

Reverse-Engineering Constraint Models from CNFs: An Agentic Approach

Johannes K. Fichte¹, Stefan Szeider²

¹Linköping University, Linköping, Sweden

²TU Wien, Vienna, Austria

Abstract

Compiling constraint models to conjunctive normal form (CNF) is commonly regarded as a one-way transformation, since auxiliary variables and clausal decomposition obscure the original problem structure. Related methods of structure recovery include gate extraction and cardinality detection, but those recover only fragments, not the complete encoding. We present an autonomous agent, powered by an LLM, that reverse-engineers a complete, semantically equivalent PySAT encoding from a DIMACS CNF file. We use a SAT-based equivalence checker to verify the high-level encoding and obtain counterexamples for iterative refinement. The agent queries the formula through bounded analysis tools. It combines structural evidence with abductive reasoning to hypothesize the underlying combinatorial problem. We evaluate our approach on several combinatorial problems with instances generated by independent encoders (PySAT, Picat, Sugar, BEE), giving the agent no metadata. We obtain 85% reconstruction on 48 instances and demonstrate that the agent handles direct, order, and log encodings of the same problems. Our scaling study confirms feasibility up to 1.2 million clauses.

Keywords

SAT, CNF reconstruction, large language models, semantic equivalence, constraint modeling

1. Introduction

Today’s SAT-solvers are extremely effective tools that are widely applied [1]. They rely on the DIMACS CNF format as input [2], which is a very simple textual representation of a *propositional formula* in conjunctive normal form (CNF). While propositional formulas are quite flexible, as they capture discrete, fixed finite-precision computation [3], encoding into them is often considered difficult, error-prone, and low-level. Even encoding standard problems, such as graph coloring, scheduling, or combinatorial design, can quickly become complicated. Instead, users require high-level constraints, such as all-different, exactly-one, at-most- k , exactly- k , gate semantics, or reachability. To this end, frameworks such as PySAT [4], Sugar [5], BEE [6], and Picat [7] provide a higher level of problem modeling and the frameworks automatically compile constraints into propositional formulas. This process commonly introduces auxiliary variables via the Tseitin transformation [8, 9], decomposes global constraints into clausal networks [10, 11], and replaces domain values by propositional variables. The resulting DIMACS CNF formula bears little resemblance to the original high-level constraint model, not even variable names are preserved.

However, recovering the original constraint model at a higher level of understanding has been of interest for two decades, from the perspectives of solving, modeling, and correctness. From a solving perspective, propagation on propositional variables becomes weaker than high-level constraint-level reasoning. Since knowledge about underlying structure often enables faster solving [12], researchers tried to partially recover lost structure. From the modeling perspective, a solution (satisfying assignment) to a propositional formula allows us to recover the original domain variables and thereby gives the connection to the original problem. The user usually preserves the semantic variables with the connection to the domain model, but ignores the auxiliary ones. From the correctness perspective, we want to check that the low-level encoding encodes what is promised and does not accidentally introduce “spurious” solutions.

PoS 2026: 17th International Workshop on Pragmatics of SAT, July 19, 2026, Lisbon, Portugal

✉ johannes.fichte@liu.se (J. K. Fichte); sz@ac.tuwien.ac.at (S. Szeider)

id 0000-0002-8681-7470 (J. K. Fichte); 0000-0001-8994-1656 (S. Szeider)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Three scenarios motivate reconstruction. (1) SAT-competition and industrial CNFs often arrive without their source model; a recovered model enables preprocessing and solver selection. (2) The recovered model is an independent witness against encoder bugs: any spurious solution introduced by the compiler fails the equivalence check. (3) A recovered model can be re-encoded in a different framework (e.g., Picat to PySAT) without access to the original source.

These considerations yield three natural levels of what we can recover from a propositional formula. *Level 1* aims at *recovering gates* (AND, OR, XOR, and equivalence subformulas extractable from Tseitin-style clause groups) that are lost due to translating formulas into CNF thereby flattening structure. Gates are often present in encodings that originate from circuits, bit-blasted arithmetic, and planning. This level is well explored by various techniques that recover specific gates [13], entire circuits [14], or local functional semantic dependencies between variables, e.g., via Craig interpolation [15], integrating syntactic and semantic methods into solvers [16], and matching and eliminating gates by clausal congruence closure [17]. *Level 2* recovers *specific constraint types* for more effective solving, but yields no global picture of the encoded problem. For example, recovering syntactical and semantical at-most- k and exactly- k cardinality constraints [18], implementing cardinality extraction into solvers [19], and extracting cardinality constraints and lifting them into a native solver format [20]. *Level 3* aims at reconstructing a high-level model of the *entire combinatorial problem* that produced a given CNF, including parameters, and variable mappings. This has not been addressed to date and neither gate extraction nor constraint detection can perform the necessary abductive step.

Main Contributions We can bridge the gap between Levels 2 and 3 by an LLM agent that uses bounded analysis tools, that is, fixed library functions whose outputs are capped in size so the agent never reads the entire formula. Therefore, we combine three capabilities absent from prior methods: *world knowledge* about combinatorial problem classes, *abductive reasoning* from incomplete structure, and *code generation* to produce executable SAT encodings. In detail: First, we define *semantic equivalence* as the correctness criterion for reconstruction, via equi-satisfiability on the semantic (non-auxiliary) variables: two CNFs are equivalent when their solution sets agree on these variables up to renaming. Second, we present the first agentic architecture that *reverse-engineers constraint models* from low-level DIMACS CNFs, recovering the combinatorial problem, its parameters, the constraints, and the semantic variable mapping; we employ the AI models Gemini 3 Flash, Claude Sonnet 4.5, and Gemini 3.1 Pro, and the agent never sees the full DIMACS CNF file but works iteratively. Third, we evaluate reconstruction across direct, order, and log encodings on a benchmark of 12 problems and 48 instances encoded by four independent systems (PySAT, Sugar, Picat SAT, and BEE); a three-stage *cascade* (a sequence of models of increasing capability, where an instance advances to the next only if reconstruction fails) yields 85% reconstruction across all instances and scales to instances with up to 1.2 million clauses.

Related Works There are several works for Levels 1 and 2. Gate extraction methods [13, 14, 15, 16, 17] recover functional dependencies, while cardinality detection [18, 20] recovers specific constraint types. Both directions are algorithmic and aim for polynomial-time runtime. Due to the nature of the task, they are limited to recovering fragments. We build on top of these fragments by adding the abductive reasoning step that infers the global model. The agent’s analysis tools perform the same extractions as the aforementioned methods. Complementary, recent work has explored how LLMs translate natural-language descriptions into formal constraint models, such as streamlining constraint generation [21], evaluation benchmarks for constraint model generation [22], neuro-symbolic constraint programming [23], and solver-integrated LLM predictions [24]. These works address the *forward* direction (description to constraint model to CNF), while our work addresses the *reverse* direction (CNF to constraint model). Here, the LLM acts as a “forensic analyst” rather than as a “modeler.” Very recently, LLMs have also been used for generating tailored solver heuristics in automated reasoning [25, 26].

2. Preliminaries

We assume that the reader is familiar with basics in propositional logic [1, 27] and briefly recap notions. Let U be a universe of propositional variables. A *literal* is a variable x or its negation $\neg x$. A *clause* is a finite set of literals, interpreted as the disjunction of these literals. A propositional *formula* F in *conjunctive normal form* (CNF) is a finite set of clauses, interpreted as the conjunction of its clauses. We can efficiently transform any formula into a CNF using Tseitin’s transformations [8], but this transformation introduces new (functionally-dependent) auxiliary variables. We let $\text{vars}(F)$ be the set of the variables that occur in F . An *assignment* is a mapping $\tau : X \rightarrow \{0, 1\}$ defined for a set $X \subseteq U$ of variables. The formula F *under assignment* τ is the formula F_τ obtained from F by (i) removing all clauses c that contain a literal set to 1 by τ and then (ii) removing from the remaining clauses all literals set to 0 by τ . An assignment τ *satisfies* a given formula F if $F_\tau = \emptyset$ and τ *falsifies* F if $\emptyset \in F_\tau$. We let $\text{SAT}(F)$ consist of the set of all satisfying assignments of F .

3. Reverse-Engineering CNFs

Before we can reverse-engineer a low-level CNF encoding into a high-level constraint model, we define a basic correctness notion for reconstructing constraints. We take inspiration from equivalence checking in hardware and software verification [28], which captures when two specifications admit exactly the same set of satisfying assignments. We employ equivalence on the CNF level to decide whether two encodings are interchangeable. This allows us to replace an original formula F with a reconstructed formula G . About F , we only know that it has been generated by a high-level constraint model, but we are not aware of any details of the high-level constraint model. Instead, we iteratively generate a constraint model and thereby formula G and verify whether the resulting formula G is equivalent to F . A SAT encoding is a function that maps a constraint model to a CNF formula F over propositional variables. Typically, we distinguish between *semantic variables*, a subset $S_F \subseteq \text{vars}(F)$ that carries the problem’s meaning, and *auxiliary variables*, introduced by the encoding machinery. By semantic equivalence, we capture the idea that a reconstruction needs to preserve the semantic variables (up to renaming) but does not have to replicate the auxiliary variable structure, since only the semantic content matters. To this end, we require a bijection on the semantic variables and satisfiability on the auxiliary ones as formalized in Definition 1.

Definition 1 (Semantic Equivalence). Let F and G be CNF formulas with sets S_F and S_G of semantic variables. A bijection $\sigma : S_F \rightarrow S_G$ witnesses *semantic equivalence*, written $F \equiv_\sigma G$, if for every assignment $\alpha : S_F \rightarrow \{0, 1\}$: F_α is satisfiable $\iff G_{\sigma(\alpha)}$ is satisfiable.

Example 1 (Sudoku). Consider the *Sudoku* problem, which is a logic-based, combinatorial number-placement puzzle that is well known [29, 30]. Classically, we have a 9×9 -grid with nine 3×3 -sub-grids. Each entry (*cell*) in the grid may contain a digit from 1 to 9 subject to row, column, and sub-grid conditions. When encoding into a CNF formula F , we introduce propositional variables $x_{r,c,n}$ expressing that in cell (r, c) number n is placed. These are the semantic variables of our encoding. Then, we construct the constraints, which mostly contain exactly once conditions on digits splitting into an “at most one” and “at least one” part. The at most constraints may become exclusions or a more compact cardinality encoding. The compact variant requires auxiliary variables; from a modeling perspective the two are still semantically equivalent, even though cardinalities can be encoded differently. $\perp$

Testing Semantic Equivalence To reverse-engineer CNFs, we employ semantic equivalence (Def. 1). We take two CNFs F and G and bijection $\sigma : S_F \rightarrow S_G$ and check whether they are semantically equivalent, i.e., $F \equiv_\sigma G$ holds. Therefore, we test as follows:

1. We check whether there is a satisfying assignment of F that is missing from G , i.e., there is an assignment $\alpha \in \text{SAT}(F)$ whose renaming $\sigma(\alpha)$ falsifies G .

Table 1

Bounded analysis tools provided to the agent.

Function	Description
<code>stats()</code>	Variable/clause counts, clause-length distribution.
<code>clause_patterns()</code>	Polarity pattern frequencies across all clauses.
<code>detect_eo_groups()</code>	Find exactly-one and at-most-one constraint groups.
<code>analyze_gates(limit)</code>	Identify Tseitin gate structures; <code>limit</code> caps output size.
<code>extract_graph()</code>	Build conflict graph from binary negative clauses.
<code>get_clauses(start, limit)</code>	Bounded clause access; never dumps entire formula.
<code>var_profile(v)</code>	Per-variable polarity and clause-membership statistics.
<code>solve()</code>	Obtain a satisfying assignment.
<code>check_equiv(other, σ)</code>	SAT-based semantic equivalence verification.

2. We check whether there is a satisfying assignment of G that is missing from F , i.e., a satisfying assignment $\beta \in \text{SAT}(G)$ such that the inverse renaming $\sigma^{-1}(\beta)$ falsifies F .

If we run these checks using a SAT solver, we also obtain a counterexample in the negative case. Then, we can iteratively modify G and test again. Concretely, each direction runs a counterexample-guided loop. The forward check repeatedly asks the solver for a model of F , projects it onto the semantic variables, and tests whether that projection extends to a model of G ; the backward check is symmetric. A projection that does not extend is a witness that the agent can use to refine G . We add each witness as a blocking clause and continue, so we never materialize the full solution set, and in practice the loop closes after a few iterations (two per direction in Example 2). If no counterexample can be found in either direction, the procedure terminates and establishes equivalence. For satisfiable instances, it is easy to see that the procedure is sound and complete. In practice, it turns out that it requires only a small number of iterations when running within an agentic architecture.

Agentic Architecture to Reconstruct the Constraint Model Our agentic approach to reconstruct the constraint model then proceeds in four phases. Figure 1 illustrates the architecture and experimental setup. We assume that the original CNF is constructed from a constraint model and give the instance to the agent, which never reads all clauses and accesses the formula only through the library functions and a bounded subset of clauses. It runs a bounded analysis, hypothesizes a PySAT constraint model, reconstructs the new CNF, compares, and iterates. In more detail: The agent runs a bounded analysis on the input formula F_{orig} using the tools listed in Table 1, including clause statistics, EO group counts and sizes, gate types and frequencies, and the conflict graph. No other interface is available to the agent. All access is bounded to prevent context overflow. These tools mirror what Level 1 and Level 2 recovery methods (as listed in the introduction) compute algorithmically. They serve as *inputs* to LLM reasoning rather than as standalone detectors. In addition, the agent also obtains a satisfying assignment. Overall, the data provides the agent with a fingerprint of the instance. From the fingerprint, the agent infers the combinatorial problem and its parameters via abductive reasoning. For example, in a standard Sudoku-to-SAT encoding, we would expect roughly 729 indicator variables together with many size-9 exactly-one groups (cell, row-digit, column-digit, and box-digit). A characteristic length/polarity pattern, together with mutual-exclusion patterns, would show up, all of which could be revealed by `stats()`, `detect_eo_groups()`, `clause_patterns()`, and `extract_graph()`. We illustrate the full process on a 9×9 Sudoku in Example 2. Inspecting the satisfying assignment confirms whether the decoded solution is consistent with the hypothesis. Then, the agent synthesizes a constraint model (PySAT encoding) for the hypothesized problem, defining semantic variables and all necessary constraints. The synthesized model is an executable PySAT script with named global constraints (such as exactly-one, at-most- k , all-different), explicit semantic-variable indexing, and recovered problem parameters (grid size, ruler length, etc.); it is not a clause dump. Example 2 shows a concrete instance. By running PySAT on the reconstructed constraint model, the agent obtains a new DIMACS file of the inferred CNF formula G_{inf} together with a variable mapping σ between the semantic variables of

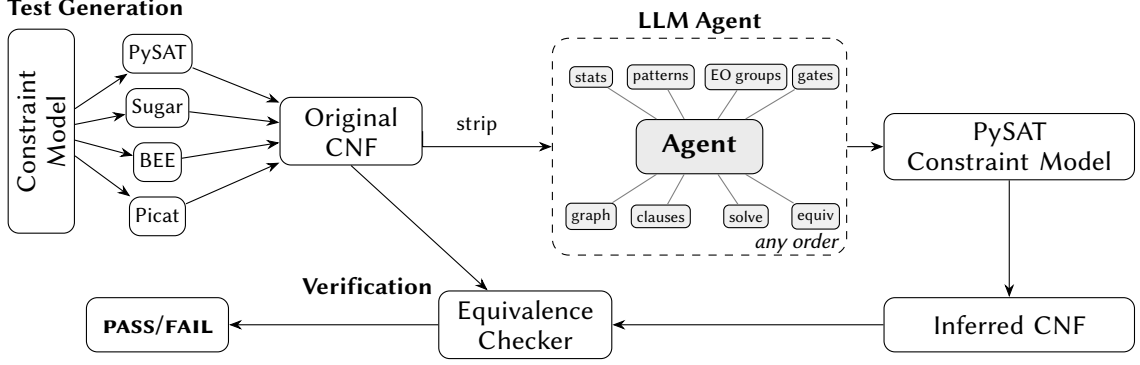


Figure 1: Experimental Setup. In *Test Generation* (left), we produce a CNF via one of the frameworks and strip metadata. Then, the *LLM Agent* (center) freely queries in any order bounded analysis tools (including CNF generation and equivalence check) within the budget to hypothesize a problem/PySAT model. When the agent outputs the constraint model, we generate the inferred CNF, and verify (bottom) semantic equivalence of both CNFs to decide pass or fail.

F_{orig} and G_{inf} . Next, the agent may employ the SAT-based equivalence checker as described above to compare the input CNF F_{orig} and generated CNF G_{inf} under σ . If equivalence fails, the counterexample identifies a concrete assignment that satisfies one formula but not the other. A satisfying assignment present in the formula G_{inf} obtained from the hypothesized model but absent from the original F_{orig} indicates a missing constraint (under-constrained model). Vice-versa, a satisfying assignment present in F_{orig} but missing in G_{inf} indicates too strict/too many constraints (over-constrained). Then, the agent can apply a targeted fix and re-verify.

Example 2 (Blind Recovery of a Sudoku Encoding). We illustrate the reconstruction process with a 9×9 Sudoku instance (cf. Example 1); the Sudoku instances evaluated in Section 4 are larger (16×16 and 25×25).

The input to the agent is a DIMACS file; **all comments and any indication of the problem type have been removed**. We reproduce the agent’s reasoning verbatim below, condensed for space. The complete run required five tool calls (four code executions plus the final model submission) and 37 seconds.

Step 1: Structural fingerprint. The agent calls `stats()`, `clause_patterns()`, `detect_eo_groups()`, `analyze_gates()`, and `get_clauses()` in a single action. The results show 729 variables (all semantic, zero auxiliary), 12,018 clauses with lengths $\{1:30, 2:11,664, 9:324\}$, 324 exactly-one (EO) groups of uniform size 9, zero Tseitin gates, and 10,206 conflict-graph edges. The agent determines that the formula is satisfiable.

Step 2: Detailed analysis. The agent verifies that the variables span a contiguous range 1–729 and that every variable appears in exactly four EO groups ($324 \times 9/729 = 4$). It confirms that all 324 clauses of length 9 are fully positive (at-least-one clauses) and that all 11,664 binary clauses are fully negative (pairwise exclusions). This is consistent with a pairwise encoding: $324 \times (1 + \binom{9}{2}) + 30 = 324 \times 37 + 30 = 12,018$. The agent identifies the 30 unit clauses as given clues.

Step 3: Problem identification. From the numerical fingerprint, the agent determines the problem type. We quote its reasoning from the code comments: “ $729 = 9 \times 9 \times 9$. This is likely a 9×9 Sudoku. Variables: (row, col, val) where row, col, val in $0..8$. Let’s see if the EO groups correspond to the standard Sudoku constraints: (1) each cell has exactly one value (81 groups), (2) each row has each value exactly once (81 groups), (3) each column has each value exactly once (81 groups), (4) each 3×3 box has each value exactly once (81 groups). Total = $81 \times 4 = 324$ groups. This matches.”

The agent hypothesizes the variable mapping $\text{var}(r, c, v) = 81r + 9c + v + 1$ and verifies it; in particular, the first EO group $\{1, \dots, 9\}$ corresponds to cell $(0, 0)$, and $\{1, 10, 19, 28, 37, 46, 55, 64, 73\}$ corresponds to row 0, value 0.

Step 4: Reconstruction and verification. The reconstructed encoding consists of all four families of

exactly-one constraints (cell, row, column, box), each encoded pairwise, together with the 30 unit clauses for the given digits. The reconstructed variable indexing coincides with the original, and the resulting mapping is the identity. The equivalence checker confirms: “*equivalent: True, forward_iterations: 2, backward_iterations: 2*”, that is, the solutions agree under projection onto the 729 semantic variables. $\perp$

4. Empirical Evaluation

To investigate whether our proposed agentic approach is able to reasonably reconstruct high-level structure and the constraint models in practice, we conduct an experimental analysis with CNF encodings of classical constraint problems. The agent operates under a *blind test protocol*. It can only access the file using the bounded analysis tools (see Table 1) and never reads the entire file. In addition, we sanitize the DIMACS input, strip all comments, filename hints, and metadata. We provide all details of the evaluation, source code, data, prompts, and outputs [31].

Experimental Setup, Modeling Frameworks, and Metrics In our experiments, we employ a *three-stage cascade* of LLMs with increasing capability. Stage A uses Google Gemini 3 Flash, a cost-efficient model (temperature 0, max output 16,384 tokens). Stage B uses Anthropic Claude Sonnet 4.5, a mid-tier model. Stage C uses Google Gemini 3.1 Pro with extended thinking enabled at low reasoning effort, serving as a reasoning-enhanced rescue stage. All models are accessed via the OpenRouter API. An instance advances to the next stage only if reconstruction fails at the current one. Retrying a failed instance with the same model does not help in our setting. All stages use the same agentic approach with a 35-step budget per attempt. Within this budget, the agent must complete analysis, hypothesis formation, code generation, and verification. We encode each problem by four independent modeling frameworks that produce structurally different CNFs from the same specification: PySAT [4], using direct Boolean encoding with sequential counters [10] and totalizers [11] for cardinality constraints, Sugar [5], employing the order encoding resulting in a chain of Boolean order variables for each finite-domain variable, Picat SAT [7], using the log encoding with $\lceil \log_2 d \rceil$ bits per domain variable, and BEE [6], taking the order encoding with equi-propagation that simplifies constraints before clausification and may merge or eliminate variables. Note that we enforced distinct import aliases in the Python imports, otherwise a name collision between PySAT’s CNF class and the analysis library’s CNF class caused silent failures in cheaper models. An instance counts as a PASS if and only if the equivalence checker confirms semantic equivalence between the original and reconstructed CNF. All other outcomes (incorrect reconstruction, timeout, or no output) count as FAIL.

Benchmark Problems and Instance Dataset As our benchmark suite, we select 12 classical constraint problems spanning diverse constraint types, including equitable coloring, Sudoku, quasigroup completion, BIBD, social golfer, Golomb ruler, Costas array, graceful labeling, NAND synthesis, Schur partition, Langford pairing, and sports tournament scheduling, which cover a wide range of structural patterns such as specific semantic variables, cardinality constraints, symmetry, arithmetic structure, and circuit-derived gates. All instances are satisfiable due to equivalence checking. In total, we have 48 instances (12 problems $\times$ 4 encoders).

Experimental Design and Aims To test our agentic approach and its reconstruction capabilities, we conduct two experiments: a general encoding reconstruction check and a scaling study. First, by *Experiment 1 (reconstruction across varying encoders)*, we challenge our agentic-reasoning approach to recognize the same combinatorial problem through a fundamentally different variable representation. Order-encoded formulas (Sugar, BEE) exhibit monotonic implication chains absent from direct Boolean encodings. Log-encoded formulas (Picat) represent each domain variable with $\lceil \log_2 d \rceil$ bits, producing clause structures that share no syntactic pattern with PySAT encodings. BEE adds a further complication: equi-propagation may merge or eliminate variables before clausification, obscuring the original constraint boundaries. The cross-encoder design also separates structural reconstruction

Table 2

Reconstruction Results. Stage (S): A (cheapest), B (mid-tier), C (reasoning model); # = number of tool calls; Tok = total tokens in thousands; — means failure across all stages.

Problem	PySAT				Picat				Sugar				BEE			
	Vars/Cls	S	#	Tok	Vars/Cls	S	#	Tok	Vars/Cls	S	#	Tok	Vars/Cls	S	#	Tok
<i>BIBD</i>	1,944/4,503	A	23	372	735/5,502	B	29	801	2,514/9,108	—	—	—	836/4,589	—	—	—
<i>Costas</i>	931/10,795	A	12	236	397/5,667	A	13	209	364/2,751	A	24	478	523/4,640	B	31	974
<i>Golomb</i>	720/27,604	A	15	256	131/980	A	21	712	157/948	A	21	338	99/792	—	—	—
<i>Graceful</i>	181/2,054	A	26	596	298/2,416	—	—	—	477/3,445	A	29	759	181/1,531	A	22	458
<i>Langford</i>	584/1,489	A	27	561	224/1,765	A	27	606	568/1,990	A	28	561	232/971	A	13	162
<i>Eq. coloring</i>	140/370	B	34	986	76/1,044	B	33	871	136/1,622	C	4	38	105/386	C	4	35
<i>NAND</i>	188/4,205	B	26	729	406/1,626	A	33	679	126/2,885	A	15	205	150/2,349	A	18	268
<i>QCP</i>	216/1,737	A	3	34	162/1,140	A	21	356	399/1,278	A	35	945	88/479	A	13	171
<i>Schur</i>	160/1,881	A	7	85	4,458/10,770	C	4	38	157/1,658	A	29	842	156/1,715	A	10	136
<i>SGP</i>	444/2,016	—	—	—	126/1,206	A	33	687	372/1,188	C	4	35	217/1,108	—	—	—
<i>Sports</i>	196/1,849	A	5	52	1,267/3,385	—	—	—	988/4,014	A	15	253	178/1,573	A	6	70
<i>Sudoku</i>	4,096/124,004	A	7	79	987/7,383	A	23	483	10,707/66,924	A	31	1,353	813/5,715	A	35	806

from pattern memorization: a memorized PySAT template cannot pass the equivalence checker on a log-encoded CNF, so success on the Picat encodings can only come from recovering the bit-to-value mapping structurally. Second, by *Experiment 2 (scaling)*, we assess how reconstruction scales with instance size. Therefore, we apply our approach to four problem families, which were the easiest to scale among the considered instances: equitable graph coloring on grids (1,106–69,886 clauses), Costas arrays (1,511–164,583 clauses), Golomb rulers (208,339–5,415,366 clauses), and Sudoku (16×16 with 124,004 clauses and 25×25 with 752,750 clauses). The two experiments use independent runs. A few instances appear in both (Sudoku 16×16 and Costas-7); their tool-call counts and token totals differ, since LLM sampling is not fully deterministic even at temperature 0.

Experiment 1 (Reconstruction Across Varying Encodings) We provide our results for 12 problems × 4 encoders under the three-stage cascade in Table 2. Stage A (cheapest) solves 32 of 48 instances. Stage B (mid-tier) recovers 5 additional instances that Stage A missed. Stage C (reasoning), which uses a reasoning-enhanced model with extended thinking, rescues 4 more. The cascade reaches 41/48 overall (85%). Reconstruction difficulty varies more by encoder than by problem size. PySAT instances pass at the highest rate (11/12, 92%), followed by Sugar (11/12, 92%), Picat (10/12, 83%), and BEE (9/12, 75%). BEE’s lower rate reflects the structural simplification performed by equi-propagation: the resulting formulas contain fewer variables but less regular clause patterns, making abductive reasoning harder. Difficulty tracks the encoding structure rather than how familiar the problem is. Sudoku and the social golfer problem are both classical, yet Sudoku is recovered under all four encoders while the social golfer problem is among the hardest in the suite; conversely, problems that are rarely cast as SAT, such as Costas arrays and NAND synthesis, are recovered under all four. The agent thus relies on structural recovery, not on recalling a known encoding. When considering the three-stage cascade, we observe that the reasoning model at Stage C exhibits a distinctive pattern: it solves hard instances in only 4 steps with 35k tokens, compared to the 400k+ tokens typical of the failed Stage A attempts. This suggests that extended reasoning compensates for fewer tool interactions. However, the reasoning model is unreliable on instances that Stage A handles easily, making the cascade essential: each stage contributes instances that other stages miss. When investigating failures, we observe that 7 of 48 instances remain unsolved across all three stages. The social golfer problem and BIBD are the hardest, each failing on two of the four encoders. The failures have three sources. First, some problems lack a distinctive local fingerprint: the social golfer problem couples groups and weeks through a global pairwise-balance condition (no

Table 3

Scaling Results. Stage (S): A = Gemini 3 Flash, B = Sonnet 4.5, # = number of tool calls; Tok = total tokens in thousands. — means unsolved by all stages.

Problem	Instance	Vars/Cls	S	#	Tok
<i>Equitable coloring</i> (grid, $k=3$)	Grid 5×5	507/1,106	—	—	—
	Grid 6×6	972/2,089	A	18	300
	Grid 7×7	1,779/3,758	A	11	161
	Grid 8×8	2,964/6,197	—	—	—
	Grid 10×10	7,032/14,501	A	22	408
	Grid 12×12	14,256/29,161	A	21	328
	Grid 15×15	34,425/69,886	A	23	519
<i>Costas array</i>	Costas-5	225/1,511	A	13	181
	Costas-6	486/4,343	A	24	412
	Costas-7	931/10,795	A	11	132
	Costas-8	1,632/23,985	B	31	830
	Costas-9	2,673/48,763	B	27	615
	Costas-10	4,150/92,271	B	35	843
	Costas-11	6,171/164,583	B	32	836
<i>Golomb ruler</i>	Ruler(6, 17)	2,403/208,339	A	27	565
	Ruler(7, 25)	7,007/1,239,727	B	32	745
	Ruler(8, 34)	16,940/5,415,366	—	—	—
<i>Sudoku</i>	16×16	4,096/124,004	A	5	171
	25×25	15,625/752,750	B	15	707

two golfers are grouped together more than once), which the agent fails to recognize in both the direct PySAT and the BEE encoding. Second, order encodings of complex cardinality structure (BIBD Sugar and BEE) produce deeply nested implication chains that the agent cannot decompose into recognizable constraint groups. Third, BEE’s equi-propagation can erase the structural regularity that hypothesis formation relies on: the Golomb BEE instance has fewer variables (99) than any other Golomb encoding, yet is the only Golomb encoding that fails. The two remaining failures are log encodings, graceful labeling and sports scheduling under Picat, where the bit-level representation obscures the constraint boundaries.

Experiment 2 (Scaling the Size of Encodings) We investigate the results by encoding framework and report the results in Table 3. Three observations are particularly interesting: difficulty is non-monotonic, recognizability dominates size, and the cascade allows us to reconstruct larger instances. In more detail: Grid 10×10 (14,501 clauses) passes Stage A, while the smaller Grid 5×5 (1,106 clauses) and Grid 8×8 (6,197 clauses) fail all three stages. For equitable coloring, the encoding requires cardinality constraints that vary irregularly with the grid dimensions, making the auxiliary structure more complicated and some mid-sized instances harder to reconstruct than larger ones. Sudoku at 124,004 clauses passes by 5 tool calls, as its structural fingerprint is immediately recognizable. It has 1024 exactly-one groups of size 16 in a regular pattern. Golomb rulers at 208,339 clauses require 27 calls because the all-different-distances constraint produces a less distinctive clause pattern. Stage B pushes the boundary for Costas from size 7 to 11 (164,583 clauses) and the boundary for Golomb from (6, 17) to (7, 25) at 1.2 million clauses. Stage C does not help when scaling, in contrast to Experiment 1. Finally, Ruler(8, 34) consisting of 5.4 million clauses remains the largest unsolved instance.

Cost and Overhead Reconstruction is more expensive than algorithmic fragment detection. A single reconstruction consumes between 30,000 and 1,400,000 tokens, depending on the instance size and the number of refinement iterations. The cascade keeps the average cost moderate, since the cheap Stage A already reconstructs 32 of 48 instances and only the failures escalate to the more expensive reasoning stages. In return, we obtain a complete, re-runnable model together with a machine-checked

equivalence certificate, rather than an isolated gate or cardinality constraint. Several routes can reduce the overhead further: the bounded analysis tools are themselves cheap and can act as a first-pass filter that decides whether an instance warrants an LLM call, repeated or low-information analysis calls can be pruned or aggregated, and tool outputs can be cached across refinement attempts on the same instance.

Limitations Our approach targets CNFs that are compiled from a high-level constraint model, where the input admits an identifiable split into semantic and auxiliary variables and the instance is satisfiable, as required by the equivalence check. Our benchmark covers classical combinatorial problems, and we expect the behavior on industrial or less regular CNFs to differ. Such formulas need not originate from a single global combinatorial model, and there we expect the bounded analysis tools (Level 1 and 2 recovery) to keep applying while full Level 3 reconstruction degrades to recovering fragments rather than a complete model. The cross-encoder results are mildly encouraging in this respect, since the agent already succeeds on log and order representations that share no surface structure with the direct encoding. BEE’s equi-propagation and complex cardinality structures remain the main source of cascade failures.

5. Conclusion

We established the first agentic approach for reconstructing constraint models from DIMACS CNF input. Our results challenge the long-standing assumption that compiling to CNF irrecoverably destroys problem structure. In fact, by using an LLM agent, we can recover the combinatorial problem, its parameters, the full constraint model, and the variable mapping. Notably, the agent gets only bounded access to a formula and can formulate queries using a fixed set of library functions, which employ existing structural techniques (gate and constraint fragment recovery). On top, a crucial part is an abductive step that hypothesizes a global model using those fragments. In practice, we cascade three types of models (cheap, mid-tier, reasoning) and obtain 85% reconstruction across the four different CNF generation frameworks. Thereby, we demonstrate that reconstructing is feasible. Our scalability experiments illustrate that, unsurprisingly, problem recognizability matters more than instance size. Sudoku at 752,750 clauses is reconstructed in 15 tool calls, while smaller Costas instances require over 30. The cascade extends the scaling boundary to 1.2 million clauses for Golomb rulers. Our approach is quite promising but is also much more costly than algorithmic techniques that just detect fragments. For future work, we are interested in exploring natural extensions, in particular, industrial CNF from SAT Competition instances and hardware verification, systematic evaluation without semantic variable annotations, and using cheaper models as first-pass filters to reduce the cost of the cascade.

Declaration on Generative AI

During the preparation of this work, the authors used Claude Code (Anthropic) for grammar and spelling check and code review, and ChatGPT (OpenAI) for literature search assistance. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication’s content.

References

- [1] A. Biere, M. Heule, H. van Maaren, T. Walsh, Handbook of Satisfiability, IOS Press, 2021. doi:10.3233/FAIA336.
- [2] M. Trick, V. Chvátal, B. Cook, D. Johnson, C. McGeoch, B. Tarjan, The 2nd DIMACS implementation challenge: 1992–1993 on NP hard problems: Maximum clique, graph coloring, and satisfiability, <http://archive.dimacs.rutgers.edu/pub/challenge/sat/benchmarks/>, 1993.

- [3] C. E. Shannon, A symbolic analysis of relay and switching circuits, *Electrical Engineering* 57 (1938) 713–723. doi:10.1109/EE.1938.6431064.
- [4] A. Ignatiev, A. Morgado, J. Marques-Silva, PySAT: A python toolkit for prototyping with SAT oracles, in: O. Beyersdorff, C. M. Wintersteiger (Eds.), *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT'18)*, volume 10929 of *Lecture Notes in Computer Science*, Springer Verlag, 2018, pp. 428–437. doi:10.1007/978-3-319-94144-8_26.
- [5] N. Tamura, A. Taga, S. Kitagawa, M. Banbara, Compiling finite linear CSP into SAT, *Constraints* 14 (2009) 254–272. doi:10.1007/S10601-008-9061-0.
- [6] A. Metodi, M. Codish, Compiling finite domain constraints to SAT with BEE, *Theory Pract. Log. Program.* 12 (2012) 465–483. doi:10.1017/S1471068412000130.
- [7] N. Zhou, H. Kjellerstrand, J. Fruhman, *Constraint Solving and Planning with Picat*, Springer Briefs in Intelligent Systems, Springer, 2015. doi:10.1007/978-3-319-25883-6.
- [8] G. C. Tseitin, On the complexity of derivations in propositional calculus, in: *Studies in Mathematics and Mathematical Logic, Part II*, Matematicheskii institut im. V.A. Steklova. Leningradskoe otделение, 1968, pp. 115–125. doi:10.1007/978-3-642-81955-1_28.
- [9] D. A. Plaisted, S. Greenbaum, A structure-preserving clause form translation, *J. Symb. Comput.* 2 (1986) 293–304. doi:10.1016/S0747-7171(86)80028-1.
- [10] C. Sinz, Towards an optimal CNF encoding of Boolean cardinality constraints, in: P. van Beek (Ed.), *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, volume 3709 of *Lecture Notes in Computer Science*, Springer Verlag, 2005, pp. 827–831. doi:10.1007/11564751_73.
- [11] O. Bailleux, Y. Bouffkhad, Efficient CNF encoding of Boolean cardinality constraints, in: F. Rossi (Ed.), *Proceedings of the 9th International Conference Principles and Practice of Constraint Programming (CP'03)*, volume 2833 of *Lecture Notes in Computer Science*, Springer Verlag, 2003, pp. 108–122. doi:10.1007/978-3-540-45193-8_8.
- [12] J. Elffers, J. Nordström, Divide and conquer: Towards faster Pseudo-Boolean solving, in: J. Lang (Ed.), *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*, volume 18, International Joint Conferences on Artificial Intelligence Organization, 2018, pp. 1291–1299. doi:10.24963/ijcai.2018/180.
- [13] R. Ostrowski, É. Grégoire, B. Mazure, L. Sais, Recovering and exploiting structural knowledge from CNF formulas, in: P. V. Hentenryck (Ed.), *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *Lecture Notes in Computer Science*, Springer Verlag, Ithaca, NY, USA, 2002, pp. 185–199. doi:10.1007/3-540-46135-3_13.
- [14] Z. Fu, S. Malik, Extracting logic circuit structure from conjunctive normal form descriptions, in: N. Bhat, S. Chakradhar (Eds.), *Proceedings of the 20th International Conference on VLSI Design (VLSID'07)*, IEEE Computer Soc., 2007, pp. 37–42. doi:10.1109/VLSID.2007.81.
- [15] F. Slivovsky, Interpolation-based semantic gate extraction and its applications to QBF preprocessing, in: S. K. Lahiri, C. Wang (Eds.), *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV'20)*, volume 12224 of *Lecture Notes in Computer Science*, Springer Verlag, 2020, pp. 508–528. doi:10.1007/978-3-030-53288-8_24.
- [16] M. Fleury, A. Biere, Mining definitions in kissat with kittens, *Formal Methods in System Design* 60 (2022) 381–404. doi:10.1007/S10703-023-00421-2.
- [17] A. Biere, K. Fazekas, M. Fleury, N. Froleys, Clausal congruence closure, in: S. Chakraborty, J. R. Jiang (Eds.), *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT'24)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl Publishing, 2024, pp. 6:1–6:25. doi:10.4230/LIPICS.SAT.2024.6.
- [18] A. Biere, D. L. Berre, E. Lonca, N. Manthey, Detecting cardinality constraints in CNF, in: C. Sinz, U. Egly (Eds.), *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT'14)*, volume 8561 of *Lecture Notes in Computer Science*, Springer Verlag, 2014, pp. 285–301. doi:10.1007/978-3-319-09284-3_22.
- [19] D. L. Berre, A. Parrain, The Sat4j library, release 2.2: System description, *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2011) 59–64. doi:10.3233/SAT190075.

- [20] J. E. Reeves, M. J. H. Heule, R. E. Bryant, From clauses to klauses, in: A. Gurfinkel, V. Ganesh (Eds.), Proceedings of the 36th International Conference on Computer Aided Verification (CAV'24), volume 14681 of *Lecture Notes in Computer Science*, Springer Verlag, 2024, pp. 110–132. doi:10.1007/978-3-031-65627-9_6.
- [21] F. Voboril, V. P. Ramaswamy, S. Szeider, Generating streamlining constraints with large language models, *Journal of Artificial Intelligence Research* 84 (2025). doi:10.1613/JAIR.1.18965.
- [22] K. Michailidis, D. Tsouros, T. Guns, CP-bench: Evaluating large language models for constraint modelling, in: I. Lynce, N. Murano, M. Vallati, S. Villata, F. Chesani, M. Milano, A. Omicini, M. Dastani (Eds.), Proceedings of the 28th European Conference on Artificial Intelligence (ECAI'25), volume 413 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2025, pp. 861–868. doi:10.3233/FAIA250890.
- [23] W. Shi, M. Liu, W. Zhang, L. Shi, F. Jia, F. Ma, J. Zhang, ConstraintLLM: A neuro-symbolic framework for industrial-level constraint programming, in: C. Christodoulopoulos, T. Chakraborty, C. Rose, V. Peng (Eds.), Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP'25), Association for Computational Linguistics, 2025, pp. 15999–16019. doi:10.18653/V1/2025.EMNLP-MAIN.809.
- [24] F. Régim, E. D. Maria, A. Bonlarron, Combining constraint programming reasoning with large language model predictions, in: P. Shaw (Ed.), Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP'24), volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl Publishing, 2024, pp. 25:1–25:18. doi:10.4230/LIPICS.CP.2024.25.
- [25] Y. Sun, F. Ye, X. Zhang, S. Huang, B. Zhang, K. Wei, S. Cai, AutoSAT: Automatically optimize SAT solvers via large language models, *CoRR* abs/2402.10705 (2024). doi:10.48550/ARXIV.2402.10705.
- [26] A. B. Corrêa, A. G. Pereira, J. Seipp, Classical planning with LLM-generated heuristics: Challenging the state of the art with Python code, in: M. Ghassemi, P. Koniusz, R. Pascanu, M. N. Chen (Eds.), Proceedings of the 39th Annual Conference on Neural Information Processing Systems (NeurIPS'25), 2025, pp. 1–32. doi:10.48550/arXiv.2503.18809.
- [27] H. Kleine Büning, T. Lettmann, Propositional logic – deduction and algorithms, volume 48 of *Cambridge tracts in theoretical computer science*, Cambridge University Press, Cambridge, 1999. doi:10.5555/554728.
- [28] J. Marques-Silva, T. Glass, Combinational equivalence checking using satisfiability and recursive learning, in: Proceedings of the Conference on Design, Automation and Test in Europe (DATE'99), Assoc. Comput. Mach., New York, 1999, pp. 33–38. doi:10.1145/307418.307477.
- [29] H. Simonis, Sudoku as a constraint problem, in: CP Workshop on modeling and reformulating Constraint Satisfaction Problems, volume 12, 2005, pp. 13–27.
- [30] I. Lynce, J. Ouaknine, Sudoku as a SAT problem, in: S. Zilberstein (Ed.), Proceedings of the International Symposium on Artificial Intelligence and Mathematics (AI&Math'06), Fort Lauderdale, Florida, USA, 2006, pp. 1–9.
- [31] J. K. Fichte, S. Szeider, Reverse-engineering constraint models from CNFs: An agentic approach (supplementary material), Zenodo, 2026. URL: <https://doi.org/10.5281/zenodo.20762636>. doi:10.5281/zenodo.20762636, version 1.0.0.