Semi-declarative Language for Combinatorial Search

ZIYI YANG, National University of Singapore, Singapore
ILYA SERGEY, National University of Singapore, Singapore

Combinatorial search—finding solutions that meet constraints within an exponentially large space of candidates—underpins problems from hardware verification to scheduling and combinatorial design. The most successful approach in practice is Propositional Satisfiability (SAT) solving, but modelling a problem for a SAT solver requires manually translating high-level requirements into conjunctions of boolean clauses, a tedious step that sacrifices clarity and modularity. Answer Set Programming (ASP) offers a higher-level alternative, with a rule-based language whose variables and finite-domain reasoning yield more compact problem descriptions that can be automatically compiled into low-level solver input. Yet, ASP has its own shortcomings: its semantics, defined via a notion of *stable models*, is hard to build intuition for; it does not allow arbitrary first-order logic formulas as constraints; and its programs tend to be monolithic, making modular design and reuse difficult.

We propose SetLah!, a semi-declarative language for combinatorial search that addresses the shortcomings of both SAT and ASP. A SetLah! program is a sequence of stratified blocks, each containing rules that define the search space and arbitrary first-order logic constraints that prune it. This block structure enables modular problem decomposition, allowing for an intuitive semantics: candidate solutions are generated and filtered block by block. We built a compiler from SetLah! programs into ASP, allowing us to take full advantage of the existing efficient ASP solvers, while also automatically optimising the generated encodings. Our empirical evaluation demonstrates that SetLah! offers substantially more concise and intuitive specifications for common combinatorial search problems, and its compiled ASP encodings can significantly outperform SAT-based tools.

CCS Concepts: • **Theory of computation → Constraint and logic programming**; • **Computing methodologies → Logic programming and answer set programming**.

Additional Key Words and Phrases: Answer set programming, first-order logic, constraint solving

ACM Reference Format:
Ziyi Yang and Ilya Sergey. 2026. Semi-declarative Language for Combinatorial Search. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 399 (October 2026), 29 pages. https://doi.org/10.1145/3839531

1 Introduction

Combinatorial search—finding constrained solutions over large, discrete search spaces—is hard. It is hard in principle: unless P = NP, there is no polynomial-time algorithm that can solve all instances of NP-complete problems, the class to which most combinatorial problems belong. It is also hard in practice: while small problem instances might be amenable to brute-force enumeration (*e.g.*, the classical eight-queens problem), real-world instances such as hardware verification or large-scale scheduling are far beyond the reach of naïve search.

Guess-and-check is a dominant approach used to solve instances of NP-hard problems in practice, and Propositional Satisfiability (SAT) [13] is its most prominent incarnation: a problem instance is encoded as a set of boolean variables and constraints over them, and the solver searches for a satisfying truth assignment, *i.e.*, a mapping of all variables to true or false that meets every constraint. Thanks to highly efficient SAT solvers equipped with Conflict-Driven Clause Learning

Authors' Contact Information: Ziyi Yang, National University of Singapore, Singapore, Singapore, yangziyi@u.nus.edu; Ilya Sergey, National University of Singapore, Singapore, Singapore, ilya@nus.edu.sg.

ACM 2475-1421/2026/10-ART399
https://doi.org/10.1145/3839531

(CDCL) [82], advanced heuristics [60, 69], and established techniques for parallel [35], and incremental solving [31], many large-scale combinatorial problems—from formal verification [10] to planning [52] and automated theorem proving [48]—can now be solved effectively in practice.

Despite this efficiency, a significant *modelling gap* persists: to use a SAT solver, one must map the search space to boolean variables and encode the constraints as formulas in the conjunctive normal form (CNF). This process is tedious and error-prone, requiring careful handling of logical equivalences, as even minor changes in the problem specification can invalidate the entire encoding.

Answer Set Programming (ASP) [16, 71] presents an alternative to manual encoding of combinatorial search problems into logic programs, offering a declarative paradigm. Rather than requiring the user to flatten constraints into boolean clauses, ASP provides a rule-based language with first-order variables and constraints over finite domains, enabling compact and potentially reusable problem descriptions. For this reason, many ASP researchers describe it as "a modelling language for SAT" [61, 63]. Yet, despite its expressive capabilities, ASP poses significant barriers to entry for the newcomers. Rooted in logic programming (LP) and knowledge representation (KR), ASP relies on *stable model semantics* [44] to define the meaning of its programs. This semantics determines which sets of facts (called *stable models*) an ASP program admits, but the definition of stable models involves a non-trivial model theory that makes it difficult for non-experts to predict a program's behaviour or to understand why a particular solution is produced.

Although ASP provides a higher level of abstraction for encoding specific search problems than propositional formulas, it has been recognised that its declarative language does not fully close the modelling gap. Efforts to make stable models more accessible have proposed alternative formulations (as many as *thirteen* distinct definitions [62]), but these build on logical foundations such as Circumscription [67], equilibrium logic [49], and the situation calculus [76], which remain unfamiliar outside the LP/KR communities. Other approaches offer a simpler [25, 33] or refined syntax [27, 66] for ASP, but they either restrict the class of problems that can be modelled or require their own solver infrastructure, forgoing the optimisations of mature ASP systems.

We believe that what is needed is a language that combines the solver efficiency exploited by SAT with the modelling convenience promised by ASP, while avoiding the pitfalls of both. In our attempt to design it, we identified three core challenges that have hindered the broader adoption of ASP for combinatorial search (*cf.* Sec. 3), and that existing approaches do not adequately address:

(1) *Unintuitive semantics:* SAT and ASP both follow the guess-and-check paradigm, yet ASP's stable-model semantics is less immediate than SAT's truth-assignment semantics: the model-theoretic ASP semantics defines *which* sets of facts are valid under logic programs, but offers little intuition about *what* is guessed and *how* checking works, making it hard to understand.

(2) *Restricted first-order logic support:* Although ASP allows for using variables in its programs, it does not accept arbitrary first-order logic (FOL) formulas as constraints. Programmers can only describe specifications with restricted forms of FOL, sacrificing the clarity that first-order notation provides and complicating the modelling.

(3) *Lack of modularity:* ASP programs are monolithic: they declare the entire search space at once even when a problem naturally decomposes into dependent subproblems. Users must track implicit control flow to ensure correctness, making programs difficult to compose and reuse.

In this work, we propose a new programming language SetLah! (SET constraint LAnguage on choice Horn clauses) designed to address these challenges. The key idea behind SetLah! is to occupy a middle ground between ASP's monolithic declarativeness and SAT's low-level encodings. A SetLah! program is a sequence of blocks, ordered so that each block may only refer to predicates defined in earlier blocks. Within each block, the user writes rules that generate candidate solutions and arbitrary FOL formulas that constrain them. We call this design *semi-declarative*: blocks are

declarative internally—users specify *what* to search for, not *how*—but their sequencing gives explicit control over the problem decomposition. The semantics mirrors this structure: each block computes a set of candidate solutions via a fixpoint that branches on non-deterministic choices, then filters the result against the FOL constraints before passing it to the next block.

Contributions. In summary, this work makes the following contributions:

- We design SetLah!, a semi-declarative language for combinatorial search: it is *declarative* within each block, where users state guess-and-check specifications with arbitrary FOL constraints, and *semi*-declarative across blocks, where users explicitly sequence dependent subproblems for modular decomposition (Sec. 3).
- We give SetLah! an intuitive *branching fixpoint* semantics that generalises Datalog's least model construction to sets, and remains consistent with ASP's stable model semantics (Sec. 4).
- We develop a compilation pipeline from SetLah! to ASP that automatically splits, normalises, and optimises FOL constraints into efficient ASP encodings, reusing mature ASP solvers (Sec. 5).
- We evaluate SetLah! on Graeco-Latin squares, bounded model checking, and quantum circuit mapping, showing that its ASP output can significantly outperform SAT-based tools, while offering substantially more concise and easier-to-understand problem modelling (Sec. 6).

2 Background: SAT and ASP

Before introducing SetLah!, we outline the fundamental concepts of SAT and ASP that it builds upon. For this, we assume the reader's familiarity with standard propositional and first-order logic.

2.1 Propositional Satisfiability

The SAT problem asks whether there exists an interpretation (*i.e.*, an assignment of all Boolean variables) that satisfies a given propositional formula. SAT solvers typically operate on formulas in *Conjunctive Normal Form* (CNF). A CNF formula ϕ is a conjunction of clauses, where each clause is a disjunction of literals (a variable x or its negation $\neg x$).

$$\phi = \bigwedge_i C_i, \quad C_i = \bigvee_j l_{ij}$$

An interpretation I (a truth assignment to variables) satisfies ϕ, denoted $I \models \phi$, if it satisfies at least one literal in every clause. While SAT is the canonical NP-complete problem, modelling constraints in CNF often results in a loss of structural information: quantified or nested constraints must be flattened into individual clauses, obscuring the relationship between the original specification and its encoding (we will demonstrate this using the Latin square example in Sec. 3.1).

2.2 Datalog

While SAT operates on flat propositional formulas, ASP takes a different approach rooted in the logic programming tradition: problems are expressed as collections of rules that derive new facts from existing ones. Before introducing ASP proper, we recall Datalog [20, 56], the fragment of logic programming in which rules can only derive new facts, never retract them.

A *Horn clause* is a disjunction of literals containing at most one positive literal. A *definite clause* $A \vee \neg B_1 \vee \cdots \vee \neg B_m$ has exactly one, and is equivalently written as the rule A :- $B_1, \ldots, B_m$. A (propositional) *definite logic program* is a finite set of rules of the form A :- $B_1, \ \ldots, \ B_m$, where A (*head* of the rule) and each B_i in the *body* are atoms. For clarity, we present the propositional (ground) case first; the practically important extension to first-order rules with variables is discussed in Sec. 2.5. A rule with $m = 0$ is called a *fact* and is written simply as A. An *interpretation* I is a set of ground atoms, *i.e.*, atoms whose arguments are all constants rather than variables (by convention, lowercase names like a, b denote constants, while uppercase names like X, Y denote variables). We

say that the body $B_1, \ldots, B_m$ is satisfied in I iff $\{B_1, \ldots, B_m\} \subseteq I$. An interpretation I is a *model* of a program P when for every rule A `:-` $B_1, \ldots, B_m$. in P, if the body is satisfied in I then $A \in I$.

Many semantics exist for Datalog [56], among which the "operational" one based on the *immediate consequence operator* (T_P) is considered the most intuitive. T_P maps an interpretation I to the set of all heads A of rules in a program P whose bodies are satisfied in I. The *least model* of a definite program P is the least fixed point of T_P, obtained by iterating from the empty interpretation:

$$I_0 = \emptyset, \qquad I_{n+1} = T_P(I_n),$$

and taking interpretation I_k where $I_k = I_{k+1}$. This process always reaches a finite fixed point in Datalog; monotonicity of T_P ensures atoms are only added (facts are derived) and never retracted.

Example 2.1 (Reachability). *We illustrate the propositional account above with a small ground program that defines reachability in a directed graph.*

```
node(a). node(b). node(c). node(d).
edge(a,b). edge(b,c). edge(c,d).
reach(b).
reach(c) :- edge(b,c), reach(b).  % rule 1
reach(d) :- edge(c,d), reach(c).  % rule 2
```

Starting from the facts I_0, the T_P iteration adds `reach(c)` *(rule 1 fires), then* `reach(d)` *(rule 2 fires), and stabilises at $I_2 = I_0 \cup \{$reach(c), reach(d)$\}$, which is the least model.*

We should note that the `:-` in the rules is semantically different from the implication $\rightarrow$. Considering the tiny program `a :- b. b :- a.` without facts, its least model is $\emptyset$: the T_P iteration starts from $\emptyset$ and neither rule fires (each requires the other's head), so the fixpoint is reached immediately. In contrast, the logical implication $(b \rightarrow a) \wedge (a \rightarrow b)$ has two models: $\{a, b\}$ and $\emptyset$ as in SAT.

2.3 Answer Set Programming

ASP extends Datalog with *default negation* (or negation as failure). Whereas Datalog can only derive facts from other facts, default negation lets a rule fire when some atom *cannot* be derived, enabling programs to express choices and exclusions—the key ingredient for modelling combinatorial search. As with Datalog, we begin with the propositional (ground) formulation; the first-order extension will be discussed in Sec. 2.5. A (propositional) *normal logic program* consists of rules of the form:

$$A \texttt{ :- } B_1, \ldots, B_m, \texttt{ not } C_1, \ldots, \texttt{ not } C_n.$$

where A, each B_i, and each C_j are ground atoms. The default negation means that `not` C succeeds when C cannot be derived, realising the *closed-world assumption* (CWA) [75]: anything not entailed by the program is assumed false. This is natural for combinatorial search, where the goal is to enumerate exactly those facts that satisfy the constraints, with everything else excluded by default. The most common semantics of ASP is defined as a *stable model* semantics [44]. For a candidate interpretation I, the *reduct* P^I is the definite program obtained by removing every rule whose body contains `not` C for some $C \in I$ (those negations are "resolved" to false), and then stripping the remaining `not` C literals (resolved to true) from the surviving rules. An interpretation I is a *stable model* (or answer set) of P if I equals the least model of P^I. This semantics elegantly captures the closed-world assumption. However, it brings little intuition on *what* is being guessed and checked during the search process, in contrast to the model of SAT or even the least model of Datalog.

Note that a rule with an empty body (m=n=0) is simply a fact. An *integrity constraint* is a rule whose head is the distinguished atom `false` (conventionally omitted), written as:

$$\texttt{false :- } B_1, \ldots, B_m, \texttt{ not } C_1, \ldots, \texttt{ not } C_n.$$

Because no stable model may contain `false`, any candidate interpretation in which every B_i holds and no C_j holds would derive `false` and is therefore rejected. In other words, integrity constraints act as filters that discard unwanted candidate solutions.

Example 2.2. *Consider the program with unstratified negation (% are comments in programs below):*

```
a :- not b.  % rule1
b :- not a.  % rule2
```

The `rule1` *says that a is derived if b cannot be derived. To check if* $I_1 = \{a\}$ *is a stable model:*

- *Form the reduct* $P^{\{a\}}$*: the second rule is removed (since it contains* `not` *a and* $a \in I_1$*), leaving* `a`.
- *The least model of the reduct is* $\{a\}$*, which matches* I_1*, so* $I_1 = \{a\}$ *is a stable model.*

Similarly, $I_2 = \{b\}$ *is also a stable model; but* $I_3 = \{a, b\}$ *is not: from the reduct* $P^{\{a,b\}}$*, both rules are removed (since both contain negated atoms in* I_3*), leaving an empty program. Therefore, the program has exactly two stable models:* $\{a\}$ *and* $\{b\}$*. And if we add an integrity constraint* `:- a.` *to forbid a from being derived, then only* $\{b\}$ *remains as the unique stable model.*

This example shows the non-monotonicity of ASP: in pure Datalog (*i.e.*, without extensions like stratified negation [20]), the T_P operator can only add atoms, so every program has a unique least model determined by a single ascending chain of derivations. However, in ASP, unstratified negation creates mutual exclusions (*e.g.*, deriving *a* blocks *b* and vice versa), so the same program can admit multiple stable models—one for each consistent way of resolving the exclusions.

2.4 ASP Extensions: Choices and Aggregates

To facilitate practical modelling of combinatorial problems, modern ASP systems, such as Clingo [37], WASP [4], support syntax extensions that go beyond normal rules. These extensions operate on ground programs but are typically written in compact first-order notation. Two widely used constructs are *choice rules* and *aggregates*.

Choice Rules. A choice rule allows for the non-deterministic derivation of atoms.

$$\{a\} \texttt{ :- Body.}$$

This rule states that if the Body is satisfied, the atom *a may* be included in the model. This allows for "guessing" in combinatorial problems, so the solver could explore different atom combinations.

Example 2.3. *Consider the program with two rules:*

```
{ a }.
b :- a.
```

The choice rule on a results in two stable models: $\{a, b\}$ *(if a is chosen) and* $\emptyset$ *(if not).*

Curiously, the propositional formula $a \Leftrightarrow b$ has the same models as the ASP program above, but the space for guessing is $2 \times 2 = 4$ interpretations instead of 2 in ASP (where only *a* needs to be guessed). This highlights the "rules vs. constraints" distinction between ASP and SAT (*cf.*, Sec. 7).

Aggregates. ASP also supports so-called aggregate atoms (*e.g.*, #count, #sum, #min) in the body of program rules. An aggregate atom takes the form:

$$\text{op}\{V_1 : \text{Cond}_1; V_2 : \text{Cond}_2; \dots\} \prec \text{Bound}$$

where op is an aggregation function over the multiset of values V_i derived from the condition Cond_i, and $\prec$ is a comparison operator. Aggregates allow for compact representation of numerical or structural properties but can increase the complexity of the grounding and solving phases.

Example 2.4. *Consider the program:*

```
{a}. {b}. {c}.
:- #sum { 1 : a; 2 : b; 3 : c } != 3.
```

The first three choice rules allow any subset of $\{a, b, c\}$ to be chosen. The integrity constraint uses the aggregate #sum to enforce that the sum of the weights (1 for a, 2 for b, and 3 for c) must equal 3. The stable models of this program are: $\{a, b\}$ and $\{c\}$.

In modern ASP systems, aggregate atoms are allowed to appear in the head of rules, and the choice rules can be seen as a special case of head aggregates. For recursive aggregates, the semantics is somewhat less straightforward [65], so we only focus on body aggregates in our work.

2.5 First-Order Logic Programming and Grounding

In practice, both Datalog and ASP programs are written in a *first-order* syntax using variables rather than listing ground atoms explicitly. A first-order rule has the form:

$$H \texttt{ :- } L_1, \ldots, L_k,$$

where H (head atom) and each L_i (body literals, *i.e.*, positive or negative atoms) can contain variables (typically written in uppercase, *e.g.*, X, Y).

To compute models in either Datalog or ASP, first-order programs need to be *grounded*, *i.e.*, translated into equivalent propositional programs by replacing each variable with concrete constants. This can be done eagerly (all at once) or lazily (on demand during solving). A naïve grounding instantiates each rule with all possible substitutions of variables drawn from a finite domain (constructed from constants appearing in the program). For instance, the rules in Example 2.1 can be written compactly as `reach(Y) :- edge(X,Y), reach(X).` By lazily applying the substitutions during the fixed-point computation, only those two ground rules (rules with no remaining variables) relevant to the input graph are instantiated instead of all possible combinations of nodes.

Example 2.5 (An ASP program with all features above). *Consider the first-order ASP program:*

```
dom(1). dom(2). dom(3).
{ f(X) } :- dom(X).                    % choice rule
:- X=1, not f(X).                      % integrity constraint
:- #count { X : f(X) } != 2.           % aggregate constraint
```

The grounding process generates the following propositional rules:

```
dom(1). dom(2). dom(3).                 % facts (unchanged)
{f(1)}. {f(2)}. {f(3)}.                 % X -> 1, 2, 3
:- not f(1).                            % X -> 1
:- 2 != #count{1:f(1);2:f(2);3:f(3)}.   % X -> 1, 2, 3
```

From a guess-and-check perspective, the choice rule allows for any subset of $\{f(1), f(2), f(3)\}$ to be guessed. Then the later two constraints, firstly, enforce that $f(1)$ must be included in the model, and secondly, require exactly two of the $f(X)$ atoms to be true. The stable models of this program are: $\{f(1), f(2)\}$ and $\{f(1), f(3)\}$.

As the example above illustrates, grounding is a key step in the ASP workflow and a major distinction from SAT [61]: whereas SAT formulas are fed directly to a solver, ASP programs are first grounded and then solved (*e.g.*, the Clingo system contains Gringo and Clasp).

First-order rules make programs reusable (*e.g.*, the same `reach` rule set works for any graph) and enable optimisations such as semi-naïve evaluation [6]. However, grounding can also become a bottleneck [8, 43]: the size of the ground program can explode for rules with many variables or large domains, necessitating careful modelling (see the example in Sec. 3.1).

Besides grounding, a subtle limitation of ASP is that integrity constraints are the *only* mechanism for expressing logical properties of models, yet each one is restricted to the *negated existential* form: $\nexists \bar{X}.\,(B_1 \wedge \cdots \wedge B_m \wedge \neg C_1 \wedge \cdots \wedge \neg C_n)$. Not every FOL formula has this shape; for instance, $\forall X.\,\exists Y.\,\forall Z.\,(f(X) \rightarrow g(Z) \wedge Y > X)$, which mixes universal and existential quantifiers, cannot be expressed as a single integrity constraint. In the next section, we show how this restriction and the grounding bottleneck manifest in a concrete modelling task, and introduce SetLah!—a language designed to support arbitrary FOL constraints while compiling its programs to efficient ASP.

3 Overview of SetLah!

We now illustrate the design philosophy of SetLah! through a progression of motivating examples, using the *orthogonal Latin squares* problem to compare modelling in SAT, ASP, and SetLah!.

3.1 Euler's Mistaken Conjecture and Latin Squares

A *Latin square* of order n is an $n \times n$ array filled with integers from 1 to n, each occurring exactly once in each row and in each column. A *Graeco–Latin square* (equivalently, a pair of *orthogonal* Latin squares) of order n can be viewed as two Latin squares g (for Graeco) and l (for Latin) on the same $n \times n$ grid such that the superposition $(l(R,C), g(R,C))$ yields a unique ordered pair in every cell (no pair repeats). A Latin square and a Graeco–Latin square of order 3 are shown in Fig. 1. Determining whether such objects exist is a classic problem in combinatorial design: the constraints are simple but tightly coupled, and the search space grows rapidly with n. Leonhard Euler conjectured in 1782 that no Graeco–Latin square exists for orders $n \equiv 2 \pmod 4$, a claim that stood for nearly two centuries before being disproven [54]. Finding such counterexamples is a compelling benchmark for combinatorial search tools, so we use a general order n to illustrate modelling in SAT, ASP, and SetLah!, starting with a single Latin square and then adding orthogonality constraints.

3.2 A Latin Square in SAT and ASP

To illustrate the modelling gap between the approaches from Sec. 2, we compare how SAT, ASP, and SetLah! model the Latin square problem. The key modelling decision is shared by all three: introduce an atom $l(R,C,A)$, where $1 \le R, C, A \le n$, that holds when cell (R,C) contains number A. In SAT, the constraint for the square to be Latin is expressed as follows.

$$\bigwedge_{1\le R,C\le n}\left(\bigvee_{1\le A\le n} l(R,C,A)\right) \wedge \bigwedge_{\substack{1\le R,C\le n\\ 1\le A_1<A_2\le n}} (\neg l(R,C,A_1) \vee \neg l(R,C,A_2)) \qquad \text{(one number each cell)}$$

$$\wedge \bigwedge_{\substack{1\le R\le n\\ 1\le A\le n}}\left(\bigvee_{1\le C\le n} l(R,C,A)\right) \wedge \bigwedge_{\substack{1\le R\le n\\ 1\le C_1<C_2\le n\\ 1\le A\le n}} (\neg l(R,C_1,A) \vee \neg l(R,C_2,A)) \quad \text{(one row each col, number)}$$

$$\wedge \bigwedge_{\substack{1\le C\le n\\ 1\le A\le n}}\left(\bigvee_{1\le R\le n} l(R,C,A)\right) \wedge \bigwedge_{\substack{1\le C\le n\\ 1\le R_1<R_2\le n\\ 1\le A\le n}} (\neg l(R_1,C,A) \vee \neg l(R_2,C,A)) \quad \text{(one col each row, number)}$$

Each line enforces one of the three permutation properties via two parts: an *at-least-one* disjunction ("some A must fill this cell") and an *at-most-one* set of pairwise exclusions ("no two distinct values in the same cell"). The greyed-out at-least parts in the second and third lines are redundant, already implied by "each cell has exactly one symbol." In FOL terms, at-least-one is $\forall R, C.\,\exists A.\,l(R,C,A)$,

Fig. 1. A Latin square (left) and a Graeco–Latin square (right) of order $n = 3$.

while at-most-one is $\forall R, C, A_1, A_2.\ ((l(R, C, A_1) \wedge l(R, C, A_2)) \rightarrow A_1 = A_2)$; the $A_1 < A_2$ idiom in the CNF halves the clause count by breaking symmetry. Notice how the CNF encoding obscures the original specification: what is naturally a handful of quantified FOL statements becomes a wall of indexed clauses with no visible logical structure. One might wonder why not solve this with an SMT solver; as we discuss below at page 10, SMT solvers struggle with the combinatorial explosion inherent in finite-domain quantification, making them impractical for this class of problems.

In ASP, the difficulty is different. The "at-most-one" constraints map directly to integrity constraints, because "no two atoms hold simultaneously" is already a negated existential ($\neg\exists$). However, the "at-least-one" part—"some A must exist"—has a $\forall\exists$ structure that cannot be rewritten as $\neg\exists$. The standard workaround is to introduce an auxiliary predicate that witnesses the existential. The full ASP encoding is as follows (`dom/1` defines the domain $\{1, \ldots, n\}$):

```
dom(1). dom(2). dom(3). ... dom(n).
{ l(R, C, A) } :- dom(R), dom(C), dom(A).

latin_aux(R,C):- l(R,C,A).
:- dom(R), dom(C), not latin_aux(R,C).
:- dom(R), dom(C), l(R, C, A1), l(R, C, A2), A1 < A2.
:- dom(R), dom(A), l(R, C1, A), l(R, C2, A), C1 < C2.
:- dom(C), dom(A), l(R1, C, A), l(R2, C, A), R1 < R2.
```

With the stable model semantics, the predicate `latin_aux(R, C)` is derived when there exists some A such that `l(R, C, A)` holds, then the following integrity constraint captures the at-least one condition with the help of `latin_aux`. The example stable model corresponding to the Latin square in Fig. 1 (left) contains following atoms (with `..` omitting the enumeration for brevity):

```
dom(1..3), latin_aux(1..3,1..3), l(1,1,1), l(1,2,2), l(1,3,3),
l(2,1,2), l(2,2,3), l(2,3,1), l(3,1,3), l(3,2,1), l(3,3,2)
```

3.3 Defining a Latin Square in SetLah!

As we have just seen, both SAT and ASP force the programmer to manually decompose intuitive mathematical properties into low-level encoding idioms. SetLah! eliminates this burden: the user writes the constraints as FOL formulas, and the compiler handles the rest:

```
[dom(1). dom(2). dom(3). ... dom(n).]          % block 1: domain
[
    l(R, C, A) ?:- dom(R), dom(C), dom(A).     % guess: which cells hold which numbers?
    ∀R, C. (∃A. l(R, C, A) ∧                   % check: exactly one number per cell
          ∀A1, A2. (l(R, C, A1) ∧ l(R, C, A2) → A1 = A2))
    ∀R, A, C1, C2. (l(R, C1, A) ∧ l(R, C2, A) → C1 = C2)   % check: one per row
    ∀C, A, R1, R2. (l(R1, C, A) ∧ l(R2, C, A) → R1 = R2)   % check: one per column
] % block 2: search + constraints
```

The program consists of two blocks (delimited by square brackets). Block 1 defines the domain.[1] Block 2 contains a *choice rule* (`?:-`) that guesses which `1` atoms to include, followed by three FOL constraints whose purpose is to filter out invalid guesses. The constraints read exactly like their mathematical specification: "every cell has exactly one number" (at-least-one and at-most-one combined), "each number appears at most once per row", and "each number appears at most once per column". Notice that, in contrast with the SAT and ASP encodings, in SetLah! no manual decomposition or auxiliary predicates are needed. This directly addresses challenges (1) and (2) from Sec. 1. For (1), the block structure makes the search visible: block 1 establishes the input (domain facts), block 2 separates the *guess* (choice rule) from the *check* (FOL constraints), so one can see at a glance what is assumed, searched, and verified—unlike ASP, where stable model semantics interleaves all three. For (2), the constraints are written as unrestricted FOL formulas rather than decomposed into the restricted $\neg\exists$ form that ASP integrity constraints require.

Informal semantics. Each block operates on a *set of candidate interpretations* produced by the previous block (detailed in Sec. 4.2.2). Block 1 contains only facts, so starting from $\mathcal{M}_0 = \{\emptyset\}$ its fixpoint yields a single interpretation $\mathcal{M}_1 = \{\{dom(1), \ldots, dom(n)\}\}$. Block 2 takes $\mathcal{M}_1$ as input: the choice rule (`?:-`) *branches* each interpretation by optionally adding `1(R,C,A)` for every combination of R, C, A, producing all 2^{n^3} candidate subsets. The FOL constraints then *filter* this set, keeping only the interpretations that satisfy every formula—these are the *answer sets*, including the one corresponding to Fig. 1 (left). A useful analogy is *monadic set comprehensions*: choice rules act as *generators* that enumerate candidates, FOL constraints act as *guards* that filter them, and blocks compose sequentially like monadic bind—each block's output feeds the next. Both deterministic (`:-`) and non-deterministic rules are handled by the same fixpoint, generalising Datalog's T_P to sets of interpretations—hence we call SetLah! a *set-constraint* language.

Compilation to ASP. The semantics above does not scale in practice, as it enumerates all $2^{|\mathcal{A}|}$ candidates. The key observation is that it is consistent with ASP's stable model semantics when FOL constraints are translated into equivalent integrity constraints, so the computation can be delegated to well-optimised ASP systems [4, 37]! The SetLah! compiler performs this translation exactly, block by block, in two phases (detailed in Sec. 5). First, *conjunction splitting* (Sec. 5.3) decomposes each FOL constraint into smaller conjuncts to reduce grounding cost, *e.g.*, the "one number per cell" formula with 5 variables splits into two parts needing only $n^3 + n^4$ ground instances instead of n^5. Second, *quantifier elimination* (Sec. 5.4) converts each conjunct into the restricted $\neg\exists$ form required by ASP, introducing auxiliary predicates or aggregates to handle arbitrary FOL formulas not directly expressible as integrity constraints. After these translations, the input SetLah! program becomes an ASP program that can be solved by an off-the-shelf ASP solver.

3.4 From Latin to Graeco–Latin Squares

To obtain a Graeco–Latin square, we pair two Latin squares g and l of the same order and require *orthogonality*: every pair (A, B) must appear exactly once across the grid. In FOL, such constraints can be expressed naturally as: $\forall A, B.\exists R, C.(l(R, C, A) \wedge g(R, C, B))$.

For modelling the property in CNF, we need auxiliary atoms $p(R, C, A, B)$ for $l(R, C, A) \wedge g(R, C, B)$,[2] leading to the following encoding where three sub-CNFs are used for the bi-implication.

[1] In the specification of the full SetLah! language (Sec. 4), domains are declared via typed signatures and regarded as the input of SetLah! programs; here we use explicit unary `dom/1` facts for simplicity.

[2] It is possible to project out the existential quantifications of `R,C` but the corresponding CNF becomes more complex.

$$\bigwedge_{1\le A,B\le n}\left(\bigvee_{1\le R,C\le n} p(R,C,A,B)\right)\wedge \bigwedge_{1\le R,C,A,B\le n}((\neg l(R,C,A)\vee\neg g(R,C,B)\vee p(R,C,A,B)))$$
$$\wedge \bigwedge_{1\le R,C,A,B\le n}((\neg p(R,C,A,B)\vee l(R,C,A)))\wedge \bigwedge_{1\le R,C,A,B\le n}((\neg p(R,C,A,B)\vee g(R,C,B)))$$

In ASP, the inner existential can be handled in two ways, mirroring the auxiliary-predicate pattern:

```
% With auxiliary predicate as in SAT
product(A, B) :- g(R, C, A), l(R, C, B).
:- dom(A), dom(B), not product(A, B).

% Or alternatively with Aggregates
:- dom(A), dom(B), #count {R, C : g(R, C, A), l(R, C, B) } = 0.
```

In SetLah!, the block structure naturally decomposes the Graeco–Latin square into dependent subproblems: one block defines the first Latin square `l`, and a subsequent block defines the second square `g` while imposing orthogonality.

```
[dom(1). ... dom(n).]                          % block 1: domain
[l(R,C,A) ?:- dom(R), dom(C), dom(A). ...]     % block 2: first square
[g(R,C,B) ?:- dom(R), dom(C), dom(B). ...      % block 3: second square
 ∀A, B. ∃R, C. (l(R,C,A) ∧ g(R,C,B))]         %      + orthogonality
```

The FOL constraint in block 3 directly references predicate `l` from block 2, expressing the cross-square dependency declaratively. The SetLah! compiler translates this into either of the ASP encodings shown above: aggregates by default, since they are model-preserving (the auxiliary-predicate encoding introduces new atoms; *cf.* Sec. 5.4). Alternatively, the user may add a separate block 4 defining `product(A,B)` explicitly, if that reads more naturally.

This flexibility in block-based task decomposition directly addresses challenge (3) from Sec. 1: the user decides the subproblem structure based on the problem's natural dependencies, while the compiler handles the low-level ASP encoding—enabling the kind of modular decomposition that monolithic ASP programs lack. More broadly, the ease of modelling in SetLah! stems from a one-to-one correspondence between the constraints in the user's mind and the FOL formulas in the program; in SAT and ASP, bridging that gap is left as manual work for the programmer.

Why not SMT? One might argue that all these constraints can be expressed directly in FOL and solved using modern Satisfiability Modulo Theories (SMT) solvers, such as Z3 [24], cvc5 [7], which support unrestricted quantifiers over theories like integer arithmetic. It is, however, well-known that SMT rarely scales well for combinatorial problems with non-trivial quantifier structure, due to lazy quantifier instantiation [36, 78] (*cf.* Sec. 7). Specifically, on the Graeco-Latin square problem, SMT solvers struggle with $n \geq 6$ (unless the user manually grounds the atoms), while SAT and ASP solvers handle n up to 10 in reasonable time with the same encoding (*cf.* Sec. 6.2).

4 Language Design of SetLah!

We present the syntax and the formal semantics of SetLah! in two stages. We first introduce *Positive Choice Logic Programming* (PCLP), a small kernel language that takes *choice* as a primitive—in contrast to ASP, which derives non-determinism from negation-as-failure. A program in PCLP takes a single answer set as its input and produces a set of answer sets via a least fixpoint computation. We then define SetLah! as *stratified PCLP*: a program is a sequence of PCLP blocks whose answer sets are chained together block by block.

PCLP (kernel language)

t	::=	$X \mid c \mid f(t_1, \ldots, t_k)$	term
a	::=	$p(t_1, \ldots, t_k)$	atom
L	::=	$a \mid t_1 \; op \; t_2$	body literal
φ	::=	$a \mid \top \mid \bot \mid \neg\varphi$	FOL formula
	|	$\varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$	
	|	$\varphi_1 \rightarrow \varphi_2$	
	|	$\forall X.\, \varphi \mid \exists X.\, \varphi$	
	|	$t_1 \; op \; t_2$	comparison
r_d	::=	$H \leftarrow L_1, \ldots, L_m$	definite rule
r_c	::=	$\{H\} \leftarrow L_1, \ldots, L_m$	choice rule
P	::=	$(\mathcal{R},\ C,\ \Phi)$	program

where $op \in \{=, \neq, <, \leq, >, \geq\}$.

SetLah! (stratified PCLP)

d	::=	`ExtDOM` $D : \tau$	external domain
	|	`DefDOM` $D : \tau$ $-\{c_1, \ldots, c_k\}$	domain definition
	|	`ExtPred` $p : \tau_1 \times \cdots \times \tau_k$	external predicate
	|	`Const` $c : \tau$	constant
B	::=	$(\Sigma,\ P)$	block
Π	::=	$(\mathcal{D},\ \langle B_1, \ldots, B_n \rangle)$	program

Fig. 2. Grammar of PCLP and SetLah!.

4.1 PCLP: The Kernel Language

4.1.1 Syntax. Fig. 2 gives the grammar of PCLP and SetLah!. Variables are in upper case ($X, Y, \ldots$) and constants in lower case ($a, b, \ldots$); predicate symbols are lower case ($p, q, \ldots$). A PCLP program is a triple $P = (\mathcal{R}, C, \Phi)$, where *definite rules* $\mathcal{R}$ are Horn clauses that deterministically derive atoms, *choice rules* C are called *choice Horn clauses*—clauses whose head is optional—that introduce non-deterministic branching, and *constraints* Φ are closed FOL sentences (no free variables) that filter the search space. Thus, PCLP lifts SAT's propositional guess-and-check pattern to finite first-order structures: the guessed objects are ground atoms, deterministic rules derive all positive consequences of those guesses, and FOL constraints perform the check. The implication $\varphi_1 \rightarrow \varphi_2$ is syntactic sugar for $\neg\varphi_1 \vee \varphi_2$. All rule bodies are *positive* (contain no negations besides comparisons); this restriction ensures monotonicity of the rule operators defined below, which in turn guarantees a well-defined fixpoint. We write $\leftarrow$ in formal definitions and `:-` (resp. `?:-`) in the concrete syntax for definite (resp. choice) rules.

4.1.2 Semantics. We define the semantics of P relative to a single *input answer set* I (a set of ground atoms). Let $\mathcal{A}$ be the *Herbrand base* of P, *i.e.*, the set of all ground atoms constructible from the predicates and constants in P and I. We write σ for a substitution of variables to ground terms; for a rule body $\bar{L} = L_1, \ldots, L_m$, write $\sigma(\bar{L})$ for $\{\sigma(L_1), \ldots, \sigma(L_m)\}$.

Rule operator. For a rule $r \in \mathcal{R} \cup C$, the *rule operator* T_r acts differently on the candidate set depending on whether r is definite or a choice:

$$T_r(\mathcal{S}) = \begin{cases} \left\{ J \cup \{\sigma(H) \mid \sigma(\bar{L}) \subseteq J\} \;\middle|\; J \in \mathcal{S} \right\} & r = (H \leftarrow \bar{L}) \in \mathcal{R} \\ \mathcal{S} \;\cup\; \left\{ J \cup \{\sigma(H)\} \;\middle|\; J \in \mathcal{S},\ \sigma(\bar{L}) \subseteq J \right\} & r = (\{H\} \leftarrow \bar{L}) \in C \end{cases}$$

The two cases capture the semantic distinction: a definite rule *updates* each candidate J in place by adding all applicable consequences (mandatory derivation), while a choice rule *grows* the candidate set by adding optional branches $J \cup \{\sigma(H)\}$ alongside the existing J. Built-in comparisons in $\bar{L}$ are evaluated directly over the substitution σ rather than checked against J; since their truth value depends only on σ, they do not affect monotonicity of T_r.

Candidate set. Starting from $\{I\}$, we apply rule operators one at a time in any order. The *candidate set* of P from I is the fixpoint reached when no operator changes the set:

$$\mathcal{S}^*(P, I) = \mathrm{lfp}\big(\{T_r\}_{r \in \mathcal{R} \cup C},\ \{I\}\big).$$

Since every T_r for $r \in \mathcal{R}$ maps each J to a superset of J (adding atoms), and T_r for $r \in C$ only adds elements, the set stabilises. Every element of $\mathcal{S}^*(P, I)$ is $\mathcal{R}$*-closed, i.e.*, closed under all definite rules in $\mathcal{R}$: any J with an unfired definite rule is updated by T_r before the fixpoint is reached.

Confluence and convergence. The fixpoint is independent of the order in which rules are applied, so the semantics is deterministic and implementations may evaluate rules in any order. Formally, this follows from a diamond property: for any $r_1, r_2 \in \mathcal{R} \cup C$,

$$T_{r_1}(\mathcal{S}) \subseteq T_{r_2}(T_{r_1}(\mathcal{S})) \quad \text{and} \quad T_{r_2}(\mathcal{S}) \subseteq T_{r_1}(T_{r_2}(\mathcal{S}))$$

The inclusions hold because both cases of T_r only add atoms—they never remove them—so applying r_2 after r_1 preserves every interpretation reachable via r_1, and may only enable further firings. Hence all orderings converge to the same $\mathcal{S}^*(P, I)$. When $\mathcal{A}$ is finite, convergence in finitely many steps is guaranteed; SetLah! enforces finiteness via domain declarations (Sec. 4.2).

Answer sets of P. Since every $M \in \mathcal{S}^*(P, I)$ is already $\mathcal{R}$-closed, the *answer sets* of P from I are those candidates whose atoms satisfy every FOL constraint—that is, the constraints in Φ act as filters over the search space generated by $\mathcal{R}$ and C:

$$\mathcal{M}(P, I) = \big\{\, M \in \mathcal{S}^*(P, I) \;\big|\; \forall\, \varphi \in \Phi,\ M \models \varphi \,\big\}.$$

For a set of input answer sets $\mathcal{I}$, define $\mathcal{M}(P, \mathcal{I}) = \bigcup_{I \in \mathcal{I}} \mathcal{M}(P, I)$.

When $C = \emptyset$, the fixpoint $\mathcal{S}^*(P, I)$ is the singleton containing the unique minimal $\mathcal{R}$-closure of I, recovering Datalog semantics seeded from I. Conversely, PCLP captures NP: each ground choice rule $\{a\} \leftarrow$ corresponds to a Boolean variable (include a or not), and FOL constraints subsume CNF clauses in SAT.

Example 4.1. *Consider P with the following components and input:*

$$\mathcal{R} = \{b(X) \leftarrow a(X)\}, \qquad C = \{\{a(X)\} \leftarrow d(X)\}, \qquad \Phi = \emptyset, \qquad I = \{d(1),\ d(2)\}.$$

Applying T_r for each rule from $\{I\}$: (A) T_r for $r_{\mathcal{R}} = b(X) \leftarrow a(X)$: no $a(\cdot)$ in I, so I maps to itself; (B) T_r for $r_C = \{a(X)\} \leftarrow d(X)$: adds $I \cup \{a(1)\}$ and $I \cup \{a(2)\}$ as new branches; (C) T_r for $r_{\mathcal{R}}$ on the new branches: $I \cup \{a(1)\}$ becomes $I \cup \{a(1), b(1)\}$; similarly $I \cup \{a(2)\}$ becomes $I \cup \{a(2), b(2)\}$; (D) T_r for r_C on updated branches: adds $I \cup \{a(1), a(2)\}$; then T_r for $r_{\mathcal{R}}$ extends it to $I \cup \{a(1), a(2), b(1), b(2)\}$. The fixpoint is $\mathcal{S}^*(P, I) = \big\{\, I,\ \ I \cup \{a(1), b(1)\},\ \ I \cup \{a(2), b(2)\},\ \ I \cup \{a(1), a(2), b(1), b(2)\} \,\big\}$, and all four candidates are $\mathcal{R}$-closed. Since $\Phi = \emptyset$, every candidate passes the filter: $\mathcal{M}(P, I) = \mathcal{S}^*(P, I)$.

In summary, PCLP provides the core mechanism—deterministic derivation, non-deterministic branching, and FOL filtering—but leaves open how to bound the search space and decompose multi-stage problems. SetLah! addresses both, as we will discuss next.

4.2 SetLah! as Stratified PCLP

SetLah! extends PCLP with two features that make it a practical language: (i) *declarations* that specify domains, external input predicates, and constants, bounding the Herbrand base finitely so that the fixpoint always terminates, and (ii) *stratification* into an ordered sequence of blocks, so that a multi-stage problem can be decomposed into dependent sub-problems solved one after another.

4.2.1 Syntax. A SetLah! program $\Pi = (\mathcal{D}, \langle B_1, \ldots, B_n \rangle)$ consists of *domain declarations* $\mathcal{D}$ followed by a sequence of *blocks* (Fig. 2, right).

Declarations. Each declaration in $\mathcal{D}$ introduces a named domain, predicate, or constant with an associated type τ (such as INT or a product type):

- `ExtDOM` $D : \tau$. declares an *external* domain whose constants are supplied as input (*e.g.*, node identifiers of a graph).
- `DefDOM` $D : \tau - \{ c_1, \ldots, c_k \}$. declares an *internal* domain with explicitly listed constants $c_1, \ldots, c_k$ (*e.g.*, `Set` : 1..9 for Sudoku digits).
- `ExtPred` $p : \tau_1 \times \cdots \times \tau_k$. declares an *external predicate* whose extension is supplied as input (*e.g.*, an adjacency relation edge : Node × Node).
- `Const` $c : \tau$. declares a named constant of type τ (*e.g.*, a parameter such as the square size n).

Every variable is typed by a declared domain, and each domain contains finitely many constants, so the set of possible ground atoms is finite. This bounds the Herbrand base $\mathcal{A}_{\mathcal{D}}$ and guarantees that the fixpoint computation always terminates. Each predicate is tied to a fixed tuple of domains which is inferred from rule bodies (if not external), enabling static type checking.

Blocks. Each block $B_i = (\Sigma_i, P_i)$ pairs a set of *output predicates* Σ_i with a PCLP program $P_i = (\mathcal{R}_i, C_i, \Phi_i)$, where Φ_i contains arbitrary FOL sentences over the predicates visible to B_i.

Stratification of a SetLah! program. Π is *well-formed* if for every block B_i:

- every predicate in the body of a definite rule in $\mathcal{R}_i$ belongs to $\Sigma_1 \cup \cdots \cup \Sigma_{i-1}$ or to Σ_i, so definite rules may refer to their own block's predicates, enabling recursive definitions;
- every predicate in the body of a choice rule in C_i belongs strictly to $\Sigma_1 \cup \cdots \cup \Sigma_{i-1}$ (no self-referential guessing—a choice can only branch over facts already established, preventing circular dependencies in the search space);
- every predicate in a constraint in Φ_i belongs to $\Sigma_1 \cup \cdots \cup \Sigma_i$.

The second condition is one key design choice of SetLah!, which enables the decomposition of a multi-stage problem into a sequence of dependent sub-problems. Considering that we want to express: *a person can be a faculty or a student, and a student can be a graduate or an undergraduate*, we can write the following SetLah! program:

```
[   faculty(P) ?:- dom_person(P).
    student(P) ?:- dom_person(P).
    ∀P. ¬(faculty(P) ∧ student(P)).]
[   graduate(S) ?:- student(S).
    undergraduate(S) ?:- student(S).
    ∀S. ¬(graduate(S) ∧ undergraduate(S)).]
```

Unlike the example in Sec. 3.4 where the blocks of two squares are independent, the second block here branches over `student/1` atoms produced by the first block. Such dependency is common in combinatorial problems, *e.g.*, the program above is partially from the benchmark in Tab. 4.

4.2.2 Semantics. The semantics of Π processes blocks left to right, feeding each block's output into the next. Let I_0 denote the set of ground facts supplied by external predicates (`ExtPred`) and constants (`Const`), and let $\mathcal{M}_0 = \{I_0\}$. Each successive block $B_i = (\Sigma_i, P_i)$ takes the answer sets produced so far and computes:

$$\mathcal{M}_i = \mathcal{M}(P_i, \mathcal{M}_{i-1}) = \bigcup_{I \in \mathcal{M}_{i-1}} \mathcal{M}(P_i, I),$$

using the Herbrand base $\mathcal{A}_{\mathcal{D}}$ (finite by domain declarations). The *answer sets* of Π are the result after all n blocks: $\mathcal{M}(\Pi) = \mathcal{M}_n$. In other words, each block generates and filters its own candidates, then passes the survivors as input to the next block.

Algorithm 1 TranslateConstraints(Φ_i): Translate FOL constraints of block B_i

Input: FOL constraints Φ_i from block B_i
Output: Auxiliary rules $\mathcal{R}_{\text{aux}}$ and integrity constraints $\mathcal{IC}$

$\mathcal{IC} \leftarrow \emptyset$; $\mathcal{R}_{\text{aux}} \leftarrow \emptyset$
for each $\varphi \in \Phi_i$ **do**
  $splits \leftarrow \textsc{Split}(\varphi)$ ▷ Phase 1: conjunction splitting
  for each $\varphi' \in splits$ **do**
    $E \leftarrow \textsc{Normalize}(\varphi')$ ▷ Set of logically equivalent formulas
    if $\exists \psi \in E$ in IC form **then**
      $\mathcal{IC} \leftarrow \mathcal{IC} \cup \{\text{ToIC}(\psi)\}$ ▷ Phase 2: direct conversion
    else
      $(\psi, R) \leftarrow \textsc{EliminateQuantifiers}(E)$ ▷ Phase 2: QE with auxiliary
      $\mathcal{R}_{\text{aux}} \leftarrow \mathcal{R}_{\text{aux}} \cup R$
      $\mathcal{IC} \leftarrow \mathcal{IC} \cup \{\text{ToIC}(\psi)\}$
return $(\mathcal{R}_{\text{aux}},\ \mathcal{IC})$

Relationship to ASP. Three structural differences between SetLah! and ASP are worth noting: (1) PCLP takes choice as a primitive instead of deriving non-determinism from negation-as-failure, but the two mechanisms have the same expressiveness [33, 72, 88]; (2) PCLP's FOL constraints Φ admit arbitrary quantifiers and connectives, whereas ASP's integrity constraints are restricted to denials ($:- L_1, \ldots, L_m$), but both equivalently filter on the answer sets; and (3) SetLah!'s block structure is less monolithic than ASP's flat rule set, while the chaining semantics $\mathcal{M}_i = \mathcal{M}(P_i, \mathcal{M}_{i-1})$ remains consistent with stable model semantics. The latter two points are given by the translation from SetLah! to ASP, which we will discuss next.

5 Translating SetLah! Programs to ASP

One of the main technical contributions of this work is a compiler that translates SetLah! programs into efficient ASP encodings, automatically bridging the gap between arbitrary FOL constraints and the restricted integrity-constraint form that ASP solvers require. Without this translation, users would have to decompose every constraint by hand—precisely the modelling burden that SetLah! is designed to eliminate. We describe how the compiler translates a SetLah! program Π into an equivalent Clingo ASP program. The translation operates block by block on each $P_i = (\mathcal{R}_i, C_i, \Phi_i)$. For space reasons, all proofs in this section are deferred to the supplementary material.

5.1 Translating a Block

Rules. The definite and choice rules translate directly into Clingo syntax: $H \leftarrow \bar{L}$ becomes `H :- L1, ..., Lm.` and $\{H\} \leftarrow \bar{L}$ becomes `{H} :- L1, ..., Lm.`. Since all bodies are positive and the Herbrand base $\mathcal{A}_{\mathcal{D}}$ is finite, no further transformation is needed.

FOL constraints. Each $\varphi \in \Phi_i$ must be translated into a set of ASP *integrity constraints* (ICs), possibly introducing auxiliary predicates. An IC has the form:

$$:- \ \ell_1, \ldots, \ell_m.$$

where each ℓ_j is an *ASP body element*: an atom, the default negation of an atom (`not a`), a built-in comparison, or an aggregate expression—strictly more general than the body literals L in PCLP (Sec. 4.1), which are restricted to atoms and comparisons. Semantically, the IC forbids any answer set from satisfying $\ell_1 \wedge \cdots \wedge \ell_m$. When the constraint contains variables $\bar{X}$, grounding replaces

Table 1. Equivalence rewriting rules. One-way rules (→) strictly reduce formula size; bidirectional rules (↔) rearrange structure. Meta-variables φ, ψ, χ range over arbitrary formulas.

Scope	Category	**Rule** (dual means swapping $\vee$ and $\wedge$)	**Effect on formula**
Propositional	Identity	$\varphi \wedge \top \;\to\; \varphi, \quad \varphi \vee \bot \;\to\; \varphi$	Reduces size
	Annihilation	$\varphi \wedge \bot \;\to\; \bot, \quad \varphi \vee \top \;\to\; \top$	Reduces size
	Idempotence	$\varphi \wedge \varphi \;\to\; \varphi$, dual	Reduces size
	Double negation	$\neg\neg\varphi \;\to\; \varphi$	Reduces size
	Commutativity	$\varphi \wedge \psi \;\leftrightarrow\; \psi \wedge \varphi, \quad \varphi \vee \psi \;\leftrightarrow\; \psi \vee \varphi$	Reorders operands
	Associativity	$\varphi \wedge (\psi \wedge \chi) \;\leftrightarrow\; (\varphi \wedge \psi) \wedge \chi$, dual	Regroups operands
	De Morgan	$\neg(\varphi \wedge \psi) \;\leftrightarrow\; \neg\varphi \vee \neg\psi$, dual	Redistributes $\neg$
	Distribution	$\varphi \wedge (\psi \vee \chi) \;\leftrightarrow\; (\varphi \wedge \psi) \vee (\varphi \wedge \chi)$, dual	Duplicates/factors
First-order	Quantifier negation	$\neg\forall x\, \varphi \;\leftrightarrow\; \exists x\, \neg\varphi$	Moves $\neg$
	Quantifier commutation	$Q\,x.\, Q\,y.\, \varphi \;\leftrightarrow\; Q\,y.\, Q\,x.\, \varphi \;\; (Q \in \{\forall, \exists\})$	Reorders quantifiers
	Quantifier distribution	$(\forall x\, \varphi) \wedge (\forall x\, \psi) \;\leftrightarrow\; \forall x\, (\varphi \wedge \psi)$ $(\exists x\, \varphi) \vee (\exists x\, \psi) \;\leftrightarrow\; \exists x\, (\varphi \vee \psi)$	Scopes/pulls quantifier
	Prenex conversion	$(\forall x\, \varphi) \vee \psi \;\leftrightarrow\; \forall x\, (\varphi \vee \psi) \;\; (x \notin \mathrm{FV}(\psi))$ $(\exists x\, \varphi) \wedge \psi \;\leftrightarrow\; \exists x\, (\varphi \wedge \psi) \;\; (x \notin \mathrm{FV}(\psi))$	Scopes/pulls quantifier

them with all possible constant tuples (*cf.* Sec. 2), so the IC corresponds to the negated existential:

$$\neg\exists \bar{X}.\ (\ell_1 \wedge \cdots \wedge \ell_m)$$

We call this the *IC form*: a negated existential over a conjunction of ASP body elements. The goal is to convert each $\varphi \in \Phi_i$ into a set of ICs equivalent over the finite Herbrand base $\mathcal{A}_{\mathcal{D}}$. Since arbitrary FOL sentences are not directly in IC form—for instance, a formula like $\forall X.\, \exists Y.\, p(X, Y)$ contains a quantifier alternation that has no direct IC encoding—the compiler performs a series of equivalence-preserving rewrites to bridge this gap (Sec. 5.2). Also noticed that an ASP grounder instantiates programs over the Cartesian product of its variables' domains, a high-level goal of the translation is to reduce the number of variables in each emitted IC.

The translation proceeds in two phases. *Phase 1* (conjunction splitting, Sec. 5.3) decomposes each $\varphi \in \Phi_i$ into independent conjuncts, reducing grounding cost. *Phase 2* (conversion to IC form, Sec. 5.4) rewrites each conjunct into one or more ICs by applying standard equivalences and, when necessary, quantifier elimination (QE) via auxiliary predicates or aggregates. Together with the directly translated rules from $\mathcal{R}_i$ and C_i, the output forms a complete ASP encoding of the block P_i. Algorithm 1 summarises the procedure.

5.2 Equivalence Rewriting Rules

Both phases rely on a fixed set of equivalence rewriting rules (Tab. 1) applied to FOL formulas and their subformulas. Each rule is either *one-way* (→), strictly reducing formula size and applicable at most $\mathit{size}(\varphi_0)$ times, or *bidirectional* (↔), rearranging structure without changing size. A key invariant is immediate from inspection of the rules:

Observation 5.1 (Atom and variable invariance). *No rule in Tab. 1 introduces fresh atoms or variables. For every formula ψ reachable from φ_0,* $\mathrm{Atoms}(\psi) \subseteq \mathrm{Atoms}(\varphi_0)$ *and* $\mathrm{Vars}(\psi) \subseteq \mathrm{Vars}(\varphi_0)$.

This invariant is the key for proving that the rewriting system explores only finitely many formulas:

Theorem 5.2 (Finiteness of the reachable set). *For every formula φ_0 of finite size, the set of formulas reachable from φ_0 by applying rules from Tab. 1 is finite.*

By Observation 5.1, every reachable formula is built from the same atoms and variables as φ_0. Since each remaining source of variation is bounded—(1) one-way rules strictly decrease formula size, (2) De Morgan and double negation yield at most $2^{size(\varphi_0)}$ negation placements, (3) distribution cannot exceed the DNF/CNF size ceiling, and (4) quantifier rules permute finitely many quantifiers over finitely many positions—the total number of reachable formulas is finite.

5.3 Phase 1: Conjunction Splitting

Given $\varphi \in \Phi_i$, the splitting phase finds a logically equivalent φ' such that $\varphi' = \bigwedge_j \psi_j$, and then processes each conjunct independently. Splitting under universal quantifiers—$\forall X.\,(\psi_1 \wedge \psi_2) \equiv (\forall X.\,\psi_1) \wedge (\forall X.\,\psi_2)$—and through implications—$\varphi_1 \rightarrow (\psi_1 \wedge \psi_2) \equiv (\varphi_1 \rightarrow \psi_1) \wedge (\varphi_1 \rightarrow \psi_2)$—are obtained directly from the reachable set (Theorem 5.2). In principle, if splitting is not confluent (*i.e.*, if two irreducible forms are not bijectively equivalent), the compiler can explore all possible splits and choose the one that minimises grounding cost. In practice, the splitting is always confluent in our case studies. The general confluence of the rewriting system remains an open research question [21], and we leave a formal proof of this to future work.

By Observation 5.1, every conjunct ψ_j satisfies $\mathrm{Vars}(\psi_j) \subseteq \mathrm{Vars}(\varphi)$, so splitting never increases grounding cost per conjunct. In practice, splitting can reduce grounding cost substantially (as in the Graeco-Latin squares of Sec. 3.1) and can also simplify the subsequent conversion to IC form.

5.4 Phase 2: Conversion to IC Form

While Phase 1 rewrites within pure FOL, Phase 2 targets ASP: the rewriting rules are extended with ASP-specific constructs—predicates, negations and aggregate expressions—to convert each conjunct into IC form. As Algorithm 1 shows, each irreducible formula φ' from Phase 1 is normalised again by applying standard equivalences to obtain a finite set E of logically equivalent formulas. If any $\psi \in E$ is already in IC form ($\neg\exists\bar{X}.\,(\ell_1 \wedge \cdots \wedge \ell_m)$), it is emitted directly as an ASP constraint.

Otherwise, the formula contains a quantifier alternation that cannot be resolved by propositional rewriting alone, and the classic quantifier elimination (QE) is needed (to be further discussed in Sec. 7). We provide two QE strategies.

Auxiliary-based QE. The general QE strategy is essentially Skolemisation: given $\forall\bar{X}.\,\exists\bar{Y}.\,\psi(\bar{X},\bar{Y})$, the compiler introduces a fresh auxiliary predicate $\mathsf{aux}_k(\bar{X})$ (with a fresh k) and rewrites the body ψ to DNF, $\bigvee_i (L_{i,1} \wedge \cdots \wedge L_{i,n_i})$, where each $L_{i,j}$ is a FOL literal. Each disjunct (called a *cube*) becomes a separate defining rule:

$$\mathsf{aux}_k(\bar{X}) \texttt{ :- } L_{i,1},\ \ldots,\ L_{i,n_i}. \qquad \text{(one rule per cube)}$$

The existential variables $\bar{Y}$ are absorbed into the rule bodies, and the original formula reduces to $\forall\bar{X}.\,\mathsf{aux}_k(\bar{X})$, encoded as the IC:

$$\texttt{:- not aux\_k(X}_1\texttt{, . . . , X}_j\texttt{), dom\_T}_1\texttt{(X}_1\texttt{), . . .}$$

The domain atoms $\texttt{dom\_T}_i$ are ASP-specific to restrict each variable X_i to its declared type T_i, inferred from the declarations (Sec. 4.2.1). The auxiliary rules' bodies may themselves contain the auxiliary predicate in cases of nested quantifier alternations, so the conversion *recurses* through the same pipeline (EliminateQuantifiers in Algorithm 1).

Note that auxiliary predicates extend the program's vocabulary: the resulting ASP program is a *conservative extension* (*a.k.a. quasi-equivalence* [57]), so its answer sets, projected onto the original signature Σ, coincide with those of the source SetLaH! program—but the full answer sets (over Σ

plus auxiliaries) contain extra atoms not present in the SetLah! answer sets. This is harmless for correctness, yet it can complicate debugging since the solver's output includes auxiliary predicates the user did not write. When only a single layer of quantifier elimination is needed, an alternative strategy avoids auxiliary predicates entirely by using ASP aggregates, as described next.

Aggregate-based QE. When a single existential quantifier block remains (no further alternation underneath), the compiler can instead eliminate it using an ASP aggregate, preserving the answer sets exactly. As above, the body under the quantifier is rewritten to DNF, $\bigvee_i (L_{i,1} \wedge \cdots \wedge L_{i,n_i})$. This DNF maps elegantly into ASP: each disjunctive cube becomes a condition joined by `;` inside the aggregate. For example, $\forall \bar{X}.\, \exists \bar{Y}.\, (L_1 \wedge L_2) \vee L_3$ becomes:

$$\texttt{:- \#count\{}\bar{Y}\texttt{: } L_1\texttt{, } L_2\texttt{ ; }\bar{Y}\texttt{: } L_3\texttt{\} = 0, dom\_T}_1\texttt{(X}_1\texttt{), . . .}$$

No new predicates are introduced, so the generated ASP program has exactly the same stable models over the original vocabulary. Since aggregate bodies cannot contain nested aggregates, this strategy cannot be applied recursively for deeper quantifier alternations.

Strategy. By default, the compiler prefers aggregate-based QE to minimise auxiliary predicates, keeping the generated program as close to the input specification as possible (not cluttered by fresh predicates). However, aggregates can make the grounder enumerate a large condition space (*cf.*, [43]). When grounding dominates runtime, SetLah! can instead prefer auxiliary-based QE: the existential search is materialised as ordinary rules, enabling the ASP grounder to share intermediate predicates and apply its usual rule-level optimisations. This is the grounding-scalability option used in the Alloy benchmarks (Sec. 6.2).

Regardless of which strategy is chosen, the translation is correct and terminates:

Theorem 5.3 (Correctness and Termination of IC conversion). *For every closed FOL formula φ over a finite Herbrand base $\mathcal{A}_{\mathcal{D}}$, the translation procedure terminates and produces a finite set of ICs and auxiliary rules that is equivalent to φ over $\mathcal{A}_{\mathcal{D}}$.*

5.5 Correctness

We show that the compiled ASP program admits exactly the same solutions as the source SetLah! program: every SetLah! answer set appears as a stable model of the ASP output (after projecting away auxiliaries), and vice versa. The argument proceeds in two levels: first at the PCLP block level, then lifted to the stratified SetLah! level. Let Σ denote the original signature (predicates declared in Π), and let $\hat{\Sigma} \supseteq \Sigma$ be the extended signature that includes auxiliary predicates introduced by QE. For an interpretation M over $\hat{\Sigma}$, write $M \restriction_\Sigma$ for its projection onto Σ.

Theorem 5.4 (Block-level preservation). *Let $P_i = (\mathcal{R}_i, C_i, \Phi_i)$ be a PCLP block, I an input answer set over Σ, and $\hat{P}_i$ the ASP program produced by the translation. Then:*

(1) When $\Phi_i = \emptyset$, $\mathrm{AS}(\hat{P}_i \cup I) = \mathcal{S}^*(P_i, I)$.
(2) When $\Phi_i \neq \emptyset$, $\left\{ M \restriction_\Sigma \;\middle|\; M \in \mathrm{AS}(\hat{P}_i \cup I) \right\} = \mathcal{M}(P_i, I)$.

Theorem 5.5 (SetLah! program correctness). *Let $\Pi = B_1, \ldots, B_n$ be a well-formed SetLah! program, and let $\hat{\Pi}$ be the ASP program produced by translating each block B_i. Then*

$$\mathcal{M}(\Pi) \;=\; \left\{ M \restriction_\Sigma \;\middle|\; M \in \mathrm{AS}(\hat{\Pi}) \right\}.$$

Implementation. The translation of Phases 1 and 2 is implemented using Egglog [92], an equality saturation engine that explores equivalent formula representations over PCLP blocks. The extraction cost first favours formulas whose top-level conjunctions have been split, because this reduces the maximum number of variables per emitted IC. It then applies the QE preference described above:

aggregate-based QE is cheaper by default, while the grounding-scalability option assigns lower cost to auxiliary-based QE. After translation, the resulting blocks are rewritten into ASP syntax and merged into a single Clingo program.

5.6 Extending SetLah! with Set Operations

Beyond FOL constraints, SetLah! supports a small collection of set-level operations. These operations do not extend the core PCLP semantics: they are surface constructs that are desugared into ordinary FOL formulas or ASP aggregates *before* the translation pipeline of the previous subsections runs. Their purpose is pragmatic: they let users write common modelling idioms such as subset, partition, and cardinality constraints directly, while still compiling to the same FOL-to-ASP pipeline and to standard Clingo aggregate syntax when appropriate.

The idea is straightforward: the extension of a predicate can be named as a set of tuples, and operations such as subset or cardinality checks can be applied to that set. The compiler then expands each set atom locally. For example, subset becomes an implication under universal quantification, while cardinality and partition constraints become ASP aggregate constraints. Thus, set operations are best understood as notation for common finite-domain constraints, not as an additional semantic foundation for SetLah!. They complement the core FOL constraints when the natural specification is set-level—for instance, when a model needs cardinality over a parameterised relation. Details are provided in the supplementary material.

6 Evaluation

In this section, we evaluate SetLah! on combinatorial search problems spanning constraint satisfaction, model checking, and quantum circuit synthesis. Our experiments are designed to answer the following research questions:

- *RQ1 (Expressivity & usability):* Can SetLah! effectively model common combinatorial tasks with high-level abstractions? Are the translation techniques—splitting and quantifier elimination—necessary for different problems?
- *RQ2 (Performance):* Given the remarkable advances in SAT solving over decades [12, 34], how does the SetLah! +Clingo workflow perform on different tasks, compared to SAT-based approaches?
- *RQ3 (Practical utility):* Can SetLah! help to solve real-world combinatorial problems, where advanced techniques like incremental solving in SAT are usually required?

For conducting the experiments with potentially time-consuming and multi-threaded tasks, we use an Ubuntu server with AMD EPYC 7763. The results other than massively parallel runs are also replicable with commodity PCs. For ASP solving, we used Clingo 5.8.0 which is the standard choice for ASP solving with multi-threading support. While for SAT solving, since numerous SAT solvers are available, we picked several state-of-the-art solvers from either MiniSat ([31], as the most influential on later systems) family or CaDiCaL ([11], as the recently performance-leading one) family. The specific solver choices are described in the corresponding sections.

6.1 RQ1: Modelling Common Combinatorial Tasks

To assess the expressivity of SetLah!, we implemented a suite of standard combinatorial problems from the Maple benchmark [17]. Maple is a computer algebra system based on MapleSAT [60]; in [17], problems such as N-queens and Sudoku are demonstrated as examples of its modelling capabilities. We use the same case studies to showcase SetLah!'s expressivity, investigating two questions: (1) which translation techniques—splitting and quantifier elimination (Sec. 5)—are needed for each problem, and (2) where set-level surface constructs make the model closer to the problem statement than pure FOL alone. Tab. 2 summarises the results. Each problem is modelled with one constraint per logical check, reflecting how the problem is naturally stated, rather than conjoining

Table 2. Comparison of SetLah! models written with FOL only and with FOL plus set operations. The "Split" and "QE" columns indicate whether FOL splitting and quantifier elimination are used during translation.

Problem	#Block	#FO	Split	QE	#(FO+Set)	Split	QE
15-puzzle	1	8	✓	✓	8	✗	✓
Einstein riddle	5	25	✓	✓	25	✗	✓
Graeco-Latin Square	3	10	✓	✓	10	✓	✗
N-Queens	1	3	✓	✓	3	✓	✗
N-clique	1	NA			2	✗	✗
Sudoku	1	5	✓	✓	4	✓	✗

all constraints into a single formula. Problems that decompose into sub-problems (*e.g.*, Einstein riddle with 5 blocks, Graeco-Latin Square with 3) directly benefit from SetLah!'s block structure.

As the table shows, set operations are primarily a modelling convenience rather than a separate solving technique. They generally do not reduce the number of constraints; instead, they replace quantified or relational boilerplate with direct set-level notation. For problems like Graeco-Latin Square, N-Queens, and Sudoku, cardinality constraints avoid spelling uniqueness through quantified FOL formulas, eliminating the need for quantifier elimination while still requiring splitting. Conversely, for 15-puzzle and Einstein riddle, set operations express relational bookkeeping more compactly, removing the need for splitting while quantifier elimination remains necessary. The performance contribution evaluated below is still the FOL-to-ASP pipeline—especially splitting and QE—whereas set operations explain why the source models can stay concise.

The concision benefit is not only the number of constraints, but their alignment with the mathematical specification. Taking the Einstein riddle as an example (a puzzle of 5 people, each with 5 attributes), the 25 FOL constraints without set operations correspond to the 15 given clues and 10 bijectivity constraints, naturally matching the problem statement and distributing the model across 5 blocks for the attributes. By contrast, the translated ASP program is a single monolithic encoding with 9 appearances of different forms of aggregates, which is less intuitive to implement even when the program size is comparable to the SetLah! version.

A notable case is N-clique, which *cannot* be expressed in pure FOL within SetLah! (marked NA) since the formula would require a conjunction of N atoms, which is not achievable when the value of N is not fixed at modelling time. With set operations, the problem is directly expressed by cardinality constraints. By contrast, N-queens *can* be expressed in pure FOL because its constraints are pairwise (no two queens attack each other), so they do not require an N-way conjunction.

The compilation time from SetLah! to ASP is short—around 1 second for most cases and up to 5 seconds across our full evaluation. Since compilation is performed once and the resulting ASP program is reused across instances, all runtime comparisons below exclude compilation time. Note that, unlike in RQ1 we use all valid set operations, later benchmarks only utilise set partition as the extension of constraints, which desugars into standard ASP aggregates (see supplementary material). The performance results are therefore attributable to the FOL constraint pipeline.

6.2 RQ2: Performance and Scalability

GL Square Evaluation. Among the cases in Tab. 2, the Graeco-Latin Square is the only one where performance merits detailed investigation, as the others are either solved within seconds or well-studied (*e.g.*, N-queens [14], N-clique [55]). The GL square is also a historically significant benchmark: in [17], the author notes that "Euler could have disproved the conjecture (with 23

Table 3. PAR-2 Performance of GL Square in seconds ($\text{Mean}_{\text{CV\%}}$). Comparison of Sequential (1T), Parallel (8T), and Massively Parallel (64T) execution. *Timeout = 86,400s (sequential), 7,200s (parallel)*

	Sequential (1 Thread)			Parallel (8 Threads)			Parallel (64 Threads)		
Ord	**SetLah!**	**CaDiCaL**	**Glucose**	**SetLah!**	**Parkissat**	**CMS**	**SetLah!**	**Parkissat**	**CMS**
6	$10.7_{13\%}$	$\mathbf{9.9}_{8\%}$	$12.0_{0\%}$	$\mathbf{2.6}_{12\%}$	$9.9_{5\%}$	$39.8_{7\%}$	$\mathbf{1.7}_{12\%}$	$3.9_{13\%}$	$14.5_{10\%}$
7	$\mathbf{0.6}_{69\%}$	$0.8_{53\%}$	$2.7_{1\%}$	$\mathbf{0.2}_{35\%}$	$1.3_{7\%}$	$1.7_{95\%}$	$\mathbf{0.2}_{17\%}$	$1.3_{4\%}$	$0.4_{4\%}$
8	$\mathbf{50}_{69\%}$	$72_{110\%}$	$73_{0\%}$	$10.6_{71\%}$	$23.1_{36\%}$	$\mathbf{2.7}_{3\%}$	$\mathbf{0.9}_{109\%}$	$4.6_{39\%}$	$3.0_{2\%}$
9	$2941_{110\%}$	$1073_{83\%}$	$\mathbf{259}_{1\%}$	$618_{126\%}$	$1053_{79\%}$	$\mathbf{426}_{1\%}$	$\mathbf{34}_{78\%}$	$239_{83\%}$	$76_{13\%}$
10	$82780_{80\%}$	$\mathbf{7157}_{85\%}$	$172800_{0\%}$	$\mathbf{4425}_{120\%}$	$8396_{76\%}$	$13621_{18\%}$	$\mathbf{504}_{135\%}$	$1915_{92\%}$	$1602_{13\%}$

hours) if given Maple." We also tried first-order modelling with SMT solvers, including finite model finding in CVC5 [77], but none scaled beyond small orders.

All encodings—including the Maple and SMT attempts discussed above—apply standard symmetry-breaking by fixing the first row and column (*i.e.*, the *reduced* GL square). We compare SetLah!-generated ASP encodings against SAT solvers: CaDiCaL 1.9.5 and Glucose 4.2 for sequential runs (via PySAT [50]), and ParKissat-RS [1] and CryptoMiniSat 5.1.3 [84] for parallel runs. Each configuration runs with 10 random seeds to account for heuristic variance, under timeouts of 86,400s (sequential) and 7,200s (parallel). We report *PAR-2* (mean runtime where timeouts count as 2× the limit) and the *Coefficient of Variation* (CV% under PAR-2), which captures how sensitive each solver is to random-seed choices. Tab. 3 compares three execution modes: Sequential (1T), Parallel (8T), and Massively Parallel (64T).

Analysis. The key message from Tab. 3 is not that any single solver dominates—the high CV% values confirm that runtime is heavily influenced by solver heuristics.

In the sequential baseline, CaDiCaL leads on larger orders (Order 9–10), which is unsurprising given decades of SAT-specific engineering. However, CaDiCaL does not natively support parallel execution. Clingo, by contrast, scales seamlessly: at Order 10, moving from 1 to 8 threads yields a super-linear speedup (from 82,780s to 4,425s), surpassing both parallel SAT solvers.

With 64 threads, Clingo achieves the best PAR-2 across *all* orders, confirming that ASP solving is a strong competitor for hard combinatorial problems even against state-of-the-art SAT solvers.

Grounding Scalability: Alloy's benchmarks. The GL Square experiment tests solving competitiveness; we now examine whether SetLah!'s compilation also makes the grounding scalable. Unlike the GL Square, where solving dominates, many practical instances are bottlenecked by the translation from high-level specification to solver-specific encodings. We evaluate on benchmarks adapted from Alloy [51], a mature modelling tool (over 20 years of engineering) based on FOL with relational algebra and SAT as backend. Its constraint language is similar to the FOL constraints in SetLah!, making these benchmarks a natural fit. Two differences between Alloy and SetLah! should be noted. First, Alloy targets properties of software systems rather than general combinatorial problems. Second, its backend Kodkod [86] applies *equisatisfiable* transformations—rewrites that preserve satisfiability but not necessarily the exact set of models (in contrast to SetLah! which preserves the quasi-equivalent models as mentioned in Sec. 5.4). Thus the comparison is not a claim that the two tools implement identical encodings; it isolates whether SetLah!'s FOL-to-ASP translation can instantiate comparable relational constraints more efficiently.

In both tools, the expensive preprocessing step is a form of grounding: Kodkod instantiates bounded relational formulas into propositional SAT variables and clauses, while SetLah! compiles first-order constraints into ASP rules and lets Clingo ground those rules. The semi-declarative block

Table 4. Scalability comparison of Alloy vs. SetLah!. Times in seconds (Grd = grounding, Slv = solving). Speedup columns show the ratio (positive means SetLah! is faster, negative means Alloy is faster).

genealogy (recursive, SAT)
7 definitions, 11 constraints

	Alloy		SetLah!		Speedup	
n	Grd	Slv	Grd	Slv	Grd	Total
15	0.14	0.05	0.04	0.00	3.5×	4.8×
30	1.24	0.80	0.41	0.09	3.0×	4.1×
45	6.42	1.54	2.06	0.87	3.1×	2.7×
60	22.21	12.84	6.65	0.95	3.3×	4.6×
75	58.76	23.08	16.85	1.32	3.5×	4.5×

academia_3 (recursive, UNSAT)
17 definitions, 22 constraints

	Alloy		SetLah!		Speedup	
n	Grd	Slv	Grd	Slv	Grd	Total
25	0.38	0.08	0.05	0.00	7.6×	9.2×
50	2.67	0.49	0.38	0.00	7.0×	8.3×
75	10.14	1.52	1.43	0.00	7.1×	8.2×
100	25.08	1.52	3.45	0.00	7.3×	7.7×
125	49.17	7.46	8.05	0.00	6.1×	7.0×

academia_2 (non-recursive, SAT)
8 definitions, 9 constraints

	Alloy		SetLah!		Speedup	
n	Grd	Slv	Grd	Slv	Grd	Total
100	0.72	0.22	0.12	0.00	6.0×	7.8×
200	4.09	0.97	0.53	0.02	7.7×	9.2×
300	11.40	2.29	1.13	0.03	10.1×	11.8×
400	24.79	12.62	2.24	0.05	11.1×	16.3×
500	44.85	96.45	3.59	0.07	12.5×	38.6×

birthday (non-recursive, UNSAT)
4 definitions, 4 constraints

	Alloy		SetLah!		Speedup	
n	Grd	Slv	Grd	Slv	Grd	Total
5	0.01	0.00	0.01	0.00	±1.0×	±1.0×
10	0.06	0.03	0.03	0.10	2.0×	-1.4×
15	0.19	0.06	0.12	1.94	1.6×	-8.2×
20	0.29	0.08	0.44	15.28	-1.5×	-42.5×
25	0.61	0.14	1.31	81.01	-2.1×	-109.8×

structure also matters for modelling these examples. For instance, `academia_3` first chooses whether each person is faculty or student, and only then chooses whether chosen students are graduate or undergraduate. This staged dependency is written directly as two SetLah! blocks; flattening it into one ASP stratum would force the modeller to manually recover the same dependency structure.

We selected four representative instances from [68] that span two axes: *recursive vs. non-recursive* definitions, and *SAT vs. UNSAT* outcomes. For each, we scale the input size and compare grounding and solving times between Alloy (version 6.2, compiled to Glucose) and SetLah! (compiled to Clingo, single-threaded). As discussed in Sec. 5.4, the auxiliary-based QE option can improve grounding scalability by replacing large aggregate bodies with shareable ordinary rules. We enable this option for the Alloy benchmarks because their runtime is dominated by grounding rather than by search. Tab. 4 presents the results.

Analysis. The first three benchmarks show consistent advantages for SetLah!, for both grounding and solving phases. The speedup is predominantly grounding-driven: SetLah!'s translation to Clingo produces ground programs that are substantially smaller than those generated by Alloy's Kodkod backend, so that the grounding time is reduced. In other words, the splitting in SetLah!'s translation (Sec. 5) effectively mitigates the grounding bottleneck for large instances, resulting in better scalability. The grounding time dominates the total runtime across those three benchmarks for SetLah!; while in academia_2 at n=500, Alloy's solving time spikes to 96s (compared to under 0.1s in SetLah!), indicating that the redundancy of Kodkod's propositional encoding also contributes to solving overhead. Note that the "recursive" benchmarks use recursion only for transitive closure, which both Clingo and Kodkod handle natively; the grounding advantage of SetLah! is therefore not specific to transitive closures but more generally to recursive definitions.

The **birthday** benchmark reveals the differences. Unlike the others, this instance models a transition system rather than a classical combinatorial search problem, and the search space grows exponentially with depth. KODKOD's automatic symmetry-breaking is particularly effective here because model checking of transition systems is inherently highly symmetric. SETLAH! struggles from n=15 on solving time without such automated techniques; grounding also grows faster, likely due to the frame problem [46]: FOL must explicitly state non-change of unaffected variables, whereas ALLOY's "override" operator handles this naturally. Bringing manual symmetry-breaking in SETLAH! could close this gap, but we use only the plain FOL encoding matching ALLOY's specification. This is a limitation of the current surface language rather than a fundamental limit of ASP as a backend: a frame or override construct could desugar to ASP programs (at least capturing the propositional encoding in ALLOY) in the same way that set operations desugar to aggregates. We leave such transition-system-specific modelling support to future work.

6.3 RQ3: Qubit Layout Synthesis as a Case Study

A key advantage of SETLAH!'s translation is *solver transparency*: the output is a reusable ASP program, not only a one-shot ground instance (as those SAT-based modelling tools, *cf.* Sec. 7), so it can be embedded in CLINGO's multishot API and solved incrementally or in parallel (Sec. 7). We demonstrate this on *quantum circuit layout synthesis* [85] from Q-SYNTH v2 [81], a real-world problem where multishot programming [38], analogous to incremental SAT [31], is essential.

The task is to map logical qubits of a quantum circuit onto the physical topology of a quantum processor while minimising the number of SWAP gates, which is NP-complete. Q-SYNTH formulates each mapping step as a satisfiability problem solved incrementally: the SWAP count is increased step by step until a valid layout is found (SAT) or the bound is exhausted (UNSAT); see [81] for details of the original problem and method, which are orthogonal to our experimental focus.

Modelling comparison. Similar to how we demonstrate the GL Square in Sec. 3, Q-Synth's original encoding [81] uses 8 parameterised variables with 5 groups of constraints—already concise on paper, but manually encoded into CNF in the implementation, totalling roughly 600 lines of Python code. The SETLAH! encoding is substantially more compact: 10 declarations, 5 blocks with 10 rules and 22 constraints expressing the same logic. The compiled ASP program is about 40 lines, with about 60 lines of Python glue code needed to integrate it into Q-SYNTH.

The blocks follow the structure of the synthesis algorithm thanks to the *semi-declarativeness* of SETLAH!: the case when the choice on one predicate is attempted only after the previous choice is fixed appears several times in the PCLP rules, in contrast to original encoding where the dependencies are implicit in the boolean formulas. The complex FOL constraints involve at most 12 atoms or 10 connectives and quantifiers in a single formula; writing the equivalent ASP by hand is possible, and would likely match the backend performance, but it requires the same auxiliary-predicate decomposition that SETLAH! generates automatically.

Setup. We use Q-Synth's benchmark of 69 instances of varying difficulty drawn from standard quantum circuit families, and compare Q-SYNTH's original SAT-based backend against a replacement using SETLAH!'s compiled ASP program with CLINGO, in two matched settings:

- **Single-threaded:** `clingo -t1` vs. `cd19` (CaDiCaL 1.9.5, the fastest on this task in PYSAT [50]).
- **Multi-threaded (8T):** `clingo -t8` vs. `cms -t8` (CRYTOMINISAT 5.1.3 [84], a well-maintained parallel SAT solver with Python API support).

All configurations use the same incremental solving loop from Q-Synth; only the backend solver and encoding differ. The timeout is 10,800s per instance.

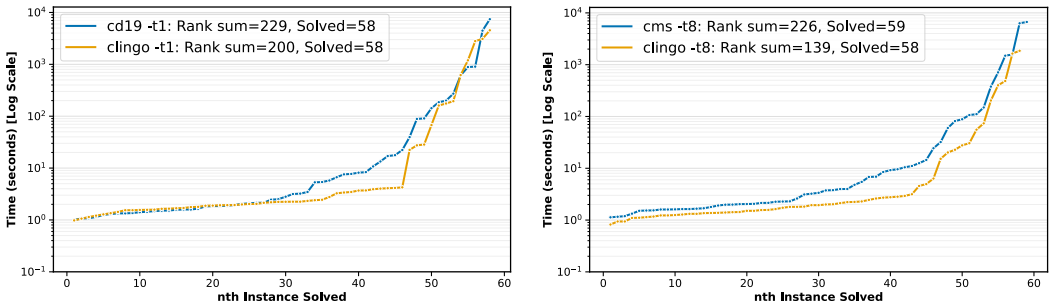

Fig. 3. Cactus plots for Q-Synth qubit layout synthesis (69 instances, timeout = 10,800s). Left: single-threaded; Right: 8 threads. The x-axis shows the n-th fastest-solved instance per configuration (not necessarily the same instance); the y-axis shows cumulative runtime on a log scale. The legend reports *rank sum* (lower = better: sum of per-instance runtime ranks across all instances) and total instances solved.

Results and Analysis. Fig. 3 presents the cactus plots. We use *rank sum* among 4 solvers—the sum of each solver's per-instance runtime rank—as an aggregate measure of runtime consistency, complementing the solved-instance count. Note that the n-th point on each curve corresponds to that solver's n-th fastest instance, which need not be the same instance across solvers; thus a solver whose curve is uniformly lower is *overall* faster, not necessarily on every individual instance.

In the single-threaded setting, both `clingo -t1` and `cd19` solve 58 of 69 instances. `clingo -t1` achieves a rank sum of **200** vs. 229 for `cd19`, indicating overall faster runtimes. Given that CaDiCaL is the current performance leader among SAT solvers, matching it on solved instances while achieving a better rank sum is strong evidence of SetLah!'s competitiveness. On the first ~25 easy instances both solvers stay in the 1–2s range. From instance 25 onward the curves separate: `clingo -t1` remains at ~2–4s through instance 45, while `cd19` climbs to ~8–20s. On the hardest instances (50+), both spike into the 10^3–10^4s range and the gap narrows.

The multi-threaded comparison shows a stronger separation. `clingo -t8` achieves a rank sum of **139**—far below `cms -t8`'s 226—with the gap visible from the earliest instances onward. In the medium-difficulty range (instances 40–55), the gap widens to roughly an order of magnitude. `cms -t8` does solve one additional instance (59 vs. 58), but Clingo dominates on aggregate runtime. This demonstrates that Clingo's native incremental solving, made accessible through SetLah!'s solver-transparent translation, is highly effective on real-world iterative search problems.

Unlike the GL Square, Q-Synth is not one monolithic satisfiability problem. It solves a sequence of increasingly deep subproblems, most of which are UNSAT before the first valid layout is found. This structure favours Clingo's incremental grounding and solving, which can retain program structure across steps. Multi-threading yields less dramatic speedups than in GL Square because each incremental subproblem is smaller; the bottleneck is the number of depth steps, not only the size of one search instance.

6.4 Discussion

State-of-the-art SAT and ASP solvers are mature systems developed over decades, with CDCL-style search providing the core performance engine [43]. Performance among different solvers is highly instance-dependent, especially on satisfiable cases where heuristics play a decisive role [34, 47]. No single solver dominates all instances, so our evaluation should be read as evidence that SetLah! with Clingo is *competitive*, not universally superior. Beyond the performance results above, we highlight several broader insights emerging from our experiments.

ASP as a viable backend. Although less well-known than SAT in the Programming Languages community, ASP is a competitive alternative for combinatorial problems. Clingo's portfolio-based parallel solving [30, 42] and its native support for incremental solving make it one of the most capable maintained systems in this space—yet it has received limited attention outside the LP/KR communities (*e.g.*, [9, 66, 90] start to address the use of ASP in PL only from 2023), partly due to the difficulty of its input language. SetLah! lowers this barrier: its block structure decomposes problems into sub-problems, each modelled declaratively with guess-and-check via FOL constraints, letting non-experts automatically obtain efficient ASP encodings.

Modularity and extensibility. SetLah!'s rewriting pipeline is modular and orthogonal to existing optimisation techniques for logic programs: rule rewriting for better grounding [8, 28] or solving [45, 89] can be integrated into the translation or post-processing without changing the source language. Supporting alternative backends such as SAT or Datalog—when ASP's full expressiveness is not needed—is a promising direction for combining the strengths of different solvers.

LLMs and SetLah! One may wonder whether large language models (LLMs) make SetLah! unnecessary. In our experience, LLMs can generate PySAT code for GL Squares but struggle with less common APIs (*e.g.*, CrytoMiniSat), and—more fundamentally—the resulting CNF encodings are hard to trust: like manual modelling (Sec. 3.1), the encoding may diverge from the problem statement, making verification difficult. Even as models improve, SetLah! can serve as a preferable generation target: its FOL constraints are closer to natural language, and its encodings are far smaller than equivalent SAT code, making them both cheaper to generate and easier to verify.

7 Related Work

Transformation of Formulas and Programs. The CNF transformations in SAT solving (*e.g.*, Tseitin transformation [87]) convert arbitrary propositional formulas into the CNF format required by SAT solvers. While the transformation can be linear in the size of the formula, the propositional formulas themselves can be exponential in the size of the original problem specification. Like in the GL square, our preliminary experiments confirmed that the formula generation grows fast and takes minutes for n=6 when applying Tseitin transformation directly on the grounded FOL formula. In contrast, the translation from SetLah! to ASP (Sec. 5) operates at a higher level of abstraction: each FOL formula is first rewritten into split ASP constraints to mitigate the grounding cost. From the aspect of mimicking the human modelling process, our idea coincides with Lean-Egg [80], which also utilises equality saturation in the domain of automated mathematical reasoning.

F2LP [58] also translates first-order formulas into logic programs, and quantifier elimination through auxiliary predicates in SetLah! is therefore not new by itself. The difference is that SetLah! treats FOL translation as part of a modelling language and an optimisation pipeline: conjunction splitting is performed before QE so that the ASP grounder sees smaller-variable constraints. This is the source of the grounding improvements discussed in Sec. 6.2.

Closely related to our work, FOLASP [27] accepts the language of IDP [19] as input for ASP solvers, also sharing the motivation of bridging FOL and ASP. However, the language is designed upon inductive definitions [26], making the semantics less intuitive than SetLah!'s branching fixpoint semantics based on guess-and-check. Moreover, FOLASP's translation to ASP is ad hoc—purely bottom-up on the formulas without considering grounding efficiency, resulting in poor scalability of the translated ASP programs in practice (as shown in [27]).

Relevant Paradigms for Constraint Solving. As illustrated in Sec. 3, the principal differences between ASP and SAT are: 1) the head-body structure of logic programs vs. the propositional implication, and 2) the first-order variables in ASP vs. the propositional syntax of SAT. The former

was first characterised by the loop formulas [64], which connect the stable model semantics to SAT solving, enabling DPLL-based SAT solvers [23] to be adapted for ASP [59, 83] and leading to the development of CDCL-based ASP solvers [4, 41].

ezASP [33] is closest to SetLah! in spirit: it also argues that many users do not need the full generality of the stable model semantics and offers a stratified, easier-to-read fragment of ASP. SetLah! follows the same ideology but makes two different choices. First, it removes negation from rule bodies and makes choice explicit; this rules out unstratified negation, and even negation-heavy encodings that are legal in ezASP, in exchange for a direct guess-and-check reading. Second, SetLah! sequences PCLP blocks, so choices in a later block may depend on atoms chosen in an earlier one. This enables stratified examples from Alloy and Q-Synth.

As discussed, SMT's quantifier handling [22] may not scale on harder combinatorial search problems in finite domains: quantifiers are processed lazily compared to ASP's eager grounding, since SMT is designed for expressive (possibly infinite) theories rather than finite-domain enumeration. Performance advantages of ASP over SMT have also been observed in Datalog synthesis [9, 74], where fixpoint-based reasoning is more natural. A common misconception is that SMT, being "more expressive" than SAT, should subsume it on finite problems—but *SMT is not first-order SAT*: a solver for problem class A is not necessarily efficient on every subset of A. This motivates future work on detecting finite-domain SMT problems and translating them into SetLah! for efficient solving.

The idea of branching fixpoint semantics is closely related to the semantics of Dusa [66], which is also a choice-based language that can mimic stable model semantics. However, Dusa uses its own backtracking-based evaluation rather than leveraging existing ASP systems, and supports restricted constraints, limiting its capability on modelling and solving general combinatorial problems. In this work, we use SetLah! to release the power of ASP solvers for combinatorial search, but neither have we illustrated all practical advantages of ASP over SAT, nor has SetLah! supported all features of ASP. For example, ASP solvers natively support enumerating all stable models (*e.g.*, 92 solutions for the 8-queens problem), which is rarely available in SAT/SMT solvers [53]—manually banning previously found models introduces significant overhead in both modelling and solving [14]. Features like brave/cautious reasoning [5] and projection [40] are also naturally supported by the compiled ASP programs. Other useful features such as weak constraints [18] and heuristics [39] require language-level extensions to SetLah!, to be investigated in future work.

Traditional constraint programming (CP) systems are rooted in Prolog-style top-down unification and backtracking, whose scalability limitations compared to SAT-based methods are well-discussed [15]. Consequently, more recent CP tools adopt SAT or other modern solvers as backends: Enfragmo [2] and Picat [93] compile constraint models to SAT. However, unlike SetLah!'s compilation to ASP which preserves *solver transparency*, their strict pipeline separation aggressively flattens high-level specifications into solver-specific, instance-grounded formats, preventing users from exploiting advanced solver features such as incremental and parallel solving. On the other hand, with certain finite domain like integer variables, classical CLP(FD) [29] or modern CP systems [73] can be a better choice than SAT/ASP-based methods.

Among CP tools, MiniZinc [70] and Conjure [3] share a similar motivation with SetLah!: providing a high-level declarative modelling language. However, their language designs differ fundamentally. CP languages are primarily *algebraic*, supporting more expressive constraints than logic formulas—including numerical and global constraints—which is why our evaluation in Sec. 6 compares with SAT rather than CP. Their backend-agnostic design requires compiling models into intermediate representations, which loses rich relational structures. In contrast, SetLah! is inherently *relational* and preserves structural information through its modular translation to ASP. Similar to SMT vs. SAT, for combinatorial problems naturally modelled in pure ASP, native ASP solvers outperform MiniZinc approaches in the benchmark of [32, 79].

Acknowledgments

We thank Matthew Flatt and Hila Peleg for their input during the brainstorming stage of this work. We are also grateful to Abtin Molavi for pointing out Q-Synth as an illustrative benchmark for SetLah!. We are also grateful the anonymous OOPSLA'26 reviewers for their constructive feedback. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001.

Data Availability Statement

The software artefact accompanying this paper is publicly available on Zenodo [91]. The artefact contains the supplementary appendix with proofs omitted from the manuscript, the source code of SetLah!, a modified version of Q-Synth v2, and all benchmark data and build scripts for reproducing the evaluation results reported in Sec. 6.

References

[1] 2026. *ParKissat-RS*. Commit 537bbea; accessed 2026-01-15. https://github.com/shaowei-cai-group/ParKissat-RS

[2] Amir Aavani, Xiongnan (Newman) Wu, Shahab Tasharrofi, Eugenia Ternovska, and David G. Mitchell. 2012. Enfragmo: A System for Modelling and Solving Search Problems with Logic. In *LPAR (LNCS, Vol. 7180)*. Springer, 15–22. doi:10.1007/978-3-642-28717-6_4

[3] Özgür Akgün, Alan M. Frisch, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. 2022. Conjure: Automatic Generation of Constraint Models from Problem Specifications. *Artif. Intell.* 310 (2022), 103751. doi:10.1016/J.ARTINT.2022.103751

[4] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. 2015. Advances in WASP. In *LPNMR (LNCS, Vol. 9345)*. Springer, 40–54. doi:10.1007/978-3-319-23264-5_5

[5] Mario Alviano, Carmine Dodaro, and Francesco Ricca. 2014. Anytime Computation of Cautious Consequences in Answer Set Programming. *Theory Pract. Log. Program.* 14, 4-5 (2014), 755–770. doi:10.1017/S1471068414000325

[6] Isaac Balbin and Kotagiri Ramamohanarao. 1987. A Generalization of the Differential Approach to Recursive Query Evaluation. *J. Log. Program.* 4, 3 (1987), 259–262. doi:10.1016/0743-1066(87)90004-5

[7] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *TACAS (LNCS, Vol. 13243)*. Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

[8] Alexander Beiser, Markus Hecher, Kaan Unalan, and Stefan Woltran. 2024. Bypassing the ASP Bottleneck: Hybrid Grounding by Splitting and Rewriting. In *IJCAI*. ijcai.org, 3250–3258. https://www.ijcai.org/proceedings/2024/360

[9] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2023. From SMT to ASP: Solver-Based Approaches to Solving Datalog Synthesis-as-Rule-Selection Problems. *Proc. ACM Program. Lang.* 7, POPL (2023), 185–217. doi:10.1145/3571200

[10] Armin Biere. 2009. Bounded Model Checking. In *Handbook of Satisfiability*, Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 457–481. doi:10.3233/978-1-58603-929-5-457

[11] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. 2024. CaDiCaL 2.0. In *CAV (LNCS, Vol. 14681)*. Springer, 133–152. doi:10.1007/978-3-031-65627-9_7

[12] Armin Biere, Mathias Fleury, Nils Froleyks, and Marijn J. H. Heule. 2023. The SAT Museum. In *Pragmatics of SAT Workshop (CEUR Workshop Proceedings, Vol. 3545)*. CEUR-WS.org, 72–87. https://ceur-ws.org/Vol-3545/paper6.pdf

[13] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press.

[14] Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. 2022. User-Propagators for Custom Theories in SMT Solving. In *SMT Workshop (CEUR Workshop Proceedings, Vol. 3185)*. CEUR-WS.org, 71–79. https://ceur-ws.org/Vol-3185/extended6630.pdf

[15] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. 2006. Propositional Satisfiability and Constraint Programming: A comparative survey. *ACM Comput. Surv.* 38, 4 (2006), 12. doi:10.1145/1177352.1177354

[16] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. 2011. Answer set programming at a glance. *Commun. ACM* 54, 12 (2011), 92–103. doi:10.1145/2043174.2043195

[17] Curtis Bright, Jürgen Gerhard, Ilias S. Kotsireas, and Vijay Ganesh. 2019. Effective Problem Solving Using SAT Solvers. In *MC (CCIS, Vol. 1125)*. Springer, 205–219. doi:10.1007/978-3-030-41258-6_15

[18] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. 2000. Enhancing Disjunctive Datalog by Constraints. *IEEE Trans. Knowl. Data Eng.* 12, 5 (2000), 845–860. doi:10.1109/69.877512

[19] Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, and Marc Denecker. 2018. Predicate logic as a modeling language: the IDP system. In *Declarative Logic Programming: Theory, Systems, and Applications*. ACM Books, Vol. 20. ACM / Morgan & Claypool, 279–323. doi:10.1145/3191315.3191321

[20] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.* 1, 1 (1989), 146–166. doi:10.1109/69.43410

[21] Hubie Chen and Stefan Mengel. 2024. Optimally Rewriting Formulas and Database Queries: A Confluence of Term Rewriting, Structural Decomposition, and Complexity. In *ICDT (LIPIcs, Vol. 290)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:17. doi:10.4230/LIPICS.ICDT.2024.16

[22] Marek Danco, Petra Hozzová, and Mikolás Janota. 2025. From MBQI to Enumerative Instantiation and Back. *CoRR* abs/2506.22584 (2025). arXiv:2506.22584 doi:10.48550/ARXIV.2506.22584

[23] Martin Davis and Hilary Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (1960), 201–215. doi:10.1145/321033.321034

[24] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[25] Marc Denecker, Yuliya Lierler, Miroslaw Truszczynski, and Joost Vennekens. 2019. The informal semantics of Answer Set Programming: A Tarskian perspective. *CoRR* abs/1901.09125 (2019). arXiv:1901.09125 http://arxiv.org/abs/1901.09125

[26] Marc Denecker and Eugenia Ternovska. 2008. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.* 9, 2 (2008), 14:1–14:52. doi:10.1145/1342991.1342998

[27] Kylian Van Dessel, Jo Devriendt, and Joost Vennekens. 2021. FOLASP: FO(.) as Input Language for Answer Ser Solvers. *CoRR* abs/2108.04020 (2021). arXiv:2108.04020 https://arxiv.org/abs/2108.04020

[28] Carmine Dodaro, Giuseppe Mazzotta, and Francesco Ricca. 2024. Blending Grounding and Compilation for Efficient ASP Solving. In *KR*. doi:10.24963/KR.2024/30

[29] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. 2005. A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3668)*, Maurizio Gabbrielli and Gopal Gupta (Eds.). Springer, 67–82. doi:10.1007/11562931_8

[30] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. 2018. Parallel Answer Set Programming. In *Handbook of Parallel Constraint Reasoning*. Springer, 237–282. doi:10.1007/978-3-319-63516-3_7

[31] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *SAT (LNCS, Vol. 2919)*. Springer, 502–518. doi:10.1007/978-3-540-24605-3_37

[32] Thomas Eiter, Tobias Geibinger, Nysret Musliu, Johannes Oetsch, and Tobias Kaminski. 2025. ASP-FZN: A Translation-Based Constraint Answer Set Solver. *Theory Pract. Log. Program.* 25, 4 (2025), 649–667. doi:10.1017/S1471068425100264

[33] Jorge Fandinno, Seemran Mishra, Javier Romero, and Torsten Schaub. 2023. Answer Set Programming Made Easy. In *Festschrift for Manuel Hermenegildo (LNCS, Vol. 13160)*. Springer, 133–150. doi:10.1007/978-3-031-31476-6_7

[34] Mathias Fleury and Daniela Kaufmann. 2024. Life span of SAT techniques. *CoRR* abs/2402.01202 (2024). arXiv:2402.01202 doi:10.48550/ARXIV.2402.01202

[35] Nils Froleyks, Marijn Heule, Ashlin Iser, Matti Järvisalo, and Martin Suda. 2021. SAT Competition 2020. *Artif. Intell.* 301 (2021), 103572. doi:10.1016/J.ARTINT.2021.103572

[36] Yeting Ge and Leonardo Mendonça de Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In *CAV (LNCS, Vol. 5643)*. Springer, 306–320. doi:10.1007/978-3-642-02658-4_25

[37] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2012. *Answer Set Solving in Practice*. Morgan & Claypool Publishers. doi:10.2200/S00457ED1V01Y201211AIM019

[38] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* 19, 1 (2019), 27–82. doi:10.1017/S1471068418000054

[39] Martin Gebser, Benjamin Kaufmann, Javier Romero, Ramón Otero, Torsten Schaub, and Philipp Wanko. 2013. Domain-Specific Heuristics in Answer Set Programming. In *AAAI*. AAAI Press, 350–356. doi:10.1609/AAAI.V27I1.8585

[40] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. 2009. Solution Enumeration for Projected Boolean Search Problems. In *CPAIOR (LNCS, Vol. 5547)*. Springer, 71–86. doi:10.1007/978-3-642-01929-6_7

[41] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. 2012. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.* 187 (2012), 52–89. doi:10.1016/J.ARTINT.2012.04.001

[42] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. 2012. Multi-threaded ASP solving with clasp. *Theory Pract. Log. Program.* 12, 4-5 (2012), 525–545. doi:10.1017/S1471068412000166

[43] Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca, and Torsten Schaub. 2018. Evaluation Techniques and Systems for Answer Set Programming: a Survey. In *IJCAI*. ijcai.org, 5450–5456. doi:10.24963/IJCAI.2018/769

[44] Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *ICLP/SLP*. MIT Press, 1070–1080.

[45] Philipp Hanisch and Markus Krötzsch. 2026. Rule Rewriting Revisited: A Fresh Look at Static Filtering for Datalog and ASP. *CoRR* abs/2601.05108 (2026). arXiv:2601.05108 doi:10.48550/ARXIV.2601.05108

[46] Patrick J Hayes. 1981. The frame problem and related problems in artificial intelligence. In *Readings in artificial intelligence*. Elsevier, 223–230.

[47] Marijn Heule and Torsten Schaub. 2015. What's Hot in the SAT and ASP Competitions. In *AAAI*. AAAI Press, 4322–4323. doi:10.1609/AAAI.V29I1.9348

[48] Marijn J. H. Heule. 2018. Schur Number Five. In *AAAI*. AAAI Press, 6598–6606. doi:10.1609/AAAI.V32I1.12209

[49] Arend Heyting. 1930. Die formalen Regeln der intuitionistischen Logik. *Sitzungsbericht PreuBische Akademie der Wissenschaften Berlin, physikalisch-mathematische Klasse II* (1930), 42–56.

[50] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. 2018. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *SAT*. 428–437. doi:10.1007/978-3-319-94144-8_26

[51] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290. doi:10.1145/505145.505149

[52] Henry A. Kautz and Bart Selman. 1999. Unifying SAT-based and Graph-based Planning. In *IJCAI*. Morgan Kaufmann, 318–325. http://ijcai.org/Proceedings/99-1/Papers/047.pdf

[53] Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. 2003. A Case for Efficient Solution Enumeration. In *SAT (LNCS, Vol. 2919)*. Springer, 272–286. doi:10.1007/978-3-540-24605-3_21

[54] Dominic Klyve and Lee Stemkoski. 2006. Graeco-Latin Squares and a Mistaken Conjecture of Euler. *The College Mathematics Journal* 37, 1 (2006), 2–15. doi:10.1080/07468342.2006.11922160

[55] Tuukka Korhonen, Jeremias Berg, and Matti Järvisalo. 2019. Enumerating Potential Maximal Cliques via SAT and ASP. In *IJCAI*. ijcai.org, 1116–1122. doi:10.24963/IJCAI.2019/156

[56] Markus Krötzsch. 2025. Modern Datalog: Concepts, Methods, Applications (Invited Paper). In *RW (OASIcs, Vol. 138)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:41. doi:10.4230/OASICS.RW.2024/2025.7

[57] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *ASE*. ACM, 110:1–110:13. doi:10.1145/3551349.3556938

[58] Joohyung Lee and Ravi Palla. 2009. System f2lp - Computing Answer Sets of First-Order Formulas. In *LPNMR (LNCS, Vol. 5753)*. Springer, 515–521. doi:10.1007/978-3-642-04238-6_51

[59] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.* 7, 3 (2006), 499–562. doi:10.1145/1149114.1149117

[60] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. 2016. Learning Rate Based Branching Heuristic for SAT Solvers. In *SAT (LNCS, Vol. 9710)*. Springer, 123–140. doi:10.1007/978-3-319-40970-2_9

[61] Yuliya Lierler. 2017. What is answer set programming to propositional satisfiability. *Constraints An Int. J.* 22, 3 (2017), 307–337. doi:10.1007/S10601-016-9257-7

[62] Vladimir Lifschitz. 2010. Thirteen Definitions of a Stable Model. In *Fields of Logic and Computation, Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday (LNCS, Vol. 6300)*. Springer, 488–503. doi:10.1007/978-3-642-15025-8_24

[63] Vladimir Lifschitz, Torsten Schaub, and Stefan Woltran. 2018. Interview with Vladimir Lifschitz. *Künstliche Intell.* 32, 2-3 (2018), 213–218. doi:10.1007/S13218-018-0552-X

[64] Fangzhen Lin and Yuting Zhao. 2002. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *AAAI*. AAAI Press / The MIT Press, 112–118. http://www.aaai.org/Library/AAAI/2002/aaai02-018.php

[65] Yanhong A. Liu and Scott D. Stoller. 2022. Recursive rules with aggregation: a simple unified semantics. *J. Log. Comput.* 32, 8 (2022), 1659–1693. doi:10.1093/LOGCOM/EXAC072

[66] Chris Martens, Robert J. Simmons, and Michael Arntzenius. 2025. Finite-Choice Logic Programming. *Proc. ACM Program. Lang.* 9, POPL (2025), 362–390. doi:10.1145/3704849

[67] John McCarthy. 1980. Circumscription - A Form of Non-Monotonic Reasoning. *Artif. Intell.* 13, 1-2 (1980), 27–39. doi:10.1016/0004-3702(80)90011-9

[68] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark W. Barrett. 2017. Relational Constraint Solving in SMT. In *CADE (LNCS, Vol. 10395)*. Springer, 148–165. doi:10.1007/978-3-319-63046-5_10

[69] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *DAC*. ACM, 530–535. doi:10.1145/378239.379017

[70] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *CP (LNCS, Vol. 4741)*. Springer, 529–543. doi:10.1007/978-3-540-74970-7_38

[71] Ilkka Niemelä. 1999. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Ann. Math. Artif. Intell.* 25, 3-4 (1999), 241–273. doi:10.1023/A:1018930122475
[72] Ilkka Niemelä. 2008. Answer Set Programming without Unstratified Negation. In *ICLP (LNCS, Vol. 5366)*. Springer, 88–92. doi:10.1007/978-3-540-89982-2_15
[73] Laurent Perron and Frédéric Didier. 2024. *CP-SAT*. Google. https://developers.google.com/optimization/cp/cp_solver/
[74] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2020. Provenance-guided synthesis of Datalog programs. *Proc. ACM Program. Lang.* 4, POPL (2020), 62:1–62:27. doi:10.1145/3371130
[75] Raymond Reiter. 1977. On Closed World Data Bases. In *Symposium on Logic and Data Bases (Advances in Data Base Theory)*. Plemum Press, 55–76. doi:10.1007/978-1-4684-3384-5_3
[76] Raymond Reiter. 1991. The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. (1991), 359–380. doi:10.1016/B978-0-12-450010-5.50026-8
[77] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstic. 2013. Finite Model Finding in SMT. In *CAV (LNCS, Vol. 8044)*. Springer, 640–655. doi:10.1007/978-3-642-39799-8_42
[78] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark W. Barrett. 2013. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In *CADE (LNCS, Vol. 7898)*. Springer, 377–391. doi:10.1007/978-3-642-38574-2_26
[79] Nicola Rizzo and Agostino Dovier. 2022. 3coSoKu and its declarative modeling. *J. Log. Comput.* 32, 2 (2022), 307–330. doi:10.1093/LOGCOM/EXAB086
[80] Marcus Rossel, Rudi Schneider, Thomas Koehler, Michel Steuwer, and Andrés Goens. 2026. Towards Pen-and-Paper-Style Equational Reasoning in Interactive Theorem Provers by Equality Saturation. *Proc. ACM Program. Lang.* 10, POPL (2026), 718–747. doi:10.1145/3776667
[81] Irfansha Shaik and Jaco van de Pol. 2024. Optimal Layout Synthesis for Deep Quantum Circuits on NISQ Processors with 100+ Qubits. In *SAT (LIPIcs, Vol. 305)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26:1–26:18. doi:10.4230/LIPICS.SAT.2024.26
[82] João P. Marques Silva and Karem A. Sakallah. 1999. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers* 48, 5 (1999), 506–521. doi:10.1109/12.769433
[83] Patrik Simons, Ilkka Niemelä, and Timo Soininen. 2002. Extending and implementing the stable model semantics. *Artif. Intell.* 138, 1-2 (2002), 181–234. doi:10.1016/S0004-3702(02)00187-X
[84] Mate Soos, Karsten Nohl, and Claude Castelluccia. 2009. Extending SAT Solvers to Cryptographic Problems. In *SAT (LNCS, Vol. 5584)*. Springer, 244–257. doi:10.1007/978-3-642-02777-2_24
[85] Bochen Tan and Jason Cong. 2020. Optimal Layout Synthesis for Quantum Computing. In *ICCAD*. IEEE, 137:1–137:9. doi:10.1145/3400302.3415620
[86] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *TACAS (LNCS, Vol. 4424)*. Springer, 632–647. doi:10.1007/978-3-540-71209-1_49
[87] Grigori S Tseitin. 1983. On the complexity of derivation in propositional calculus. In *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*. Springer, 466–483.
[88] Marina De Vos and Dirk Vermeir. 1999. On the Role of Negation in Choice Logic Programs. In *LPNMR (LNCS, Vol. 1730)*. Springer, 236–246. doi:10.1007/3-540-46767-X_17
[89] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing Recursive Queries with Progam Synthesis. In *SIGMOD*. ACM, 79–93. doi:10.1145/3514221.3517827
[90] Ziyi Yang and Ilya Sergey. 2025. Inductive Synthesis of Inductive Heap Predicates. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025), 169–195. doi:10.1145/3720420
[91] Ziyi Yang and Ilya Sergey. 2026. *SetLah!: Semi-Declarative Language for Combinatorial Search (OOPSLA 2026 Artifact)*. doi:10.5281/zenodo.21515983
[92] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI (2023), 468–492. doi:10.1145/3591239
[93] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. 2015. *Constraint Solving and Planning with Picat*. Springer. doi:10.1007/978-3-319-25883-6

Received 2026-03-17; accepted 2026-08-06