The corresponding fits explain roughly 28-30% of the problem-level variation ($R^2 = 0.303$, $p = 4.9 \times 10^{-8}$ for variables; $R^2 = 0.281$, $p = 1.84 \times 10^{-7}$ for clauses).

On `satbench`, the slopes are -0.1921 for variables ($R^2 = 0.261$, $p = 2.49 \times 10^{-70}$) and -0.1789 for clauses ($R^2 = 0.364$, $p = 2.85 \times 10^{-104}$), indicating that mean F1 decreases with formula size on this benchmark as well.

Overall, larger target formulas are associated with lower auto-formalization performance, but the magnitude of this effect is dataset-dependent. The results also suggest that problem complexity may not be fully captured by the formula size.

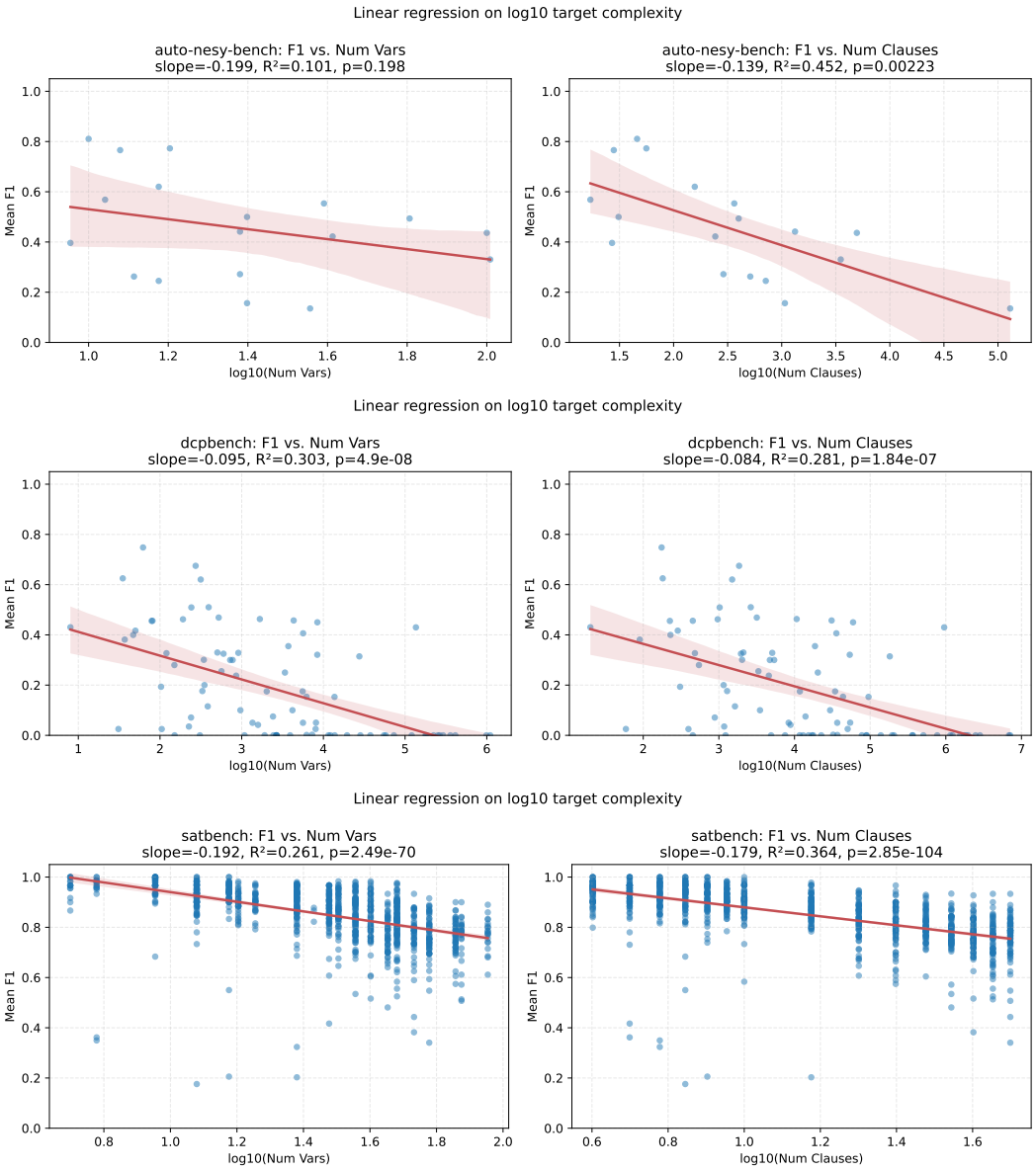


Figure 10: **Larger formulas are generally more difficult to formalize, although the magnitude of this effect is dataset-dependent.** We report mean F1 against the number of variables or clauses on a log scale for `auto-nesy-bench` (top), `dcpbench` (middle), and `satbench` (bottom). Each point represents the mean F1 for a specific benchmark instance. The red line and shaded region indicate the ordinary least-squares fit and its 95% confidence interval.

I.1.2 MODEL-LEVEL ANALYSIS

How well can each model perform under its best-performing setting? In the main text, we analyze model performance on each dataset across the ten configurations, averaging over two prompting strategies and five output formats. Here, we further examine the performance of each model by selecting its best-performing configuration. To make the comparison fair, we selected the detailed version of the prompt for `auto-nesy-bench`. For each model, we select the prompting-format configuration that achieves the highest F1 score. As shown in Table 16, selecting the best configuration for each model yields substantial improvements over averaging all configurations equally. On `auto-nesy-bench`, the mean F1 across models increases from 0.455 to 0.692; on `dcpbench`, it increases from 0.183 to 0.297; and on `satbench`, it increases from 0.849 to 0.942. These gaps indicate that matching the prompting strategy and output format to each model is important for auto-formalization performance. The selected configurations also reveal clear preferences across datasets. On `auto-nesy-bench`, ten of the eleven models select ZS-CoT, with `deepseek-r1-llama-70b` as the only exception. Seven of the eleven selected configurations use `CPMpy`, three use `SymPy`, and one uses `NAT`, indicating a strong preference for high-level symbolic programming formats. On `dcpbench`, three of the four models select ZS, while three select `CPMpy` and one selects `SymPy`. On `satbench`, all three evaluated models select ZS-CoT with `NAT`. Together, these results suggest that the preferred prompting strategy and output format depend on the dataset, although high-level code formats are predominant on `auto-nesy-bench` and `dcpbench`. Several models achieve very strong performance under their best configurations. On `auto-nesy-bench`, `gemma4-31b` achieves the highest F1 and precision, at 0.991 and 0.986, respectively, followed by `gpt-oss-120b`, with an F1 of 0.984 and precision of 0.974. Both also attain near-perfect recall, reaching 0.997 and 1.000, respectively. On `dcpbench`, `gpt-oss-120b` achieves the highest F1 (0.433) with ZS and `CPMpy`, followed by `qwen3-coder-next` (0.302) with ZS-CoT and `CPMpy`, `gpt-oss-20b` (0.286) with ZS and `CPMpy`, and `deepseek-r1-llama-70b` (0.165) with ZS and `SymPy`. On `satbench`, all three models select ZS-CoT and `NAT`, with `gemma4-31b` achieving the highest F1 of 0.992, followed by `gpt-oss-120b` at 0.976 and `qwen3-coder-next` at 0.859. These results show that, with an appropriate prompting strategy and output format, some current open-weight LLMs can formalize logical constraints from natural language descriptions with very high accuracy on `auto-nesy-bench` and `satbench`. At the same time, the substantially lower best-configuration performance on `dcpbench` indicates that reliable auto-formalization remains challenging for more difficult instances.

Table 16: **Selecting an appropriate prompting strategy and output format substantially improves best-case formalization performance.** Best formalization performance of each model on `auto-nesy-bench`, `dcpbench`, and `satbench`. For each model, we select the best of ten configurations formed by two prompting strategies and five output formats according to F1 score, breaking ties by accuracy and then by fixed prompt and format order, and report the corresponding precision, recall, and syntax error rate. Average rows give each model equal weight. Within each dataset, best results are bolded, and second-best results are underlined; averages are excluded from the ranking.

DATASET	MODEL	PROMPT	FORMAT	F1 (↑)	PRECISION (↑)	RECALL (↑)	SYN. ER (↓)
auto-nesy-bench	deepseek-r1-llama-70b	ZS	SymPy	0.676	0.640	0.810	11.1%
	gemma3-27b	ZS-CoT	NAT	0.135	0.192	0.449	22.2%
	gemma4-31b	ZS-CoT	CPMpy	**0.991**	**0.986**	0.997	**0.0%**
	gpt-oss-120b	ZS-CoT	CPMpy	0.984	0.974	**1.000**	**0.0%**
	gpt-oss-20b	ZS-CoT	SymPy	0.769	0.756	0.833	16.7%
	mistral-nemo	ZS-CoT	CPMpy	0.259	0.256	0.325	50.0%
	olmo3-32b-think	ZS-CoT	CPMpy	0.733	0.711	0.833	11.1%
	phi4-reasoning-plus	ZS-CoT	SymPy	0.721	0.720	0.722	27.8%
	qwen3-32b	ZS-CoT	CPMpy	0.935	0.922	**1.000**	**0.0%**
	qwen3-8b	ZS-CoT	CPMpy	0.649	0.694	0.664	11.1%
	qwen3-coder-next	ZS-CoT	CPMpy	0.755	0.802	0.770	16.7%
	AVERAGE	—	—	0.692	0.696	0.764	15.2%
dcpbench	deepseek-r1-llama-70b	ZS	SymPy	0.165	0.162	0.271	37.6%
	gpt-oss-120b	ZS	CPMpy	**0.433**	**0.441**	**0.506**	**5.9%**
	gpt-oss-20b	ZS	CPMpy	0.286	0.286	0.333	36.5%
	qwen3-coder-next	ZS-CoT	CPMpy	0.302	0.318	0.312	29.4%
	AVERAGE	—	—	0.297	0.302	0.355	27.4%
satbench	gemma4-31b	ZS-CoT	NAT	**0.992**	**0.991**	**0.994**	**0.0%**
	gpt-oss-120b	ZS-CoT	NAT	0.976	0.973	0.988	**0.0%**
	qwen3-coder-next	ZS-CoT	NAT	0.859	0.849	0.907	2.0%
	AVERAGE	—	—	0.942	0.937	0.963	0.7%

Complete Section 5.1 results. Tables 17 to 19 report the underlying results for every available model, prompting strategy, and output format. Each entry is averaged across the problems in the corresponding dataset.

Table 17: Complete per-configuration formalization results on `auto-nesy-bench`. ZS and ZS-CoT denote zero-shot and zero-shot chain-of-thought prompting. Missing task scores are assigned zero. Syntax ER is taken directly from the selected results JSON.

Model	Prompt	Format	F1 (↑)	Precision (↑)	Recall (↑)	Syntax ER (↓)
deepseek-r1-llama-70b	ZS	CPMpy	0.503	0.532	0.551	33.3%
deepseek-r1-llama-70b	ZS	DIMACS	0.523	0.522	0.587	16.7%
deepseek-r1-llama-70b	ZS	NAT	0.363	0.409	0.619	5.6%
deepseek-r1-llama-70b	ZS	PySAT	0.541	0.695	0.604	16.7%
deepseek-r1-llama-70b	ZS	SymPy	0.676	0.640	0.810	11.1%
deepseek-r1-llama-70b	ZS-CoT	CPMpy	0.111	0.111	0.167	83.3%
deepseek-r1-llama-70b	ZS-CoT	DIMACS	0.388	0.360	0.484	11.1%
deepseek-r1-llama-70b	ZS-CoT	NAT	0.390	0.451	0.578	16.7%
deepseek-r1-llama-70b	ZS-CoT	PySAT	0.316	0.325	0.405	38.9%
deepseek-r1-llama-70b	ZS-CoT	SymPy	0.313	0.300	0.462	50.0%
gemma3-27b	ZS	CPMpy	0.049	0.043	0.167	83.3%
gemma3-27b	ZS	DIMACS	0.006	0.020	0.062	11.1%
gemma3-27b	ZS	NAT	0.064	0.060	0.361	33.3%
gemma3-27b	ZS	PySAT	0.056	0.065	0.077	66.7%
gemma3-27b	ZS	SymPy	0.029	0.017	0.222	77.8%
gemma3-27b	ZS-CoT	CPMpy	0.056	0.056	0.111	88.9%
gemma3-27b	ZS-CoT	DIMACS	0.002	0.022	0.061	16.7%
gemma3-27b	ZS-CoT	NAT	0.135	0.192	0.449	22.2%
gemma3-27b	ZS-CoT	PySAT	0.003	0.002	0.017	66.7%
gemma3-27b	ZS-CoT	SymPy	0.001	0.000	0.111	88.9%
gemma4-31b	ZS	CPMpy	0.886	0.887	0.941	5.6%
gemma4-31b	ZS	DIMACS	0.254	0.320	0.365	5.6%
gemma4-31b	ZS	NAT	0.334	0.461	0.462	0.0%
gemma4-31b	ZS	PySAT	0.770	0.863	0.767	5.6%
gemma4-31b	ZS	SymPy	0.748	0.728	0.859	5.6%
gemma4-31b	ZS-CoT	CPMpy	0.991	0.986	0.997	0.0%
gemma4-31b	ZS-CoT	DIMACS	0.715	0.731	0.807	5.6%
gemma4-31b	ZS-CoT	NAT	0.586	0.591	0.791	0.0%
gemma4-31b	ZS-CoT	PySAT	0.900	0.889	0.917	5.6%
gemma4-31b	ZS-CoT	SymPy	0.873	0.863	0.915	5.6%
gpt-oss-120b	ZS	CPMpy	0.925	0.971	0.939	0.0%
gpt-oss-120b	ZS	DIMACS	0.640	0.664	0.719	11.1%
gpt-oss-120b	ZS	NAT	0.562	0.568	0.650	16.7%
gpt-oss-120b	ZS	PySAT	0.813	0.856	0.828	0.0%
gpt-oss-120b	ZS	SymPy	0.812	0.804	0.826	16.7%
gpt-oss-120b	ZS-CoT	CPMpy	0.984	0.974	1.000	0.0%
gpt-oss-120b	ZS-CoT	DIMACS	0.687	0.720	0.792	5.6%
gpt-oss-120b	ZS-CoT	NAT	0.645	0.629	0.796	16.7%
gpt-oss-120b	ZS-CoT	PySAT	0.924	0.914	0.941	0.0%
gpt-oss-120b	ZS-CoT	SymPy	0.871	0.863	0.885	11.1%
gpt-oss-20b	ZS	CPMpy	0.703	0.695	0.719	27.8%
gpt-oss-20b	ZS	DIMACS	0.563	0.551	0.641	22.2%
gpt-oss-20b	ZS	NAT	0.328	0.423	0.375	50.0%
gpt-oss-20b	ZS	PySAT	0.403	0.584	0.485	11.1%
gpt-oss-20b	ZS	SymPy	0.616	0.665	0.614	33.3%
gpt-oss-20b	ZS-CoT	CPMpy	0.634	0.666	0.625	33.3%
gpt-oss-20b	ZS-CoT	DIMACS	0.366	0.400	0.472	38.9%
gpt-oss-20b	ZS-CoT	NAT	0.283	0.391	0.334	38.9%
gpt-oss-20b	ZS-CoT	PySAT	0.383	0.511	0.458	27.8%
gpt-oss-20b	ZS-CoT	SymPy	0.769	0.756	0.833	16.7%
mistral-nemo	ZS	CPMpy	0.094	0.085	0.222	66.7%
mistral-nemo	ZS	DIMACS	0.036	0.026	0.202	33.3%
mistral-nemo	ZS	NAT	0.000	0.013	0.000	77.8%
mistral-nemo	ZS	PySAT	0.020	0.012	0.184	66.7%
mistral-nemo	ZS	SymPy	0.000	0.000	0.056	94.4%
mistral-nemo	ZS-CoT	CPMpy	0.259	0.256	0.325	50.0%
mistral-nemo	ZS-CoT	DIMACS	0.058	0.054	0.184	33.3%
mistral-nemo	ZS-CoT	NAT	0.018	0.011	0.108	77.8%
mistral-nemo	ZS-CoT	PySAT	0.039	0.051	0.206	33.3%
mistral-nemo	ZS-CoT	SymPy	0.007	0.003	0.056	94.4%
olmo3-32b-think	ZS	CPMpy	0.547	0.553	0.598	33.3%
olmo3-32b-think	ZS	DIMACS	0.580	0.572	0.676	5.6%
olmo3-32b-think	ZS	NAT	0.416	0.431	0.639	16.7%

Continued on next page

Table 17 – continued

Model	Prompt	Format	F1 (↑)	Precision (↑)	Recall (↑)	Syntax ER (↓)
olmo3-32b-think	ZS	PySAT	0.650	0.722	0.676	16.7%
olmo3-32b-think	ZS	SymPy	0.698	0.735	0.758	11.1%
olmo3-32b-think	ZS-CoT	CPMpy	0.733	0.711	0.833	11.1%
olmo3-32b-think	ZS-CoT	DIMACS	0.372	0.569	0.574	5.6%
olmo3-32b-think	ZS-CoT	NAT	0.415	0.449	0.504	27.8%
olmo3-32b-think	ZS-CoT	PySAT	0.567	0.675	0.631	16.7%
olmo3-32b-think	ZS-CoT	SymPy	0.716	0.676	0.878	5.6%
phi4-reasoning-plus	ZS	CPMpy	0.499	0.554	0.611	33.3%
phi4-reasoning-plus	ZS	DIMACS	0.167	0.222	0.167	72.2%
phi4-reasoning-plus	ZS	NAT	0.249	0.299	0.253	55.6%
phi4-reasoning-plus	ZS	PySAT	0.333	0.333	0.333	66.7%
phi4-reasoning-plus	ZS	SymPy	0.613	0.612	0.667	27.8%
phi4-reasoning-plus	ZS-CoT	CPMpy	0.548	0.544	0.552	38.9%
phi4-reasoning-plus	ZS-CoT	DIMACS	0.222	0.222	0.222	66.7%
phi4-reasoning-plus	ZS-CoT	NAT	0.333	0.333	0.333	66.7%
phi4-reasoning-plus	ZS-CoT	PySAT	0.542	0.541	0.543	38.9%
phi4-reasoning-plus	ZS-CoT	SymPy	0.721	0.720	0.722	27.8%
qwen3-32b	ZS	CPMpy	0.838	0.863	0.868	0.0%
qwen3-32b	ZS	DIMACS	0.574	0.615	0.692	11.1%
qwen3-32b	ZS	NAT	0.530	0.555	0.610	16.7%
qwen3-32b	ZS	PySAT	0.644	0.630	0.667	22.2%
qwen3-32b	ZS	SymPy	0.690	0.694	0.687	22.2%
qwen3-32b	ZS-CoT	CPMpy	0.935	0.922	1.000	0.0%
qwen3-32b	ZS-CoT	DIMACS	0.468	0.488	0.485	38.9%
qwen3-32b	ZS-CoT	NAT	0.536	0.595	0.680	11.1%
qwen3-32b	ZS-CoT	PySAT	0.717	0.726	0.741	16.7%
qwen3-32b	ZS-CoT	SymPy	0.618	0.689	0.693	5.6%
qwen3-8b	ZS	CPMpy	0.498	0.483	0.548	11.1%
qwen3-8b	ZS	DIMACS	0.291	0.380	0.431	11.1%
qwen3-8b	ZS	NAT	0.292	0.285	0.403	33.3%
qwen3-8b	ZS	PySAT	0.122	0.181	0.151	61.1%
qwen3-8b	ZS	SymPy	0.600	0.596	0.604	11.1%
qwen3-8b	ZS-CoT	CPMpy	0.649	0.694	0.664	11.1%
qwen3-8b	ZS-CoT	DIMACS	0.435	0.510	0.475	11.1%
qwen3-8b	ZS-CoT	NAT	0.272	0.272	0.356	44.4%
qwen3-8b	ZS-CoT	PySAT	0.000	0.056	0.000	88.9%
qwen3-8b	ZS-CoT	SymPy	0.564	0.563	0.601	22.2%
qwen3-coder-next	ZS	CPMpy	0.460	0.498	0.480	38.9%
qwen3-coder-next	ZS	DIMACS	0.135	0.126	0.481	5.6%
qwen3-coder-next	ZS	NAT	0.235	0.231	0.305	27.8%
qwen3-coder-next	ZS	PySAT	0.491	0.540	0.582	16.7%
qwen3-coder-next	ZS	SymPy	0.538	0.558	0.527	16.7%
qwen3-coder-next	ZS-CoT	CPMpy	0.755	0.802	0.770	16.7%
qwen3-coder-next	ZS-CoT	DIMACS	0.519	0.584	0.703	11.1%
qwen3-coder-next	ZS-CoT	NAT	0.303	0.295	0.444	44.4%
qwen3-coder-next	ZS-CoT	PySAT	0.729	0.827	0.751	0.0%
qwen3-coder-next	ZS-CoT	SymPy	0.611	0.608	0.718	22.2%

Table 18: Complete per-configuration formalization results on dcpbench. ZS and ZS-CoT denote zero-shot and zero-shot chain-of-thought prompting. Missing task scores are assigned zero. Syntax ER is taken directly from the selected results JSON.

Model	Prompt	Format	F1 (↑)	Precision (↑)	Recall (↑)	Syntax ER (↓)
deepseek-r1-llama-70b	ZS	CPMpy	0.121	0.119	0.200	40.0%
deepseek-r1-llama-70b	ZS	DIMACS	0.078	0.075	0.319	25.9%
deepseek-r1-llama-70b	ZS	NAT	0.061	0.063	0.210	29.4%
deepseek-r1-llama-70b	ZS	PySAT	0.085	0.088	0.174	30.6%
deepseek-r1-llama-70b	ZS	SymPy	0.165	0.162	0.271	37.6%
deepseek-r1-llama-70b	ZS-CoT	CPMpy	0.067	0.071	0.124	44.7%
deepseek-r1-llama-70b	ZS-CoT	DIMACS	0.030	0.030	0.270	18.8%
deepseek-r1-llama-70b	ZS-CoT	NAT	0.071	0.073	0.270	30.6%
deepseek-r1-llama-70b	ZS-CoT	PySAT	0.049	0.046	0.076	69.4%
deepseek-r1-llama-70b	ZS-CoT	SymPy	0.052	0.048	0.106	68.2%
gpt-oss-120b	ZS	CPMpy	0.433	0.441	0.506	5.9%
gpt-oss-120b	ZS	DIMACS	0.266	0.297	0.383	5.9%
gpt-oss-120b	ZS	NAT	0.184	0.200	0.263	45.9%
gpt-oss-120b	ZS	PySAT	0.410	0.421	0.465	3.5%
gpt-oss-120b	ZS	SymPy	0.356	0.367	0.469	17.6%
gpt-oss-120b	ZS-CoT	CPMpy	0.391	0.395	0.467	12.9%
gpt-oss-120b	ZS-CoT	DIMACS	0.266	0.301	0.344	21.2%
gpt-oss-120b	ZS-CoT	NAT	0.141	0.167	0.210	55.3%

Continued on next page

Table 18 – continued

Model	Prompt	Format	F1 (↑)	Precision (↑)	Recall (↑)	Syntax ER (↓)
gpt-oss-120b	ZS-CoT	PySAT	0.429	0.439	0.458	3.5%
gpt-oss-120b	ZS-CoT	SymPy	0.360	0.371	0.433	8.2%
gpt-oss-20b	ZS	CPMpy	0.286	0.286	0.333	36.5%
gpt-oss-20b	ZS	DIMACS	0.183	0.194	0.261	40.0%
gpt-oss-20b	ZS	NAT	0.122	0.120	0.180	64.7%
gpt-oss-20b	ZS	PySAT	0.123	0.136	0.177	21.2%
gpt-oss-20b	ZS	SymPy	0.261	0.260	0.312	40.0%
gpt-oss-20b	ZS-CoT	CPMpy	0.214	0.213	0.235	40.0%
gpt-oss-20b	ZS-CoT	DIMACS	0.157	0.156	0.227	51.8%
gpt-oss-20b	ZS-CoT	NAT	0.106	0.106	0.172	69.4%
gpt-oss-20b	ZS-CoT	PySAT	0.244	0.247	0.276	25.9%
gpt-oss-20b	ZS-CoT	SymPy	0.232	0.237	0.271	41.2%
qwen3-coder-next	ZS	CPMpy	0.164	0.167	0.182	69.4%
qwen3-coder-next	ZS	DIMACS	0.005	0.003	0.594	0.0%
qwen3-coder-next	ZS	NAT	0.045	0.041	0.142	35.3%
qwen3-coder-next	ZS	PySAT	0.118	0.122	0.151	51.8%
qwen3-coder-next	ZS	SymPy	0.053	0.051	0.118	72.9%
qwen3-coder-next	ZS-CoT	CPMpy	0.302	0.318	0.312	29.4%
qwen3-coder-next	ZS-CoT	DIMACS	0.188	0.200	0.467	7.1%
qwen3-coder-next	ZS-CoT	NAT	0.143	0.142	0.300	32.9%
qwen3-coder-next	ZS-CoT	PySAT	0.202	0.200	0.229	21.2%
qwen3-coder-next	ZS-CoT	SymPy	0.170	0.181	0.197	48.2%

Table 19: Complete per-configuration formalization results on satbench. ZS and ZS-CoT denote zero-shot and zero-shot chain-of-thought prompting. Missing task scores are assigned zero. Syntax ER is taken directly from the selected results JSON.

Model	Prompt	Format	F1 (↑)	Precision (↑)	Recall (↑)	Syntax ER (↓)
gemma4-31b	ZS	CPMpy	0.988	0.987	0.991	0.1%
gemma4-31b	ZS	DIMACS	0.530	0.520	0.568	0.0%
gemma4-31b	ZS	NAT	0.965	0.962	0.980	0.0%
gemma4-31b	ZS	PySAT	0.982	0.981	0.986	0.0%
gemma4-31b	ZS	SymPy	0.970	0.970	0.976	1.0%
gemma4-31b	ZS-CoT	CPMpy	0.991	0.990	0.995	0.0%
gemma4-31b	ZS-CoT	DIMACS	0.974	0.973	0.978	0.0%
gemma4-31b	ZS-CoT	NAT	0.992	0.991	0.994	0.0%
gemma4-31b	ZS-CoT	PySAT	0.975	0.974	0.981	0.0%
gemma4-31b	ZS-CoT	SymPy	0.988	0.988	0.994	0.0%
gpt-oss-120b	ZS	CPMpy	0.967	0.963	0.985	0.0%
gpt-oss-120b	ZS	DIMACS	0.966	0.962	0.978	0.0%
gpt-oss-120b	ZS	NAT	0.970	0.966	0.985	0.2%
gpt-oss-120b	ZS	PySAT	0.951	0.945	0.971	0.0%
gpt-oss-120b	ZS	SymPy	0.960	0.954	0.984	0.0%
gpt-oss-120b	ZS-CoT	CPMpy	0.972	0.968	0.989	0.0%
gpt-oss-120b	ZS-CoT	DIMACS	0.961	0.957	0.976	0.0%
gpt-oss-120b	ZS-CoT	NAT	0.976	0.973	0.988	0.0%
gpt-oss-120b	ZS-CoT	PySAT	0.954	0.948	0.971	0.1%
gpt-oss-120b	ZS-CoT	SymPy	0.958	0.953	0.981	0.0%
qwen3-coder-next	ZS	CPMpy	0.632	0.631	0.639	32.4%
qwen3-coder-next	ZS	DIMACS	0.123	0.116	0.156	1.8%
qwen3-coder-next	ZS	NAT	0.756	0.744	0.807	3.7%
qwen3-coder-next	ZS	PySAT	0.614	0.610	0.627	31.5%
qwen3-coder-next	ZS	SymPy	0.704	0.701	0.753	13.6%
qwen3-coder-next	ZS-CoT	CPMpy	0.700	0.696	0.720	24.2%
qwen3-coder-next	ZS-CoT	DIMACS	0.639	0.631	0.702	0.5%
qwen3-coder-next	ZS-CoT	NAT	0.859	0.849	0.907	2.0%
qwen3-coder-next	ZS-CoT	PySAT	0.764	0.757	0.791	10.6%
qwen3-coder-next	ZS-CoT	SymPy	0.687	0.682	0.716	14.9%

How does auto-formalization performance vary with model size? Understanding how model capabilities change with scale is a central question in the study of LLMs. Comparing models with different parameters can reveal whether a capability improves with scale and how consistent this relationship is. We therefore study how auto-formalization performance varies with model size. For each model, we average F1 across the ten configurations formed by two prompting strategies and five output formats. We then fit an ordinary least-squares regression of the mean F1 compared to the LLM parameter count on a log-10 scale. The slope measures the expected change in mean F1 associated with a tenfold increase in model size. Our comparison includes eleven models on

auto-nesy-bench, ranging from 8.2B to 117B total parameters, four models on dcpbench, and three models on satbench.

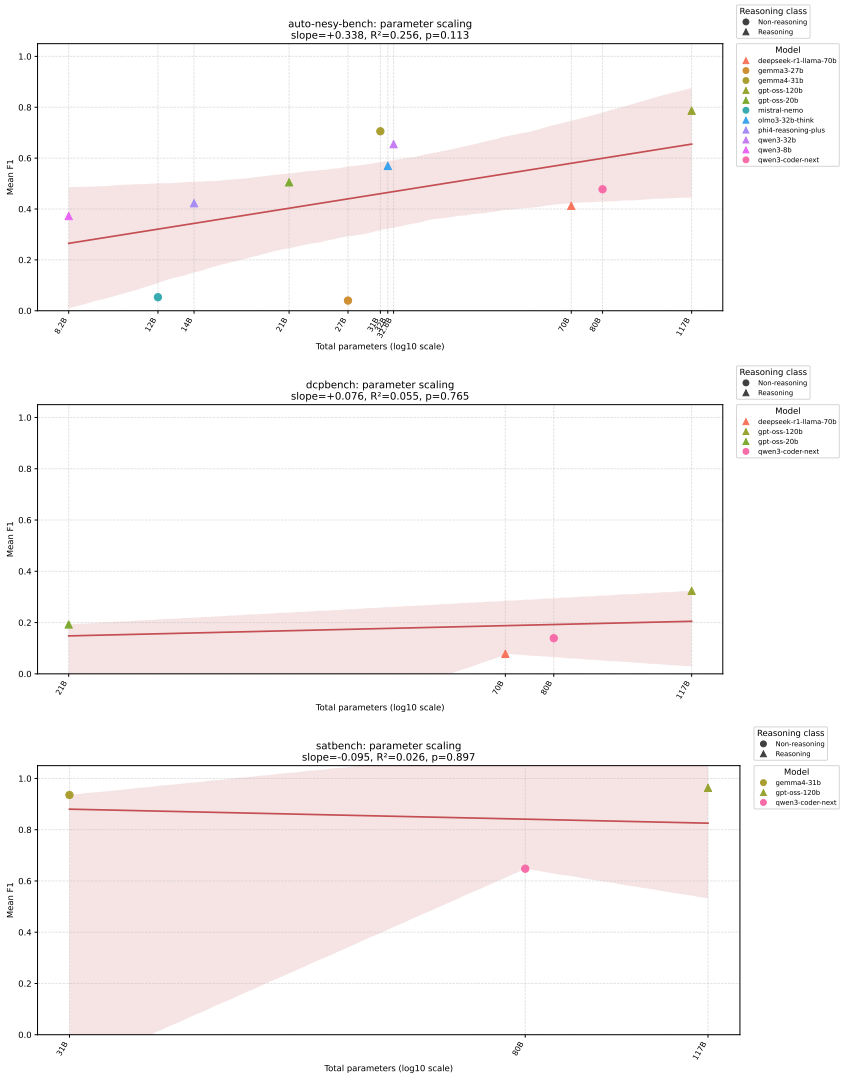


Figure 11: **The relationship between model size and auto-formalization performance varies across datasets. Larger models tend to perform better on auto-nesy-bench and dcpbench, but this trend does not extend to satbench.** We report mean F1 as a function of total model parameter count on auto-nesy-bench (top), dcpbench (middle), and satbench (bottom). Each point represents a model mean F1 across matched runs. Circles denote non-reasoning models, while triangles denote reasoning models. The red line and shaded region show the ordinary least-squares fit and its 95% confidence interval, respectively. Parameter count is shown on a base-10 logarithmic scale. None of the fitted associations is statistically significant.

As shown in Fig. 11, the connection between auto-formalization performance and model size varies across datasets. On auto-nesy-bench, the fitted slope is $+0.3380$, corresponding to an estimated F1 increase of 0.3380 for a tenfold increase in parameter count. The correlation is moderate ($r = 0.506$), with model size explaining 25.6% of the between-model variation ($R^2 = 0.256$); however, the association is not statistically significant ($p = 0.113$). With only eleven models spanning different architectures, model families, and training procedures, this result does not establish a general scaling law. On dcpbench, the fitted slope is smaller ($+0.0764$), with $R^2 = 0.055$ and $p = 0.765$. This association is also not statistically significant, and the regression is based on only four models; it should therefore be interpreted as a descriptive trend rather than evidence of general scaling. On satbench, the

fitted slope is weakly negative (-0.0948), with $r = -0.161$, $R^2 = 0.026$, and $p = 0.897$. The 117B `gpt-oss-120b` achieves higher F1 than the 80B `qwen3-coder-next` (0.963 vs. 0.648), but the 31B `gemma4-31b` also substantially outperforms the latter (0.936 vs. 0.648). With only three models, this comparison cannot establish a scaling relationship. Model size also does not fully determine performance. On `auto-nesy-bench`, `gpt-oss-20b` and `olmo3-32b-think` achieve F1 scores of 0.505 and 0.569, respectively, outperforming the larger 70B `deepseek-r1-llama-70b` (0.412) and 80B `qwen3-coder-next` (0.478). Even `qwen3-8b` achieves an F1 of 0.372. Overall, larger models tend to perform better on `auto-nesy-bench` and `dcpbench`, but this trend does not extend to `satbench`, and parameter count explains only part of the observed variation. The deviations from the fitted trends suggest that architecture, training, and other model characteristics also contribute substantially to auto-formalization performance.

I.1.3 Output-Level Analysis

Which output format achieves the best performance? The proposed pipeline allows an LLM to generate a CNF in one of five output formats: `DIMACS`, `NAT`, `PySAT`, `CPMpy`, or `SymPy` code. Each representation is then deterministically compiled into `DIMACS`, providing a common representation for evaluation. The choice of output format substantially affects performance. In the main paper, we report the average performance of each output format across matched runs. Here, we break down these results by model to examine how individual models perform across output formats. Figure 12 shows the F1 score for each model-format pair, averaged over the two prompting strategies (ZS and ZS-CoT). Output formats are ordered from highest to lowest according to their mean F1 across models. Models are similarly ordered according to their mean F1 across output formats.

Code-based formats perform best on average on `auto-nesy-bench`. `CPMpy` achieves the highest across-model mean F1 at 0.575, followed by `SymPy` at 0.549. `PySAT` ranks third at 0.453, while direct `DIMACS` generation reaches 0.364 and `NAT` performs worst at 0.331. Thus, compared with generating the final representation directly, the two strongest higher-level formats improve mean F1 by approximately 0.19-0.21. This suggests that allowing models to generate a high-level symbolic program may make formalization easier than requiring them to construct the low-level `DIMACS` encoding directly, potentially because code is more prevalent in LLM training data.

However, the best output format varies across models. The strongest model overall on `auto-nesy-bench`, `gpt-oss-120b`, achieves its highest F1 with `CPMpy` (0.954), compared with 0.869 for `PySAT`, 0.841 for `SymPy`, 0.664 for `DIMACS`, and 0.604 for `NAT`. In contrast, `SymPy` performs best for several other competitive models, including `olmo3-32b-think` (0.707), `gpt-oss-20b` (0.693), and `qwen-8b` (0.582). `qwen-coder-next` performs similarly with `PySAT` (0.610) and `CPMpy` (0.608). Thus, while `CPMpy` and `SymPy` are the strongest formats on average, choosing the format for a specific model can yield substantial additional gains.

A similar preference for code-based formats emerges on `dcpbench`, although the best format varies more substantially across models. Across models, `CPMpy` achieves the highest mean F1 (0.247), followed by `PySAT` (0.207) and `SymPy` (0.206). At the model level, however, `deepseek-r1-llama-70b` performs best with `SymPy` (0.109), `gpt-oss-120b` with `PySAT` (0.419), `gpt-oss-20b` with `CPMpy` (0.250), and `qwen-coder-next` with `CPMpy` (0.233). No single code-based format is therefore consistently best across models.

The pattern is different on `satbench`. Here, `NAT` is the strongest format on average across models, achieving an F1 of 0.920. It is also the strongest format for `gpt-oss-120b` (0.973) and `qwen-coder-next` (0.808), while `gemma4-31b` performs best with `CPMpy` (0.989), compared with 0.979 for `NAT`. Thus, the advantage of code-based representations observed on `auto-nesy-bench` and `dcpbench` does not transfer uniformly across datasets.

Overall, these results show that no single output format consistently dominates across models and datasets. Higher-level code representations, particularly `CPMpy` and `SymPy`, perform well on `auto-nesy-bench` and `dcpbench`, whereas `NAT` is substantially stronger on `satbench`. The choice of output format should therefore be viewed as a model- and dataset-dependent factor in formalization performance.

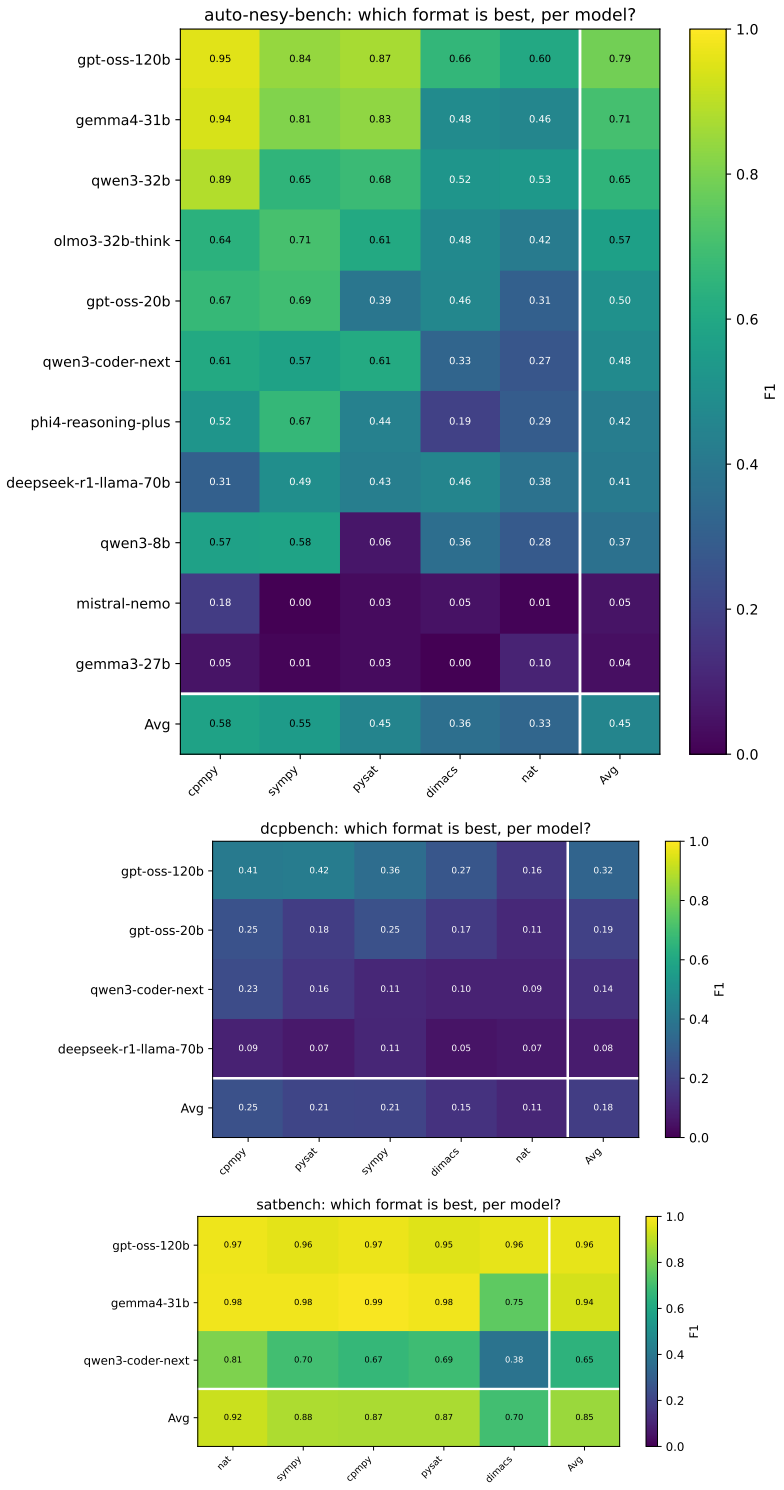


Figure 12: **Code representations, particularly `CPMpy` and `SymPy`, are generally effective on `auto-nesy-bench` and `dcpbench`, while `NAT` performs best on `satbench`.** We report mean F1 by model and output format on `auto-nesy-bench` (top), `dcpbench` (middle), and `satbench` (bottom). Each cell reports F1 averaged across the two prompting strategies for a given model-format pair. The rightmost column shows the mean across formats for each model, while the bottom row shows the mean across models for each format. Missing F1 scores are treated as zero.

I.1.4 Syntactic Error Analysis

Finally, we examine which errors most often prevent the auto-formalization pipeline from producing valid formulas and whether these errors differ across models and output formats. We focus on syntactic errors, including missing output delimiters, parsing errors, malformed `DIMACS` content, program-execution failures, and timeouts. The complete error taxonomy is provided in Section C.

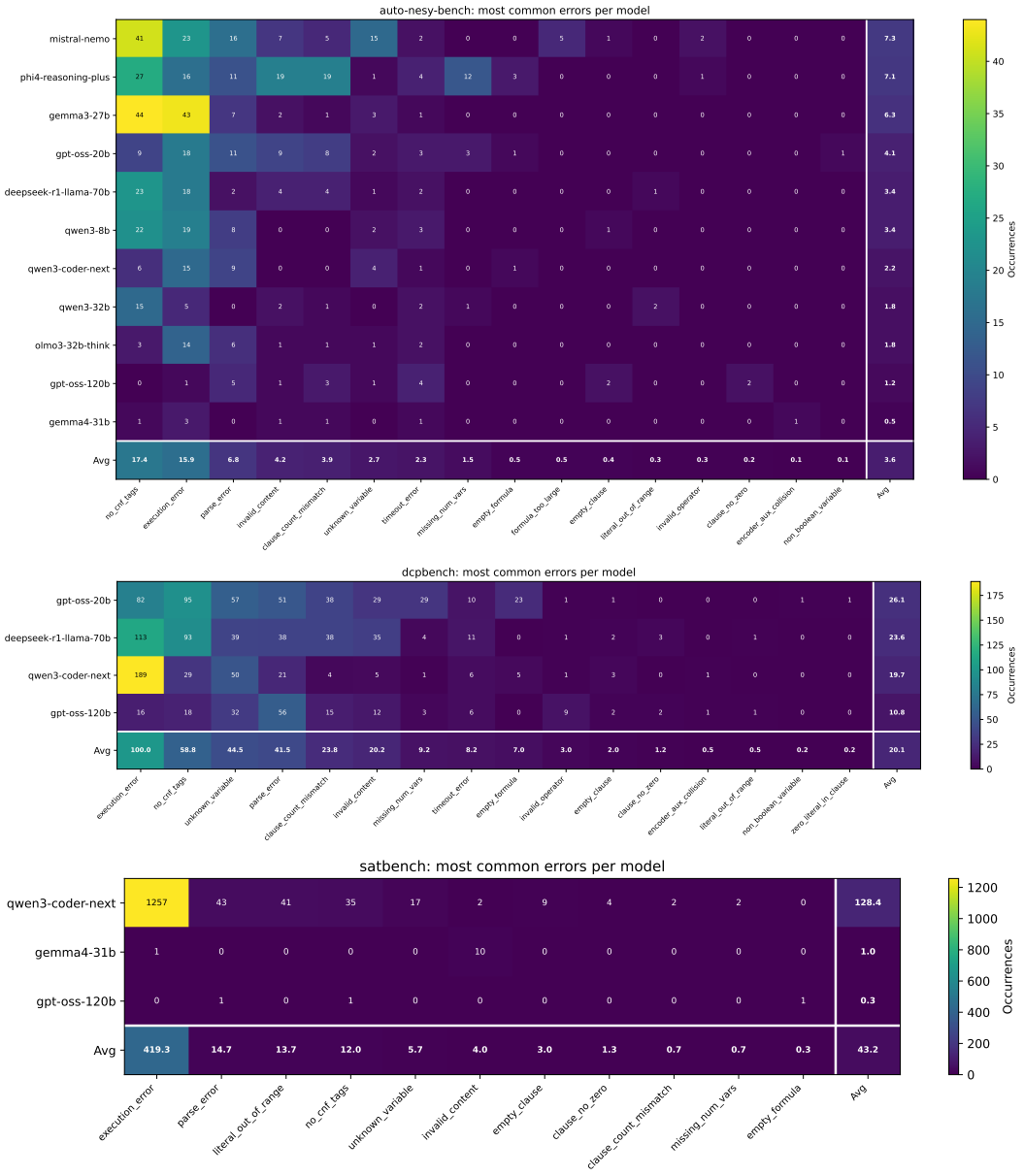


Figure 13: **Syntactic errors are concentrated in a small number of categories and models.** Occurrences of each error code in the final model outputs on `auto-nesy-bench` (top), `dcpbench` (middle), and `satbench` (bottom), aggregated across output formats and prompting strategies. A single output can contribute to multiple error codes. The rightmost column reports the mean across error types, and the bottom row reports the mean across models.

Figure 13 shows that errors are concentrated in a small number of categories and models. On `auto-nesy-bench`, `no_cnf_tags` is the most common error category, with 17.4 occurrences, followed by `execution_error` (15.9), `parse_error` (6.8), `invalid_content` (4.2), and `clause_count_mismatch` (3.9). `mistral-nemo` has the most errors overall, with an av-

erage of 7.3 errors per error type. This includes 23 execution errors and 41 missing-tag errors. `phi4-reasoning-plus` follows with an average of 7.1 errors per type, while `gemma3-27b` averages 6.3, with 43 execution errors and 44 missing-tag errors. In contrast, `gemma4-31b` has only 0.5 errors per type, and `gpt-oss-120b` has 1.2, with one execution error and no missing-tag errors. These results show that auto-formalization depends not only on the LLM reasoning ability, but also on its ability to follow formatting instructions and produce outputs that can be parsed and executed correctly. On `dcpbench`, execution errors are the most common category, averaging 100.0 occurrences per model, followed by missing tags (58.8), unknown variables (44.5), and parsing errors (41.5). `gpt-oss-20b` has the largest mean count across error types (26.1), followed by `deepseek-r1-llama-70b` (23.6), `qwen3-coder-next` (19.7), and `gpt-oss-120b` (10.8). On `satbench`, `qwen3-coder-next` has the most errors, with 1,257 execution errors and an average of 128.4 errors per type. In contrast, `gpt-oss-120b` has only three errors in total, with no execution errors. `gemma4-31b` has eleven errors: ten invalid-content errors and one execution error.

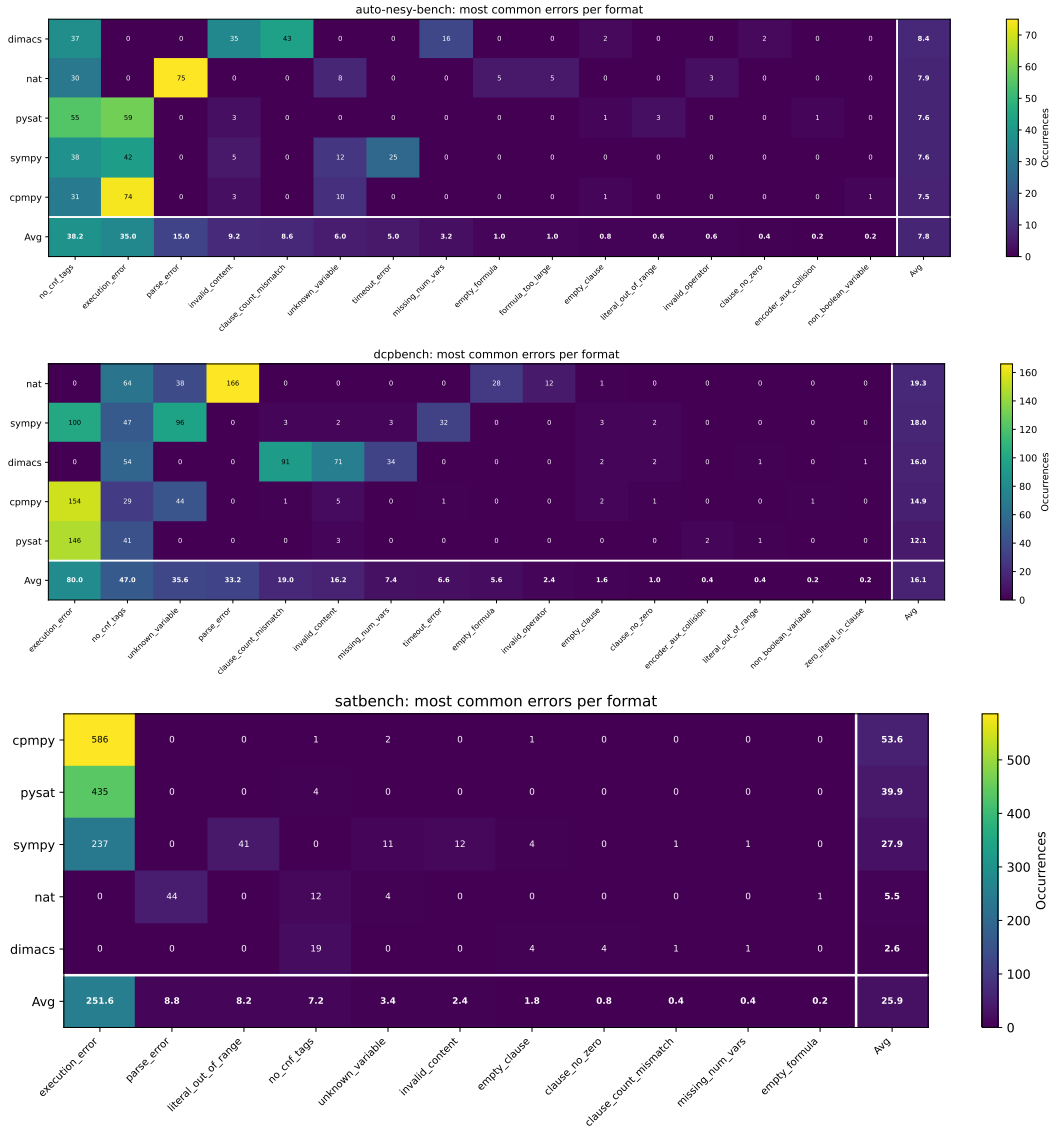


Figure 14: **Output formats exhibit distinct syntactic errors.** Occurrences of each error code in the final model outputs on `auto-nesy-bench` (top), `dcpbench` (middle), and `satbench` (bottom), aggregated over models and prompting strategies. A single output can contribute to multiple error codes. The rightmost column averages over error types, and the bottom row averages over formats.

The format-level analysis in Fig. 14 shows that the types of errors vary substantially across output formats. On auto-nesy-bench, direct DIMACS generation has the highest mean count (8.4 occurrences per error type), despite having the lowest syntax error rate, as a single output can contribute to multiple error codes. Its main errors are clause-count mismatches (43), missing <cnf> tags (37), and invalid content (35). In contrast, NAT mainly fails during parsing (75 errors), followed by missing tags (30). Code-based formats largely avoid these errors but introduce execution failures: PySAT, CPMpy, and SymPy produce 59, 74, and 42 execution errors, respectively. CPMpy has the lowest mean count overall (7.5), while SymPy produces 25 timeouts, suggesting that symbolic conversion can become a computational bottleneck. Missing <cnf> tags occur across all formats, ranging from 30 to 55 instances, indicating a general instruction-following issue. On dcpbench, PySAT has the lowest average number of errors (12.1), while DIMACS has a higher average of 16.0. Clause-count mismatches (91) and invalid content (71) are still common errors. The NAT format has the highest mean count (19.3), including 166 parsing errors, while PySAT and CPMpy produce 146 and 154 execution errors, respectively. SymPy averages 18.0, including 100 execution errors, 96 unknown-variable errors, and 32 timeouts. On satbench, DIMACS has the lowest mean count (2.6), followed by NAT (5.5). Execution errors dominate the code-based formats, with 586 for CPMpy, 435 for PySAT, and 237 for SymPy; all are produced by qwen3-coder-next except for one CPMpy execution error from gemma4-31b. Overall, the types of errors depend on the output format. Higher-level formats reduce manual encoding errors but can cause more errors considering code execution, symbolic conversion, and computational cost.

I.2 Downstream Results of LLM-Generated Formulas on NeSy Predictors

In the following tables, we report the full evaluation results for the ground-truth formulas and the formulas generated by `gpt-oss-120b`, `olmo3-32b-think`, and `qwen3-8b`, as discussed in Section 5.2.

I.2.1 Downstream Results with Detailed Prompts on SPL

Table 20: **SPL full results with `olmo3-32b-think` formulas using `CPMpy`, CoT and detailed prompt**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds.

Task	Consistency (↑)	Accuracy (↑)	Precision (↑)	Recall (↑)	F1 (↑)	Rel	MC-R
bdd-oia-2	1.00 ± 0.00	0.76 ± 0.00	0.75 ± 0.00	0.75 ± 0.00	0.75 ± 0.00	=	1.00
cebab	1.00 ± 0.00	0.63 ± 0.01	0.48 ± 0.01	0.45 ± 0.01	0.46 ± 0.01	=	1.00
cifar10	1.00 ± 0.00	0.94 ± 0.00	0.81 ± 0.01	0.81 ± 0.01	0.81 ± 0.01	=	1.00
cifar100	1.00 ± 0.00	0.99 ± 0.00	0.25 ± 0.01	0.31 ± 0.01	0.26 ± 0.01	=	1.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	=	1.00
fashion	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.01	0.91 ± 0.01	=	1.00
kand-logic-2	1.00 ± 0.00	0.84 ± 0.01	0.77 ± 0.01	0.77 ± 0.01	0.77 ± 0.01	=	1.00
mn-add	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-mul-bin	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-mul	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
sushi	1.00 ± 0.00	0.72 ± 0.01	0.36 ± 0.02	0.31 ± 0.01	0.30 ± 0.01	=	1.00
warcraft	1.00 ± 0.00	0.94 ± 0.00	0.87 ± 0.01	0.86 ± 0.02	0.86 ± 0.01	⇐	1.75

Table 21: **SPL full results with `qwen3-8b` formulas using `CPMpy`, CoT and detailed prompt**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds.

Task	Consistency (↑)	Accuracy (↑)	Precision (↑)	Recall (↑)	F1 (↑)	Rel	MC-R
bdd-oia	1.00 ± 0.00	0.78 ± 0.00	0.58 ± 0.01	0.58 ± 0.01	0.58 ± 0.01	=	1.00
cifar10	1.00 ± 0.00	0.94 ± 0.00	0.82 ± 0.01	0.81 ± 0.01	0.81 ± 0.01	=	1.00
cifar100	1.00 ± 0.00	0.99 ± 0.00	0.25 ± 0.00	0.32 ± 0.00	0.27 ± 0.00	=	1.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	=	1.00
fashion	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	0.92 ± 0.01	=	1.00
kand-logic-2	1.00 ± 0.00	0.80 ± 0.00	0.71 ± 0.00	0.70 ± 0.00	0.70 ± 0.00	⇒	0.02
mn-add-bin	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-add	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-mul	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
sudoku	1.00 ± 0.00	0.92 ± 0.03	0.85 ± 0.05	0.84 ± 0.05	0.84 ± 0.05	=	1.00
sushi	1.00 ± 0.00	0.72 ± 0.00	0.36 ± 0.02	0.31 ± 0.01	0.30 ± 0.01	=	1.00

Table 22: **SPL full results with `gpt-oss-120b` formulas using `CPMpy`, CoT and detailed prompt**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds.

Task	Consistency (↑)	Accuracy (↑)	Precision (↑)	Recall (↑)	F1 (↑)	Rel	MC-R
bdd-oia	1.00 ± 0.00	0.78 ± 0.00	0.60 ± 0.01	0.58 ± 0.01	0.59 ± 0.01	=	1.00
bdd-oia-2	1.00 ± 0.00	0.76 ± 0.00	0.75 ± 0.00	0.75 ± 0.01	0.75 ± 0.00	=	1.00
cebab	1.00 ± 0.00	0.64 ± 0.01	0.48 ± 0.01	0.47 ± 0.02	0.47 ± 0.01	=	1.00
chx	1.00 ± 0.00	0.64 ± 0.02	0.59 ± 0.07	0.47 ± 0.02	0.48 ± 0.02	=	1.00
cifar10	1.00 ± 0.00	0.94 ± 0.00	0.81 ± 0.01	0.80 ± 0.01	0.80 ± 0.01	=	1.00
cifar100	1.00 ± 0.00	0.99 ± 0.00	0.25 ± 0.00	0.32 ± 0.01	0.27 ± 0.00	=	1.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	=	1.00
fashion	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.01	0.92 ± 0.01	0.92 ± 0.01	=	1.00
kand-logic	1.00 ± 0.00	0.75 ± 0.00	0.62 ± 0.00	0.62 ± 0.00	0.62 ± 0.00	=	1.00
kand-logic-2	1.00 ± 0.00	0.84 ± 0.00	0.76 ± 0.01	0.76 ± 0.01	0.76 ± 0.01	=	1.00
mn-add-bin	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-add	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-mul-bin	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-mul	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
road-r	1.00 ± 0.00	0.92 ± 0.00	0.26 ± 0.02	0.23 ± 0.01	0.22 ± 0.01	‖	1.03
sudoku	1.00 ± 0.00	0.89 ± 0.05	0.78 ± 0.10	0.78 ± 0.11	0.77 ± 0.11	=	1.00
sushi	1.00 ± 0.00	0.72 ± 0.00	0.32 ± 0.03	0.30 ± 0.01	0.27 ± 0.01	=	1.00
warcraft	1.00 ± 0.00	0.93 ± 0.01	0.84 ± 0.02	0.84 ± 0.02	0.84 ± 0.02	⇐	1.75

Table 23: **SPL full results with ground-truth formulas using CPMpy, CoT and detailed prompt**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds. As CON and MC-R are defined against ground-truth formulas, they are left empty.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)	REL	MC-R
bdd-oia	–	0.78 ± 0.00	0.59 ± 0.00	0.57 ± 0.01	0.58 ± 0.00	–	–
bdd-oia-2	–	0.75 ± 0.00	0.75 ± 0.00	0.74 ± 0.00	0.74 ± 0.00	–	–
cebab	–	0.64 ± 0.00	0.48 ± 0.02	0.46 ± 0.02	0.47 ± 0.01	–	–
chx	–	0.68 ± 0.01	0.61 ± 0.10	0.52 ± 0.04	0.54 ± 0.05	–	–
cifar10	–	0.96 ± 0.00	0.88 ± 0.00	0.87 ± 0.01	0.87 ± 0.01	–	–
cifar100	–	0.99 ± 0.00	0.32 ± 0.00	0.41 ± 0.00	0.35 ± 0.00	–	–
cle4evr	–	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	–	–
fashion	–	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	–	–
kand-logic	–	0.80 ± 0.00	0.71 ± 0.01	0.70 ± 0.01	0.70 ± 0.01	–	–
kand-logic-2	–	0.87 ± 0.00	0.81 ± 0.01	0.81 ± 0.01	0.81 ± 0.01	–	–
mn-add-bin	–	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	–	–
mn-add	–	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	–	–
mn-mul-bin	–	0.99 ± 0.00	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	–	–
mn-mul	–	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	–	–
road-r	–	0.94 ± 0.00	0.39 ± 0.02	0.30 ± 0.01	0.31 ± 0.01	–	–
sudoku	–	0.90 ± 0.04	0.80 ± 0.07	0.79 ± 0.08	0.79 ± 0.08	–	–
sushi	–	0.70 ± 0.00	0.31 ± 0.01	0.30 ± 0.01	0.29 ± 0.01	–	–
warcraft	–	0.94 ± 0.00	0.87 ± 0.01	0.86 ± 0.01	0.87 ± 0.01	–	–

Failed attempts. Table 24 accounts for missing results in Table 4 and Table 30, all under the CPMpy +CoT configuration used throughout the SPL downstream evaluation. We distinguish three categories. *Formula degenerate* means a CNF was produced and compiles, but is unusable: either it is unsatisfiable (0 models, confirmed directly via model counting), or it declares variables the encoded constraint never actually uses, e.g., olmo3-32b-think/mn-add-bin formula, for instance, omits 4 of its 13 declared variables entirely, i.e., (sum_bit1-sum_bit4). *No formula produced* means the auto-formalization run for that (model, task, format, prompt) combination never emitted a usable CNF to begin with. *Run failure* means that a valid formula exist, but the run fails due to SDD compilation. Five of the thirteen errors are caused by incorrect formulas: olmo3-32b-think/bdd-oia, olmo3-32b-think/mn-add-bin, qwen3-8b/kand-logic, qwen3-8b/chx, and qwen3-8b/road-r.

Table 24: **Reasons behind the missing results in Table 4, CPMpy + CoT.**

MODEL	TASK	CATEGORY	REASON (CPMPY+COT)
olmo3-32b-think	bdd-oia	formula degenerate	UNSAT (0 models)
olmo3-32b-think	kand-logic	no formula produced	no CNF generated
olmo3-32b-think	sudoku	no formula produced	no CNF generated
olmo3-32b-think	chx	run failure	CUDA device-side assert
olmo3-32b-think	road-r	run failure	process stalled
olmo3-32b-think	mn-add-bin	**formula degenerate**	**4 of 13 declared variables are missing**
qwen3-8b	kand-logic	formula degenerate	UNSAT (0 models)
qwen3-8b	chx	formula degenerate	formula omits 1 of 9 oracle concept variables
qwen3-8b	bdd-oia-2	no formula produced	no CNF generated
qwen3-8b	cebab	no formula produced	no CNF generated
qwen3-8b	warcraft	run failure	oracle/formula output-shape mismatch
qwen3-8b	mn-mul-bin	run failure	SDD compilation error
qwen3-8b	road-r	**formula degenerate**	UNSAT (0 models)

I.2.2 WHAT HAPPENS WHEN SPL IS EQUIPPED WITH LLM-GENERATED FORMULAS FROM LESS DETAILED PROMPTS?

To study this ablation, we run the same experiment as for SPL in Section 5.2, but use a less fine-grained prompt description. This is possible because auto-nesy-bench is the *only* auto-formalization benchmark that supports this setting. Below, we compare the ground-truth model (which is the same as in Section 5.2) with the models trained under this setting.

Discussion. Table 25 shows the SPL results with CPMpy output using CoT and non-detailed prompt. In this setting, LLM-generated formulas are still usable in most cases, but they are much less likely to be exactly correct than when using detailed prompts (Table 4). Only about half of the usable formulas are equal to the ground-truth, i.e., REL is $= (9/16$ for gpt-oss-120b, $7/12$ for olmo3-32b-think, and $6/12$ for qwen3-8b), whereas the remaining ones are mostly

incomparable (‖) or strict (⇒). As a result, the generated formulas are farther from the ground-truth formula. The average ΔF1 and Δ recall are -0.12 and -0.10 for gpt-oss-120b, -0.15 and -0.14 for olmo3-32b-think, and -0.14 and -0.12 for qwen3-8b. With detailed prompts, these differences are less than 0.03 and 0.02, respectively. This gap is mainly caused by non-equivalent formulas, which reduce F1 by 0.19–0.28 on average. As with detailed prompts, overly strict formulas are the most harmful. On kand-logic-2, the formulas produced by gpt-oss-120b and olmo3-32b-think capture only 2% of the valid assignments (MC-R $= 0.02$), reducing F1 by 0.36 and 0.26, respectively. incomparable formulas can be just as harmful. For example, gpt-oss-120b on cifar10 loses 0.55 F1, while olmo3-32b-think loses 0.54 on chx and 0.43 on mn-add-bin. In contrast, over-approximations (⇐) have little effect. On cifar10, the formulas produced by olmo3-32b-think and qwen3-8b allow about $19\times$ more assignments than the ground truth, yet reduce F1 by only 0.06-0.07. This suggests that the model can learn to down-weight these spurious output combinations during training. equal formulas perform almost as well as the ground truth on nearly all tasks. The main exception is sudoku, where F1 drops by 0.23-0.49 even though the formula is equivalent to the ground truth. Unlike with detailed prompts, predictions are no longer always consistent with the ground-truth constraints. incomparable and unknown formulas can allow assignments that the ground truth rejects, which lowers SPL consistency. For example, on road-r, consistency drops by 0.12, 0.18, and 1.00 for gpt-oss-120b, olmo3-32b-think, and qwen3-8b, respectively. On kand-logic, it drops by 0.16 for qwen3-8b. In contrast, strict and permissive formulas never produce inconsistent predictions. Failures are also more common for the strongest model: gpt-oss-120b fails on $2/18$ tasks, compared with $0/18$ with detailed prompts. olmo3-32b-think and qwen3-8b each fail on $6/18$ tasks. Most failures are caused by degenerate formulas that leave some task variables unconstrained or are unsatisfiable (see Table 29). Overall, these results show that the task specification matters as much as the choice of LLM. Detailed prompts are important for obtaining correct formulas and, consequently, SPL performance close to that achieved with expert-provided knowledge.

Table 25: **With less-detailed prompts, generated formulas still match expert-provided ones on most tasks, but non-equivalent formulas degrade downstream SPL performance.** Formulas are generated with CPMpy + CoT from the non-detailed prompt. We report mean ΔF1/Δ Recall (REC)/Δ consistency (CONS) relative to the ground-truth (G-T) formula; standard deviations across 5 seeds are omitted (see below for the full results). REL denotes the relation between predicted and ground-truth formulas: equal (=), strict (⇒), permissive (⇐), or incomparable (‖), indicating whether the predicted formula satisfying assignments are equal to, a subset of, a superset of, or neither; unknown marks the two kand-logic formulas, for which the exact comparison, which is based on model counting, did not terminate. MC-R measures the ratio of satisfying assignments of the predicted formula to that of the ground-truth formula. Consistency drops below that of G-T only for incomparable or unknown formulas, which allow predictions the ground truth rejects. Interestingly, this does not happen for strict or permissive formulas. Missing cells indicate that the formula is either missing or was not executable (see Table 29 for details).

	G-T			gpt-oss-120b					olmo3-32b-think				
TASK	F1 (↑)	REC (↑)	CONS (↑)	ΔF1	ΔREC	ΔCONS	REL	MC-R	ΔF1	ΔREC	ΔCONS	REL	MC-R
bdd-oia	0.58	0.57	1.00	–	–	–	–	–	–	–	–	–	–
bdd-oia-2	0.74	0.74	1.00	−0.08	−0.08	+0.00	‖	1.00	–	–	–	–	–
cebab	0.47	0.46	1.00	–	–	–	–	–	–	–	–	–	–
chx	0.54	0.52	1.00	−0.27	−0.16	+0.00	⇒	0.94	−0.54	−0.52	−0.35	‖	1.00
cifar10	0.87	0.87	1.00	−0.55	−0.47	+0.00	‖	3.38	−0.07	−0.07	+0.00	⇐	19.25
cifar100	0.35	0.41	1.00	−0.08	−0.09	+0.00	=	1.00	−0.09	−0.10	+0.00	=	1.00
cle4evr	1.00	1.00	1.00	−0.00	−0.00	+0.00	=	1.00	−0.01	−0.01	+0.00	=	1.00
fashion	0.92	0.92	1.00	−0.00	+0.00	+0.00	=	1.00	−0.00	−0.00	+0.00	=	1.00
kand-logic	0.70	0.70	1.00	−0.09	−0.09	+0.00	unknown	–	–	–	–	–	–
kand-logic-2	0.81	0.81	1.00	−0.36	−0.32	+0.00	⇒	0.02	−0.26	−0.26	+0.00	⇒	0.02
mn-add-bin	0.98	0.98	1.00	−0.00	−0.00	+0.00	=	1.00	−0.43	−0.40	−0.01	‖	2.56
mn-add	0.98	0.98	1.00	−0.00	−0.00	+0.00	=	1.00	−0.00	−0.00	+0.00	=	1.00
mn-mul-bin	0.98	0.98	1.00	−0.00	−0.01	+0.00	=	1.00	–	–	–	–	–
mn-mul	0.98	0.98	1.00	−0.00	−0.00	+0.00	=	1.00	−0.00	−0.00	+0.00	=	1.00
road-r	0.31	0.30	1.00	−0.09	−0.07	−0.12	‖	76.64	−0.09	−0.07	−0.18	‖	242.22
sudoku	0.79	0.79	1.00	−0.36	−0.35	+0.00	=	1.00	−0.23	−0.23	+0.00	=	1.00
sushi	0.29	0.30	1.00	−0.02	+0.01	+0.00	=	1.00	−0.00	+0.01	+0.00	=	1.00
warcraft	0.87	0.86	1.00	−0.03	−0.03	+0.00	⇐	1.75	–	–	–	–	–

	G-T			qwen3-8b				
TASK	F1 (↑)	REC (↑)	CONS (↑)	ΔF1	ΔREC	ΔCONS	REL	MC-R
bdd-oia	0.58	0.57	1.00	–	–	–	–	–
bdd-oia-2	0.74	0.74	1.00	–	–	–	–	–
cebab	0.47	0.46	1.00	–	–	–	–	–
chx	0.54	0.52	1.00	–	–	–	–	–
cifar10	0.87	0.87	1.00	−0.06	−0.07	+0.00	⇐	19.62
cifar100	0.35	0.41	1.00	−0.08	−0.09	+0.00	=	1.00
cle4evr	1.00	1.00	1.00	−0.00	−0.00	+0.00	=	1.00
fashion	0.92	0.92	1.00	−0.00	−0.00	+0.00	=	1.00
kand-logic	0.70	0.70	1.00	−0.29	−0.26	−0.16	unknown	–
kand-logic-2	0.81	0.81	1.00	−0.24	−0.23	−0.15	‖	0.19
mn-add-bin	0.98	0.98	1.00	+0.00	+0.00	−0.00	⇐	2.56
mn-add	0.98	0.98	1.00	−0.00	−0.00	+0.00	=	1.00
mn-mul-bin	0.98	0.98	1.00	−0.24	−0.20	−0.01	‖	2.56
mn-mul	0.98	0.98	1.00	–	–	–	–	–
road-r	0.31	0.30	1.00	−0.28	−0.15	−1.00	‖	7.81
sudoku	0.79	0.79	1.00	−0.49	−0.47	+0.00	=	1.00
sushi	0.29	0.30	1.00	+0.01	+0.01	+0.00	=	1.00
warcraft	0.87	0.86	1.00	–	–	–	–	–

Table 26: **SPL results for non detailed prompt using `gpt-oss-120b` formulas**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)	REL	MC-R
bdd-oia-2	1.00 ± 0.00	0.70 ± 0.01	0.71 ± 0.00	0.66 ± 0.02	0.67 ± 0.02	‖	1.00
chx	1.00 ± 0.00	0.42 ± 0.03	0.27 ± 0.07	0.36 ± 0.01	0.26 ± 0.03	⇒	0.94
cifar10	1.00 ± 0.00	0.82 ± 0.00	0.29 ± 0.01	0.40 ± 0.01	0.32 ± 0.01	‖	3.38
cifar100	1.00 ± 0.00	0.99 ± 0.00	0.25 ± 0.00	0.32 ± 0.01	0.27 ± 0.01	=	1.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	=	1.00
fashion	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.01	0.92 ± 0.01	0.92 ± 0.01	=	1.00
kand-logic	1.00 ± 0.00	0.74 ± 0.00	0.61 ± 0.00	0.61 ± 0.00	0.61 ± 0.00	unknown	–
kand-logic-2	1.00 ± 0.00	0.66 ± 0.06	0.45 ± 0.19	0.49 ± 0.09	0.45 ± 0.16	⇒	0.02
mn-add-bin	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-add	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-mul-bin	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-mul	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
road-r	0.88 ± 0.03	0.93 ± 0.00	0.28 ± 0.02	0.23 ± 0.01	0.22 ± 0.01	‖	76.64
sudoku	1.00 ± 0.00	0.72 ± 0.04	0.45 ± 0.07	0.44 ± 0.07	0.43 ± 0.07	=	1.00
sushi	1.00 ± 0.00	0.72 ± 0.00	0.32 ± 0.04	0.30 ± 0.01	0.27 ± 0.01	=	1.00
warcraft	1.00 ± 0.00	0.93 ± 0.01	0.84 ± 0.02	0.84 ± 0.01	0.83 ± 0.01	⇐	1.75

Table 27: **SPL results for non detailed prompt using `olmo-32b-think`** formulas. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)	REL	MC-R
chx	0.65 ± 0.08	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	‖	1.00
cifar10	1.00 ± 0.00	0.94 ± 0.00	0.81 ± 0.01	0.80 ± 0.01	0.80 ± 0.01	⇐	19.25
cifar100	1.00 ± 0.00	0.99 ± 0.00	0.25 ± 0.01	0.31 ± 0.01	0.26 ± 0.01	=	1.00
cle4evr	1.00 ± 0.00	0.99 ± 0.01	0.99 ± 0.02	0.99 ± 0.02	0.99 ± 0.02	=	1.00
fashion	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.01	0.91 ± 0.01	=	1.00
kand-logic-2	1.00 ± 0.00	0.70 ± 0.00	0.56 ± 0.01	0.55 ± 0.00	0.54 ± 0.00	⇒	0.02
mn-add-bin	0.99 ± 0.00	0.53 ± 0.00	0.65 ± 0.02	0.58 ± 0.00	0.55 ± 0.01	‖	2.56
mn-add	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-mul	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
road-r	0.82 ± 0.03	0.93 ± 0.00	0.25 ± 0.01	0.23 ± 0.01	0.21 ± 0.01	‖	242.22
sudoku	1.00 ± 0.00	0.78 ± 0.02	0.58 ± 0.03	0.56 ± 0.04	0.56 ± 0.04	=	1.00
sushi	1.00 ± 0.00	0.71 ± 0.00	0.35 ± 0.03	0.31 ± 0.01	0.29 ± 0.02	=	1.00

Table 28: **SPL results for non detailed prompt using `qwen3-8b`** formulas. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)	REL	MC-R
cifar10	1.00 ± 0.00	0.94 ± 0.00	0.82 ± 0.00	0.81 ± 0.01	0.81 ± 0.01	⇐	19.62
cifar100	1.00 ± 0.00	0.99 ± 0.00	0.25 ± 0.00	0.32 ± 0.01	0.27 ± 0.00	=	1.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	=	1.00
fashion	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	0.91 ± 0.00	=	1.00
kand-logic	0.84 ± 0.07	0.62 ± 0.00	0.47 ± 0.01	0.44 ± 0.00	0.41 ± 0.01	unknown	–
kand-logic-2	0.85 ± 0.04	0.72 ± 0.01	0.61 ± 0.01	0.58 ± 0.02	0.57 ± 0.02	‖	0.19
mn-add-bin	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	⇐	2.56
mn-add	1.00 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-mul-bin	0.99 ± 0.01	0.85 ± 0.00	0.72 ± 0.01	0.79 ± 0.00	0.74 ± 0.01	‖	2.56
road-r	0.00 ± 0.00	0.80 ± 0.01	0.02 ± 0.01	0.15 ± 0.01	0.03 ± 0.00	‖	7.81
sudoku	1.00 ± 0.00	0.66 ± 0.02	0.32 ± 0.04	0.32 ± 0.03	0.30 ± 0.04	=	1.00
sushi	1.00 ± 0.00	0.72 ± 0.01	0.36 ± 0.02	0.31 ± 0.01	0.30 ± 0.01	=	1.00

Failed attempts (non-detailed prompts). Table 29 accounts for the missing results in Table 25, using the same `CPMpy` +CoT configuration but with non-detailed prompts. We use the same three categories as for detailed prompts (Table 24). Most errors are now *formula-degenerate* (10 out of 14). In `bdd-oia`, all three models omit part of the scene description: `gpt-oss-120b` never uses 5 of the 25 variables (`follow`, `left_follow`, `no_left_lane`, `right_follow`, `no_right_lane`), while `olmo3-32b-think` and `qwen3-8b` omit 6. On `bdd-oia-2`, both smaller models also drop the `follow` concept. The `qwen3-8b` formula for `mn-mul` omits 45 of its 102 variables, corresponding exactly to the product classes that no pair of digits can produce (e.g., `p_11`, `p_13`, and `p_17`). Rather than explicitly constraining these variables to false, the formula leaves them unconstrained. Two formulas are unsatisfiable: `olmo3-32b-think`/`warcraft` and `qwen3-8b`/`cebab`. Finally, the `qwen3-8b` formula for `chx` admits only a single assignment,

leaving no parameters in the compiled circuit for SPL to learn. The remaining four gaps are *no formula produced*: cebab for gpt-oss-120b and olmo3-32b-think, kand-logic and mn-mul-bin for olmo3-32b-think, and warcraft for qwen3-8b. Unlike with detailed prompts, there are no *run failures*: every usable formula is trained successfully. With detailed prompts, gpt-oss-120b produced a usable formula for every task. Without them, even the strongest model fails on $2/18$ tasks, and failures for all models are more often caused by incorrect formulas than by pipeline issues. This suggests that, without a detailed specification, LLMs tend to under-specify the constraints rather than fail to produce a formula altogether.

Table 29: **Missing results in Table 25, CPMpy + CoT with non-detailed prompts.**

MODEL	TASK	CATEGORY	REASON (CPMPY+COT, NON-DETAILED PROMPT)
gpt-oss-120b	bdd-oia	formula degenerate	5 of 25 task variables are missing
gpt-oss-120b	cebab	no formula produced	no CNF generated
olmo3-32b-think	bdd-oia	formula degenerate	6 of 25 task variables are missing
olmo3-32b-think	bdd-oia-2	formula degenerate	1 of 11 task variables is missing
olmo3-32b-think	warcraft	formula degenerate	UNSAT (0 models)
olmo3-32b-think	cebab	no formula produced	no CNF generated
olmo3-32b-think	kand-logic	no formula produced	no CNF generated
olmo3-32b-think	mn-mul-bin	no formula produced	no CNF generated
qwen3-8b	bdd-oia	formula degenerate	6 of 25 task variables are missing
qwen3-8b	bdd-oia-2	formula degenerate	1 of 11 task variables is missing
qwen3-8b	cebab	formula degenerate	UNSAT (0 models)
qwen3-8b	mn-mul	formula degenerate	45 of 102 task variables are missing
qwen3-8b	chx	formula degenerate	no trainable circuit parameters
qwen3-8b	warcraft	no formula produced	no CNF generated

I.2.3 WHAT IF THE CONSTRAINTS ARE SOFT? SEMANTIC LOSS DOWNSTREAM RESULTS WITH DETAILED PROMPTS

In addition to the SPL results, we further analyze the downstream performance under a different NeSy approach based on the *semantic loss* (SL) (Xu et al., 2018), which incorporates logical constraints as a penalty loss rather than explicitly enforcing them during prediction. Specifically, we replicate the same SPL analysis (CPMpy +CoT and detailed prompt in Section 5.2) using the SL.

Semantic Loss. Given an input $\boldsymbol{x}$ and an unconstrained predictive distribution $p_\theta(\boldsymbol{y} \mid \boldsymbol{x})$, SL (Xu et al., 2018) incorporates a logical constraint K by maximizing the probability mass assigned to its satisfying assignments. Assuming conditional independence of the variables given the model outputs, it is defined as

$$\mathcal{L}_{\text{SL}}(\mathsf{K}, p_\theta) = -\log \sum_{\boldsymbol{y} \models \mathsf{K}} \prod_{i=1}^{n} p_\theta(Y_i \mid \boldsymbol{x}), \tag{16}$$

where n is the number of primary Boolean output variables. For a Boolean variable, $p_\theta(Y_i \mid \boldsymbol{x})$ is a shorthand for $p_\theta(Y_i = 1 \mid \boldsymbol{x})$ if $y_i = 1$ and $1 - p_\theta(Y_i = 1 \mid \boldsymbol{x})$ otherwise. Thus, the sum computes the total probability assigned to all satisfying assignments, and the negative logarithm penalizes the model when this probability is low. The sum in Eq. (16) is exactly the *weighted model count* (WMC) (Chavira & Darwiche, 2008) of K under the weights induced by $p_\theta(\boldsymbol{y} \mid \boldsymbol{x})$.

Implementation. For these experiments, we reuse the same SDDs compiled for the SPL experiments. At training time, the SDD is loaded using KLay (Maene et al., 2025), which makes the WMC, and hence of $\mathcal{L}_{\text{SL}}$, faster. Unlike SPL, which explicitly incorporates logical constraints into the prediction mechanism, SL treats K as a *soft constraint* through an additional differentiable loss term. In our setting, it is combined with the downstream task objective as

$$\mathcal{L}_{\text{total}}(\mathsf{K}, p_\theta) = \mathcal{L}_{\text{task}}(p_\theta) + \lambda \mathcal{L}_{\text{SL}}(\mathsf{K}, p_\theta), \tag{17}$$

where λ controls the strength of the logical constraint. As all the benchmarks in auto-nesy-bench are classification tasks, $\mathcal{L}_{\text{task}}$ corresponds to the negative log-likelihood, while for all SL experiments λ is set to 1.

Architectures. SL uses the same per-task backbones as SPL. We only replace the SPL head, which outputs the parameters of the circuit, with a classification head: an MLP with three hidden layers with ReLU activations, whose width is set to the SPL head width tuned for that task, and one

output per variable of the CNF. Each output is passed through a sigmoid, so that p_θ factorizes into independent Bernoulli distributions over the variables; this is the distribution on which $\mathcal{L}_{\mathrm{SL}}$ computes the probability of satisfying K, with the positive and negative literal weights given by $\log p_\theta$ and $\log(1 - p_\theta)$. At test time, each variable is predicted by thresholding its probability at 0.5; for tasks with a single multi-class output, we predict the class whose output-bit pattern is most likely.

Discussion. Table 30 reports the `SL` experiments in the same setting as Section 5.2, i.e., using the formulas generated by the LLMs with detailed prompts, chain-of-thought, and `CPMpy` as output format. Without `SL`, the unconstrained network is often inconsistent with the ground-truth formula: its consistency is below 0.70 on seven of the 18 tasks, ranging from 0.06 on `sushi` and 0.11 on `sudoku` to 0.86 on `warcraft`, whereas simpler tasks such as `cle4evr`, `fashion`, and the `mn` datasets are already almost fully consistent (≥ 0.98). Adding the `SL` with an equivalent formula ($=$) never decreases consistency and increases it on all tasks that leave room for improvement, most notably on `sushi`, where it rises from 0.06 to 0.88-0.97, and on `road-r` and `cebab` ($+0.11$ to $+0.17$). Unlike `SPL` (see Section 5.2), however, whose predictions satisfy the formula by construction, `SL` only encourages consistency during training, and it rarely reaches 1: on the hardest tasks it remains far from it (`kand-logic` 0.21-0.22, `sudoku` 0.23-0.42). Downstream performance, for `equal` ($=$) formulas, is almost unaffected by the semantic loss (mean ΔF1 between -0.00 and $+0.02$ across LLMs), with gains concentrated on the most constrained tasks (`sudoku`, $+0.11$ to $+0.15$ F1; `mn-mul-bin`, up to $+0.08$). The only formula that reduces consistency is the one generated by `qwen3-8b` for `kand-logic-2` (-0.23 `Cons`, -0.05 F1). This formula is too `strict` ($\Rightarrow$): it admits only 2% of the assignments allowed by the ground truth (MC-R $= 0.02$), so it rules out most of the labels seen during training. Interestingly, as for `SPL`, `permissive` formulas ($\Leftarrow$) are much less harmful: on `warcraft`, the formulas of `gpt-oss-120b` and `olmo3-32b-think` still improve consistency by $+0.08$ and $+0.01$, respectively. Probably the classification loss also contributes by assigning low probability mass on invalid combinations.

Table 30: **Downstream `SL` performance using generated formulas, when available, is comparable to that of expert-provided formulas on `auto-nesy-bench`. Semantic loss mainly improves consistency in almost all cases compared with an unconstrained baseline.** The top table reports `SL` with the ground-truth (G-T) and `gpt-oss-120b` formulas, the bottom one with the `olmo3-32b-think` and `qwen3-8b` formulas against the unconstrained baseline. In both, the first group (Unconstrained) is the same backbone and classifier trained without the semantic loss, for which we report F1, Recall (REC) and consistency (CONS, the fraction of test predictions that satisfy the ground-truth formula); every other group reports ΔF1/ΔREC/ΔCONS of `SL` relative to Unconstrained (positive is better). Standard deviations across 5 seeds are omitted (≤ 0.02 for F1 and REC on all tasks except `sudoku`, `warcraft`, `mn-mul-bin` and `chx`); the full results, with standard deviations and all metrics, are attached below for reference. REL denotes the relationship between the predicted and ground-truth formulas: `equal` (=), `strict` (⇒), `permissive` (⇐), or `incomparable` (‖), indicating whether the predicted formula satisfying assignments are equal to, a subset of, a superset of, or neither relative to the ground truth. MC-R measures the ratio of the number of satisfying assignments of the predicted formula to that of the ground-truth formula. Missing cells indicate the errors in the formula generation, formatting or during inference. Details about the missing results is provided in Table 24.

	UNCONSTRAINED			G-T					`gpt-oss-120b`				
TASK	F1 (↑)	REC (↑)	CONS (↑)	ΔF1	ΔREC	ΔCONS	REL	MC-R	ΔF1	ΔREC	ΔCONS	REL	MC-R
bdd-oia	0.58	0.58	0.90	−0.00	+0.01	+0.02	=	1.00	+0.01	+0.02	+0.01	=	1.00
bdd-oia-2	0.73	0.71	0.92	−0.00	+0.00	+0.02	=	1.00	+0.00	+0.01	+0.00	=	1.00
cebab	0.46	0.46	0.63	+0.00	+0.00	+0.13	=	1.00	+0.01	+0.01	+0.17	=	1.00
chx	0.57	0.53	0.91	−0.00	+0.00	+0.07	=	1.00	−0.02	−0.02	+0.05	=	1.00
cifar10	0.87	0.86	0.94	−0.02	−0.01	+0.05	=	1.00	−0.02	−0.01	+0.05	=	1.00
cifar100	0.52	0.45	0.63	+0.01	+0.03	+0.07	=	1.00	+0.01	+0.02	+0.09	=	1.00
cle4evr	1.00	1.00	1.00	+0.00	+0.00	+0.00	=	1.00	+0.00	+0.00	+0.00	=	1.00
fashion	0.92	0.92	0.99	+0.00	+0.00	+0.00	=	1.00	+0.00	+0.00	+0.00	=	1.00
kand-logic	0.72	0.71	0.19	+0.00	+0.00	+0.02	=	1.00	+0.01	+0.02	+0.02	=	1.00
kand-logic-2	0.79	0.80	0.66	+0.00	+0.00	+0.04	=	1.00	+0.00	+0.00	+0.08	=	1.00
mn-add-bin	0.98	0.98	0.98	+0.00	+0.00	+0.01	=	1.00	+0.00	+0.00	+0.01	=	1.00
mn-add	0.98	0.98	0.99	+0.00	+0.00	+0.00	=	1.00	−0.01	−0.01	−0.00	=	1.00
mn-mul-bin	0.71	0.71	0.99	+0.03	+0.03	+0.00	=	1.00	+0.08	+0.08	+0.00	=	1.00
mn-mul	0.98	0.98	0.99	+0.00	+0.00	+0.00	=	1.00	+0.00	+0.00	+0.01	=	1.00
road-r	0.27	0.25	0.65	+0.01	+0.02	+0.16	=	1.00	+0.02	+0.03	+0.16	‖	1.03
sudoku	0.74	0.72	0.11	+0.11	+0.12	+0.12	=	1.00	+0.11	+0.12	+0.16	=	1.00
sushi	0.16	0.14	0.06	+0.02	+0.11	+0.82	=	1.00	+0.07	+0.13	+0.91	=	1.00
warcraft	0.78	0.78	0.86	−0.01	−0.01	+0.05	=	1.00	+0.07	+0.07	+0.08	⇐	1.75

	UNCONSTRAINED			`olmo3-32b-think`					`qwen3-8b`				
TASK	F1 (↑)	REC (↑)	CONS (↑)	ΔF1	ΔREC	ΔCONS	REL	MC-R	ΔF1	ΔREC	ΔCONS	REL	MC-R
bdd-oia	0.58	0.58	0.90	–	–	–	–	–	+0.01	+0.02	+0.04	=	1.00
bdd-oia-2	0.73	0.71	0.92	+0.00	+0.01	+0.02	=	1.00	–	–	–	–	–
cebab	0.46	0.46	0.63	+0.02	+0.02	+0.11	=	1.00	–	–	–	–	–
chx	0.57	0.53	0.91	–	–	–	–	–	–	–	–	–	–
cifar10	0.87	0.86	0.94	−0.02	−0.01	+0.05	=	1.00	−0.01	+0.00	+0.05	=	1.00
cifar100	0.52	0.45	0.63	+0.00	+0.02	+0.09	=	1.00	+0.02	+0.04	+0.10	=	1.00
cle4evr	1.00	1.00	1.00	−0.00	−0.00	+0.00	=	1.00	−0.00	−0.00	+0.00	=	1.00
fashion	0.92	0.92	0.99	+0.00	+0.00	+0.00	=	1.00	+0.00	+0.00	+0.00	=	1.00
kand-logic	0.72	0.71	0.19	–	–	–	–	–	–	–	–	–	–
kand-logic-2	0.79	0.80	0.66	+0.00	+0.00	+0.04	=	1.00	−0.05	−0.05	−0.23	⇒	0.02
mn-add-bin	0.98	0.98	0.98	–	–	–	–	–	+0.00	+0.00	+0.01	=	1.00
mn-add	0.98	0.98	0.99	−0.00	−0.00	−0.00	=	1.00	−0.00	−0.00	+0.00	=	1.00
mn-mul-bin	0.71	0.71	0.99	+0.07	+0.07	+0.00	=	1.00	–	–	–	–	–
mn-mul	0.98	0.98	0.99	+0.00	+0.00	+0.01	=	1.00	−0.00	−0.00	+0.00	=	1.00
road-r	0.27	0.25	0.65	–	–	–	–	–	–	–	–	–	–
sudoku	0.74	0.72	0.11	–	–	–	–	–	+0.15	+0.16	+0.32	=	1.00
sushi	0.16	0.14	0.06	+0.06	+0.13	+0.91	=	1.00	+0.09	+0.15	+0.89	=	1.00
warcraft	0.78	0.78	0.86	−0.16	−0.14	+0.01	⇐	1.75	–	–	–	–	–

Table 31: **Unconstrained model**. The table reports consistency, accuracy, precision and F1. Mean ± std over 5 seeds.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)
bdd-oia	0.90 ± 0.00	0.78 ± 0.00	0.59 ± 0.01	0.58 ± 0.01	0.58 ± 0.00
bdd-oia-2	0.92 ± 0.00	0.75 ± 0.00	0.76 ± 0.00	0.71 ± 0.01	0.73 ± 0.01
cebab	0.63 ± 0.03	0.64 ± 0.01	0.48 ± 0.01	0.46 ± 0.02	0.46 ± 0.01
chx	0.91 ± 0.01	0.69 ± 0.01	0.68 ± 0.04	0.53 ± 0.02	0.57 ± 0.02
cifar10	0.94 ± 0.00	0.96 ± 0.00	0.89 ± 0.00	0.86 ± 0.00	0.87 ± 0.00
cifar100	0.63 ± 0.00	0.99 ± 0.00	0.66 ± 0.00	0.45 ± 0.01	0.52 ± 0.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
fashion	0.99 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	0.92 ± 0.00
kand-logic	0.19 ± 0.01	0.82 ± 0.00	0.73 ± 0.01	0.71 ± 0.01	0.72 ± 0.01
kand-logic-2	0.66 ± 0.05	0.86 ± 0.01	0.79 ± 0.01	0.80 ± 0.01	0.79 ± 0.01
mn-add-bin	0.98 ± 0.01	0.98 ± 0.00	0.97 ± 0.00	0.98 ± 0.01	0.98 ± 0.00
mn-add	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00
mn-mul-bin	0.99 ± 0.00	0.98 ± 0.00	0.71 ± 0.04	0.71 ± 0.04	0.71 ± 0.04
mn-mul	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00
road-r	0.65 ± 0.03	0.94 ± 0.00	0.37 ± 0.02	0.25 ± 0.01	0.27 ± 0.01
sudoku	0.11 ± 0.10	0.88 ± 0.04	0.77 ± 0.08	0.72 ± 0.11	0.74 ± 0.10
sushi	0.06 ± 0.01	0.76 ± 0.00	0.34 ± 0.04	0.14 ± 0.01	0.16 ± 0.01
warcraft	0.86 ± 0.05	0.91 ± 0.03	0.80 ± 0.06	0.78 ± 0.08	0.78 ± 0.07

Table 32: **`SL` logical consistency and downstream metrics with the GROUND-TRUTH formula**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds. REL and MC-R are trivially =/1.00.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)	REL	MC-R
bdd-oia	0.92 ± 0.00	0.78 ± 0.00	0.59 ± 0.01	0.58 ± 0.01	0.58 ± 0.00	=	1.00
bdd-oia-2	0.94 ± 0.01	0.75 ± 0.01	0.75 ± 0.01	0.71 ± 0.01	0.73 ± 0.01	=	1.00
cebab	0.77 ± 0.02	0.64 ± 0.01	0.48 ± 0.01	0.47 ± 0.02	0.47 ± 0.01	=	1.00
chx	0.97 ± 0.01	0.69 ± 0.01	0.68 ± 0.02	0.54 ± 0.03	0.57 ± 0.03	=	1.00
cifar10	0.99 ± 0.00	0.95 ± 0.00	0.86 ± 0.00	0.84 ± 0.01	0.85 ± 0.00	=	1.00
cifar100	0.70 ± 0.00	0.99 ± 0.00	0.64 ± 0.00	0.47 ± 0.00	0.53 ± 0.00	=	1.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	=	1.00
fashion	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	=	1.00
kand-logic	0.21 ± 0.02	0.82 ± 0.01	0.73 ± 0.01	0.72 ± 0.01	0.72 ± 0.01	=	1.00
kand-logic-2	0.70 ± 0.06	0.87 ± 0.01	0.80 ± 0.01	0.80 ± 0.01	0.80 ± 0.01	=	1.00
mn-add-bin	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-add	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-mul-bin	0.99 ± 0.00	0.98 ± 0.00	0.74 ± 0.04	0.74 ± 0.04	0.74 ± 0.04	=	1.00
mn-mul	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
road-r	0.81 ± 0.04	0.94 ± 0.00	0.36 ± 0.01	0.27 ± 0.01	0.28 ± 0.01	=	1.00
sudoku	0.23 ± 0.10	0.93 ± 0.02	0.87 ± 0.04	0.84 ± 0.05	0.85 ± 0.05	=	1.00
sushi	0.88 ± 0.07	0.72 ± 0.00	0.19 ± 0.05	0.25 ± 0.01	0.18 ± 0.02	=	1.00
warcraft	0.91 ± 0.01	0.90 ± 0.02	0.79 ± 0.06	0.77 ± 0.06	0.77 ± 0.06	=	1.00

Table 33: **`SL` logical consistency and exact LLM-formula-vs-oracle relationship for the `gpt-oss-120b` formulas**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)	REL	MC-R
bdd-oia	0.91 ± 0.01	0.78 ± 0.00	0.59 ± 0.00	0.60 ± 0.01	0.59 ± 0.01	=	1.00
bdd-oia-2	0.92 ± 0.01	0.75 ± 0.00	0.75 ± 0.00	0.72 ± 0.01	0.73 ± 0.00	=	1.00
cebab	0.80 ± 0.02	0.64 ± 0.01	0.48 ± 0.01	0.47 ± 0.02	0.47 ± 0.01	=	1.00
chx	0.96 ± 0.01	0.68 ± 0.01	0.65 ± 0.04	0.52 ± 0.01	0.55 ± 0.01	=	1.00
cifar10	0.99 ± 0.00	0.95 ± 0.00	0.86 ± 0.00	0.85 ± 0.01	0.85 ± 0.01	=	1.00
cifar100	0.72 ± 0.01	0.99 ± 0.00	0.62 ± 0.00	0.47 ± 0.00	0.53 ± 0.00	=	1.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	=	1.00
fashion	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	=	1.00
kand-logic	0.22 ± 0.03	0.82 ± 0.01	0.74 ± 0.01	0.73 ± 0.02	0.73 ± 0.01	=	1.00
kand-logic-2	0.73 ± 0.05	0.86 ± 0.01	0.80 ± 0.02	0.80 ± 0.02	0.80 ± 0.02	=	1.00
mn-add-bin	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-add	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
mn-mul-bin	0.99 ± 0.00	0.98 ± 0.00	0.78 ± 0.04	0.78 ± 0.04	0.78 ± 0.04	=	1.00
mn-mul	0.99 ± 0.00	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	=	1.00
road-r	0.82 ± 0.03	0.94 ± 0.00	0.39 ± 0.03	0.28 ± 0.01	0.29 ± 0.01	‖	1.03
sudoku	0.26 ± 0.09	0.93 ± 0.03	0.86 ± 0.05	0.84 ± 0.06	0.85 ± 0.06	=	1.00
sushi	0.97 ± 0.02	0.72 ± 0.00	0.26 ± 0.04	0.27 ± 0.00	0.23 ± 0.01	=	1.00
warcraft	0.94 ± 0.01	0.94 ± 0.01	0.86 ± 0.01	0.85 ± 0.02	0.85 ± 0.01	⇐	1.75

Table 34: **SL logical consistency and exact LLM-formula-vs-oracle relationship for the `olmo3-32b-think` formulas**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)	REL	MC-R
bdd-oia-2	0.94 ± 0.00	0.75 ± 0.00	0.75 ± 0.00	0.71 ± 0.01	0.73 ± 0.00	$=$	1.00
cebab	0.74 ± 0.02	0.66 ± 0.01	0.49 ± 0.01	0.48 ± 0.02	0.48 ± 0.02	$=$	1.00
cifar10	0.99 ± 0.00	0.95 ± 0.00	0.86 ± 0.01	0.85 ± 0.00	0.85 ± 0.01	$=$	1.00
cifar100	0.72 ± 0.02	0.99 ± 0.00	0.61 ± 0.01	0.47 ± 0.01	0.52 ± 0.01	$=$	1.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	$=$	1.00
fashion	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	$=$	1.00
kand-logic-2	0.70 ± 0.02	0.86 ± 0.00	0.80 ± 0.00	0.80 ± 0.00	0.80 ± 0.00	$=$	1.00
mn-add	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	$=$	1.00
mn-mul-bin	0.99 ± 0.00	0.98 ± 0.00	0.77 ± 0.03	0.77 ± 0.03	0.77 ± 0.03	$=$	1.00
mn-mul	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	$=$	1.00
sushi	0.96 ± 0.02	0.72 ± 0.00	0.29 ± 0.03	0.27 ± 0.00	0.23 ± 0.00	$=$	1.00
warcraft	0.87 ± 0.05	0.86 ± 0.07	0.65 ± 0.22	0.64 ± 0.16	0.62 ± 0.20	$\Leftarrow$	1.75

Table 35: **SL logical consistency and exact LLM-formula-vs-oracle relationship for the `qwen3-8b` formulas**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)	REL	MC-R
bdd-oia	0.94 ± 0.01	0.78 ± 0.00	0.59 ± 0.01	0.60 ± 0.01	0.59 ± 0.00	$=$	1.00
cifar10	0.99 ± 0.00	0.96 ± 0.00	0.87 ± 0.00	0.86 ± 0.00	0.86 ± 0.00	$=$	1.00
cifar100	0.73 ± 0.00	0.99 ± 0.00	0.63 ± 0.00	0.49 ± 0.00	0.54 ± 0.00	$=$	1.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	$=$	1.00
fashion	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	$=$	1.00
kand-logic-2	0.42 ± 0.15	0.83 ± 0.01	0.76 ± 0.01	0.75 ± 0.01	0.75 ± 0.01	$\Rightarrow$	0.02
mn-add-bin	0.99 ± 0.00	0.98 ± 0.00	0.97 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	$=$	1.00
mn-add	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	$=$	1.00
mn-mul	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	$=$	1.00
sudoku	0.42 ± 0.09	0.94 ± 0.02	0.89 ± 0.03	0.88 ± 0.04	0.89 ± 0.04	$=$	1.00
sushi	0.95 ± 0.02	0.72 ± 0.00	0.34 ± 0.03	0.29 ± 0.00	0.26 ± 0.01	$=$	1.00

I.2.4 SOFT CONSTRAINTS WITH SEMANTIC LOSS EQUIPPED WITH LLM-GENERATED FORMULAS FROM LESS DETAILED PROMPTS

As for SPL in Section I.2.2, we replicate the same exact analysis for SL, i.e., CPMpy with CoT and less detailed prompts.

Table 36: **With non-detailed prompts, SL benefits from correct formulas, while non-equivalent constraints degrade logical consistency.** Formulas are generated with CPMpy + CoT from non-detailed prompts; the setup is identical to Table 30. The top table reports SL with the ground-truth (G-T) and gpt-oss-120b formulas, the bottom one with the olmo3-32b-think and qwen3-8b generated formulas. For the UNCONSTRAINED we report F1, Recall (REC) and consistency (CONS, the fraction of predictions that satisfy the ground-truth formula); every other method reports ΔF1/ΔREC/ΔCONS of SL relative to UNCONSTRAINED. Standard deviations across 5 seeds are omitted. Full results are reported in Section I. REL denotes the relationship between the predicted and ground-truth formulas: equal (=), strict (⇒), permissive (⇐), or incomparable (‖), indicating whether the predicted formula satisfying assignments are equal to, a subset of, a superset of, or neither relative to the ground truth; unknown indicates cases where the model count did not terminate. MC-R measures the ratio of the number of satisfying assignments of the predicted formula to that of the ground-truth formula. Missing cells indicate a degenerate or no formula, as for SPL (see Table 29 for the full explanation).

	UNCONSTRAINED			G-T					gpt-oss-120b				
TASK	F1 (↑)	REC (↑)	CONS (↑)	ΔF1	ΔREC	ΔCONS	REL	MC-R	ΔF1	ΔREC	ΔCONS	REL	MC-R
bdd-oia	0.58	0.58	0.90	−0.00	+0.01	+0.02	=	1.00	–	–	–	–	–
bdd-oia-2	0.73	0.71	0.92	−0.00	+0.00	+0.02	=	1.00	−0.01	−0.02	−0.02	‖	1.00
cebab	0.46	0.46	0.63	+0.00	+0.00	+0.13	=	1.00	–	–	–	–	–
chx	0.57	0.53	0.91	−0.00	+0.00	+0.07	=	1.00	−0.21	−0.14	−0.17	⇒	0.94
cifar10	0.87	0.86	0.94	−0.02	−0.01	+0.05	=	1.00	−0.20	−0.17	−0.22	‖	3.38
cifar100	0.52	0.45	0.63	+0.01	+0.03	+0.07	=	1.00	+0.01	+0.03	+0.09	=	1.00
cle4evr	1.00	1.00	1.00	+0.00	+0.00	+0.00	=	1.00	−0.01	−0.01	−0.01	=	1.00
fashion	0.92	0.92	0.99	+0.00	+0.00	+0.00	=	1.00	+0.00	+0.00	+0.00	=	1.00
kand-logic	0.72	0.71	0.19	+0.00	+0.00	+0.02	=	1.00	+0.01	+0.01	+0.02	unknown	–
kand-logic-2	0.79	0.80	0.66	+0.00	+0.00	+0.04	=	1.00	−0.09	−0.11	−0.31	⇒	0.02
mn-add-bin	0.98	0.98	0.98	+0.00	+0.00	+0.01	=	1.00	+0.00	+0.00	+0.01	=	1.00
mn-add	0.98	0.98	0.99	+0.00	+0.00	+0.00	=	1.00	−0.01	−0.01	−0.01	=	1.00
mn-mul-bin	0.71	0.71	0.99	+0.03	+0.03	+0.00	=	1.00	+0.03	+0.03	+0.00	=	1.00
mn-mul	0.98	0.98	0.99	+0.00	+0.00	+0.00	=	1.00	+0.00	+0.00	+0.01	=	1.00
road-r	0.27	0.25	0.65	+0.01	+0.02	+0.16	=	1.00	+0.02	+0.02	+0.01	‖	76.64
sudoku	0.74	0.72	0.11	+0.11	+0.12	+0.12	=	1.00	+0.08	+0.09	+0.10	=	1.00
sushi	0.16	0.14	0.06	+0.02	+0.11	+0.82	=	1.00	+0.08	+0.14	+0.90	=	1.00
warcraft	0.78	0.78	0.86	−0.01	−0.01	+0.05	=	1.00	+0.06	+0.06	+0.07	⇐	1.75

	UNCONSTRAINED			olmo3-32b-think					qwen3-8b				
TASK	F1 (↑)	REC (↑)	CONS (↑)	ΔF1	ΔREC	ΔCONS	REL	MC-R	ΔF1	ΔREC	ΔCONS	REL	MC-R
bdd-oia	0.58	0.58	0.90	–	–	–	–	–	–	–	–	–	–
bdd-oia-2	0.73	0.71	0.92	–	–	–	–	–	–	–	–	–	–
cebab	0.46	0.46	0.63	–	–	–	–	–	–	–	–	–	–
chx	0.57	0.53	0.91	−0.09	−0.00	−0.88	‖	1.00	−0.41	−0.28	−0.05	–	–
cifar10	0.87	0.86	0.94	−0.01	+0.00	+0.02	⇐	19.25	+0.00	+0.01	+0.02	⇐	19.62
cifar100	0.52	0.45	0.63	+0.00	+0.02	+0.09	=	1.00	+0.02	+0.04	+0.10	=	1.00
cle4evr	1.00	1.00	1.00	−0.00	−0.00	+0.00	=	1.00	−0.00	−0.00	+0.00	=	1.00
fashion	0.92	0.92	0.99	+0.00	+0.00	+0.00	=	1.00	−0.00	−0.00	−0.00	=	1.00
kand-logic	0.72	0.71	0.19	–	–	–	–	–	−0.02	−0.02	−0.02	unknown	–
kand-logic-2	0.79	0.80	0.66	−0.09	−0.11	−0.29	⇒	0.02	−0.09	−0.14	−0.53	‖	0.19
mn-add-bin	0.98	0.98	0.98	−0.33	−0.27	−0.30	‖	2.56	+0.00	+0.00	+0.01	⇐	2.56
mn-add	0.98	0.98	0.99	−0.00	−0.00	−0.00	=	1.00	+0.00	+0.00	+0.00	=	1.00
mn-mul-bin	0.71	0.71	0.99			–			−0.20	−0.19	−0.11	‖	2.56
mn-mul	0.98	0.98	0.99	+0.00	+0.00	+0.01	=	1.00	–	–	–	–	–
road-r	0.27	0.25	0.65	+0.01	+0.01	+0.06	‖	242.22	−0.01	+0.01	−0.28	‖	7.81
sudoku	0.74	0.72	0.11	+0.11	+0.12	+0.13	=	1.00	+0.15	+0.16	+0.31	=	1.00
sushi	0.16	0.14	0.06	+0.07	+0.13	+0.91	=	1.00	+0.08	+0.14	+0.89	=	1.00
warcraft	0.78	0.78	0.86	–	–	–	–	–	–	–	–	–	–

Discussion. Table 36 propose the same SL experiments from Table 30 but using formulas generated from non-detailed prompts. equal formulas (=) behave similarly to those generated from detailed prompts: they never reduce consistency by more than 0.01 and provide the same gains on all the tasks, e.g., sushi (+0.89 to +0.91 CONS) and sudoku (+0.10 to +0.31 CONS, +0.08 to +0.15 F1), while leaving downstream performance mostly unaffected. However, only about half of the produced formulas are exactly correct (9/16, 7/12, and 6/13 for gpt-oss-120b, olmo3-32b-think, and qwen3-8b, respectively), and the non-equal formulas are often harmful. With detailed prompts, only a single formula reduced consistency; here, the strict and incomparable

formulas reduce consistency on most of the tasks, often by a large margin: `kand-logic-2` (-0.29 to -0.53 Cons for all three models), `chx` (-0.88 for `olmo3-32b-think`), `mn-add-bin` (-0.30 for `olmo3-32b-think`), `road-r` (-0.28 for `qwen3-8b`), and `cifar10` (-0.22 for `gpt-oss-120b`). These formulas also reduce F1, e.g., by up to 0.41 for `qwen3-8b`/`chx`, whose formula admits only a single assignment of the concepts. Unlike `SPL`, however, `SL` can still be trained with such a formula as it is used as a loss function. The reason is the same as for the too-`strict` formula in Table 30. If the formula excludes some assignments that are actually present in the training data, the two losses give conflicting signals: the classification loss encourages the model to predict the ground-truth assignments, while the semantic loss pushes it away from them because they violate the formula. The model therefore ends up in a compromise that may satisfy neither the formula nor the ground-truth labels. As a result, the average consistency gain of `SL` over the unconstrained model drops from about $+0.10$ with detailed prompts to $+0.03$ for `gpt-oss-120b` and `qwen3-8b`, and becomes negative for `olmo3-32b-think` (-0.02). The mean ΔF1 ranges from -0.04 to -0.01. Permissive formulas ($\Leftarrow$) remain harmless, as with detailed prompts and for `SPL`. On `cifar10`, formulas admitting about $19\times$ more assignments than the ground truth still slightly improve consistency ($+0.02$). Similarly, the permissive `gpt-oss-120b` formula for `warcraft` improves both F1 ($+0.06$) and consistency ($+0.07$), since a superset of the valid assignments can still exclude some invalid ones. Overall, `SL` is more robust than `SPL` to imperfect formulas because it uses the formula as a soft training signal rather than enforcing it on every prediction. As a result, incorrect formulas do not necessarily prevent training, although they can still hurt performance when they conflict with the ground truth.

Table 37: **`SL` results with `qwen3-8b` generated formulas using `CPMpy` output, CoT and detailed prompt**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean $\pm$ std over 5 seeds.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)	REL	MC-R
chx	0.86 ± 0.10	0.43 ± 0.01	0.30 ± 0.11	0.26 ± 0.01	0.16 ± 0.02	–	–
cifar10	0.96 ± 0.00	0.96 ± 0.00	0.87 ± 0.00	0.87 ± 0.00	0.87 ± 0.00	$\Leftarrow$	19.62
cifar100	0.73 ± 0.00	0.99 ± 0.00	0.63 ± 0.00	0.49 ± 0.00	0.54 ± 0.00	$=$	1.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	$=$	1.00
fashion	0.99 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.91 ± 0.00	0.92 ± 0.00	$=$	1.00
kand-logic	0.17 ± 0.02	0.80 ± 0.01	0.75 ± 0.01	0.70 ± 0.01	0.70 ± 0.01	unknown	–
kand-logic-2	0.13 ± 0.02	0.82 ± 0.00	0.80 ± 0.01	0.66 ± 0.02	0.70 ± 0.01	$\parallel$	0.19
mn-add-bin	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.01	0.98 ± 0.00	$\Leftarrow$	2.56
mn-add	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	$=$	1.00
mn-mul-bin	0.88 ± 0.01	0.87 ± 0.01	0.54 ± 0.03	0.51 ± 0.02	0.51 ± 0.02	$\parallel$	2.56
road-r	0.38 ± 0.05	0.93 ± 0.00	0.36 ± 0.03	0.26 ± 0.01	0.25 ± 0.02	$\parallel$	7.81
sudoku	0.42 ± 0.16	0.95 ± 0.03	0.90 ± 0.06	0.88 ± 0.06	0.89 ± 0.06	$=$	1.00
sushi	0.95 ± 0.01	0.72 ± 0.00	0.30 ± 0.01	0.28 ± 0.00	0.24 ± 0.01	$=$	1.00

Table 38: **`SL` results with `olmo-32b-think` generated formulas using `CPMpy` output, CoT and detailed prompt**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean $\pm$ std over 5 seeds.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)	REL	MC-R
chx	0.02 ± 0.01	0.53 ± 0.02	0.62 ± 0.02	0.53 ± 0.01	0.48 ± 0.01	$\parallel$	1.00
cifar10	0.97 ± 0.00	0.96 ± 0.00	0.87 ± 0.00	0.86 ± 0.01	0.86 ± 0.01	$\Leftarrow$	19.25
cifar100	0.72 ± 0.01	0.99 ± 0.00	0.61 ± 0.01	0.47 ± 0.00	0.52 ± 0.00	$=$	1.00
cle4evr	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	$=$	1.00
fashion	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	$=$	1.00
kand-logic-2	0.36 ± 0.01	0.81 ± 0.00	0.72 ± 0.01	0.69 ± 0.01	0.70 ± 0.01	$\Rightarrow$	0.02
mn-add-bin	0.69 ± 0.03	0.68 ± 0.03	0.67 ± 0.01	0.70 ± 0.03	0.65 ± 0.03	$\parallel$	2.56
mn-add	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	$=$	1.00
mn-mul	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	$=$	1.00
road-r	0.72 ± 0.02	0.94 ± 0.00	0.39 ± 0.03	0.27 ± 0.00	0.28 ± 0.01	$\parallel$	242.22
sudoku	0.24 ± 0.09	0.92 ± 0.03	0.86 ± 0.06	0.83 ± 0.07	0.84 ± 0.07	$=$	1.00
sushi	0.96 ± 0.02	0.72 ± 0.00	0.29 ± 0.03	0.27 ± 0.01	0.23 ± 0.01	$=$	1.00

Table 39: **SL results with `gpt-oss-120b` generated formulas using `CPMpy` output, CoT and detailed prompt**. The table reports consistency, accuracy, precision, F1, REL, and MC-R. Mean ± std over 5 seeds.

TASK	CONSISTENCY (↑)	ACCURACY (↑)	PRECISION (↑)	RECALL (↑)	F1 (↑)	REL	MC-R
bdd-oia-simple	0.91 ± 0.02	0.74 ± 0.00	0.75 ± 0.00	0.69 ± 0.01	0.72 ± 0.00	$\parallel$	1.00
chx	0.74 ± 0.10	0.45 ± 0.02	0.48 ± 0.11	0.39 ± 0.02	0.36 ± 0.03	$\Rightarrow$	0.94
cifar10	0.73 ± 0.09	0.91 ± 0.01	0.83 ± 0.02	0.68 ± 0.01	0.67 ± 0.02	$\parallel$	3.38
cifar100	0.72 ± 0.01	0.99 ± 0.00	0.62 ± 0.00	0.48 ± 0.00	0.53 ± 0.00	$=$	1.00
cle4evr	0.99 ± 0.02	0.99 ± 0.02	0.99 ± 0.02	0.99 ± 0.03	0.99 ± 0.03	$=$	1.00
fashion-	1.00 ± 0.00	0.98 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	0.92 ± 0.00	$=$	1.00
kand-logic	0.21 ± 0.03	0.82 ± 0.01	0.74 ± 0.01	0.73 ± 0.01	0.73 ± 0.01	unknown	–
kand-logic-2	0.35 ± 0.01	0.81 ± 0.00	0.72 ± 0.01	0.69 ± 0.01	0.70 ± 0.00	$\Rightarrow$	0.02
mn-add-bin	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	$=$	1.00
mn-add	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	$=$	1.00
mn-mul-bin	0.99 ± 0.00	0.98 ± 0.00	0.74 ± 0.04	0.74 ± 0.04	0.74 ± 0.04	$=$	1.00
mn-mul	0.99 ± 0.00	0.99 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	0.98 ± 0.00	$=$	1.00
road-r	0.67 ± 0.03	0.95 ± 0.00	0.40 ± 0.02	0.27 ± 0.01	0.29 ± 0.01	$\parallel$	76.64
sudoku	0.21 ± 0.06	0.91 ± 0.02	0.84 ± 0.05	0.81 ± 0.05	0.82 ± 0.05	$=$	1.00
sushi	0.95 ± 0.02	0.72 ± 0.00	0.29 ± 0.02	0.28 ± 0.01	0.24 ± 0.02	$=$	1.00
warcraft	0.93 ± 0.01	0.93 ± 0.01	0.86 ± 0.01	0.84 ± 0.02	0.84 ± 0.02	$\Leftarrow$	1.75