Reasoning about Constraint Formulations in Lean

Pablo Manrique
TU Wien, Austria

Stefan Szeider
TU Wien, Austria

Abstract

In this short paper, we present a Lean 4 library that formalizes constraint satisfaction problems (CSPs) and provides a framework for formally proving properties on general formulations, such as equivalence, equisatisfiability, and the correctness of symmetry-breaking constraints. These properties hold for general parameterized problem families, not just for specific instances: for example, we verify the equivalence of two different formulations and the correctness of a symmetry-breaking constraint for the n-queens problem for all values of n. We demonstrate the applicability of our approach by providing translation backends to the MiniZinc and SMT-LIB formats, enabling an end-to-end workflow in which we formally verify lower bounds on Schur numbers by using an external solver to find a solution and then checking it inside Lean.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Logic and verification

Keywords and phrases Constraint Satisfaction Problems, modelling, verification, Lean

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding *Pablo Manrique*: This research was funded in whole or in part by the Austrian Science Fund (FWF) 10.55776/COE12.

1 Introduction

Constraint satisfaction problems (CSPs) constitute a powerful framework for modeling and solving a wide range of combinatorial problems, from scheduling and resource allocation to hardware verification and combinatorial design [14]. In practice, practitioners often reformulate CSPs to improve solver performance, adding symmetry-breaking constraints or switching between alternative encodings [15]. As an elementary example, the n-queens problem admits two standard encodings: one with n^2 Boolean variables representing the presence of a queen in each cell, and another with n integer variables representing the column position of the queen in each row. Moreover, different symmetry-breaking constraints can be added to the problem, such as fixing the position of the first queen to the first half of the board, which reduces the search space by pruning symmetric solutions (see Figure 1). At this point, errors can silently eliminate solutions or introduce spurious ones. How can we *ensure* that two different formulations encode the same problem, or that the added symmetry-breaking constraints only eliminate symmetric solutions without affecting satisfiability? This is where formal verification, and in particular the use of theorem provers, can help.

There are previous works that already use automated and interactive theorem provers (ATPs and ITPs) to obtain general formal guarantees in related combinatorial solving approaches. In Rocq, a certified constraint solver [6] was implemented to ensure correctness by construction, but this comes at the cost of performance. Verification of specific SAT encodings has been carried out for AI planning in Isabelle/HOL [1], for the Boolean Pythagorean Triples problem in Rocq [9], and other problems in Lean [7], such as the Empty Hexagon Number [16]. ITPs allow reasoning over parameterized problem families, establishing correctness once for all instances. ATPs have also been used to reason about constraint problems at the

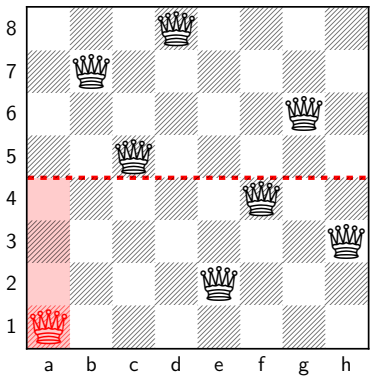

Figure 1 8-queens solution with symmetry-breaking: the first queen is fixed to rows 1-4 (highlighted).

specification level [5, 3]; however, their degree of automation is typically not sufficient to obtain fully automated proofs of non-trivial correctness properties. As a result, significant guidance is still required to complete the verification process.

Our contribution follows this approach, but rather than implementing a solver or focusing on SAT encodings, we work with an ITP at the CSP specification level: we implement LeanCSP, a library in Lean 4 [12] that allows us to formulate constraint problems by specifying variables, the value domains of those variables, and the constraints that relate them. We formalize the concepts of equivalence, equisatisfiability, and symmetry-breaking constraints in CSPs. The expressivity of Lean allows us to define both parameterized problem families (e.g., the n-queens problem for any n) and instantiated CSPs with these parameters fixed to specific values. To demonstrate the applicability of LeanCSP to real problems, we provide over 40 constraint patterns, including global constraints, cardinality constraints, and Boolean gates, that enable us to formulate different CSPs as a proof of concept. More specifically, we prove the correctness of reformulations and added symmetry-breaking constraints for five different parameterized problem families: *n-queens*, *graph coloring*, *Latin squares*, *Sudoku*, and *Schur numbers*.

Additionally, we implement two translation backends that can generate MiniZinc [13] and SMT-LIB [2] code from CSPs defined using LeanCSP after they are instantiated with specific parameter values. This allows us to solve the resulting problems with external solvers and, for satisfiable instances, solver-produced witnesses can be loaded back into Lean and checked by its kernel. We demonstrate an end-to-end pipeline on the Schur numbers problem, where we prove the correctness of a symmetry-breaking constraint of the CSP formulation and then use an external solver to obtain a solution which is finally checked inside Lean, obtaining Lean-verified lower bounds for $S(2)$, $S(3)$, and $S(4)$.

2 Formalization in Lean

Our formalization is carried out in Lean 4 [12], a proof assistant and functional programming language based on dependent type theory. It lets users construct proofs interactively using tactics to manipulate the proof state, and a small, trusted kernel checks the validity of the resulting proof terms. Its extensive Mathlib [11] library and a growing ecosystem of tools enable the formalization and verification of theorems. In this section, we formalize

constraint satisfaction problems (CSPs) and properties such as equivalence, equisatisfiability, and symmetry-breaking constraints in Lean, as well as use its programming capabilities to implement translation backends that generate code for external solvers. We assume only a passing familiarity with Lean's syntax and refer the reader to the official documentation for more details [12].

2.1 Basic definitions

A constraint satisfaction problem consists of variables, domains, and constraints. Following the standard definition, a CSP $\mathcal{P} = (X, D, C)$ comprises variables $X = (x_1, \ldots, x_n)$, domains $D = (D_1, \ldots, D_n)$ with $x_i \in D_i$, and constraints $C = (C_1, \ldots, C_m)$ where each C_j is a pair (S_j, R_j) of a scope $S_j \subseteq X$ and a relation R_j over the domains of variables in S_j.

As we mentioned earlier, Lean is based on dependent type theory. For this reason, it is more natural to define CSPs in Lean using dependent types rather than sets and tuples as in the standard CP definition above. More precisely, given any type representing the variables `VarIndex`, and a function `DomainType` mapping each variable to its (arbitrary) domain type, we define a CSP as a list of constraints:

```
abbrev CSP (VarIndex : Type) (DomainType : VarIndex → Type) :=
  List (DynamicConstraint VarIndex DomainType)
```

Since all elements of a list must have the same type, we use the `DynamicConstraint` wrapper type that allows expressing constraints without fixing their arity. The underlying `Constraint` type is defined by a scope and a checker function that expresses the relation:

```
structure Constraint (VarIndex : Type) (DomainType : VarIndex → Type) (n : ℕ)
    where
  scope : Vector VarIndex n
  check : ScopeValues VarIndex DomainType scope → Bool
```

Here, `ScopeValues` is a dependent function that maps each scope index to the corresponding domain type. Intuitively, it just represents the product $\prod_{x_i \in S_j} D_i$.

Other basic concepts regarding CSPs are also formalized: an assignment is a function that maps variables to values in their respective domains; a solution of a CSP is an assignment that satisfies all constraints of the problem; and a CSP is satisfiable if it has a solution.

2.2 CSP equivalence and equisatisfiability

Defining precisely what it is for a CSP to *encode* or *represent* a given problem, or what we mean by saying that two CSPs are equivalent, may not be that trivial [14]. A usual definition of CSP equivalence would consider two CSPs to be equivalent if they have the same sets of solutions. However, this definition could lead, for instance, to the conclusion that changing the names of the variables in a CSP produces a non-equivalent CSP, while the *information* those problems encode remains the same. We thus use a different definition of equivalence: two CSPs are equivalent if there exists a bijection between their solution sets. In Lean, the definition is the following, where `sol_set` is the set of solutions of a CSP:

```
def equivalent (csp₁ : CSP VarIndex₁ DomainType₁)
    (csp₂ : CSP VarIndex₂ DomainType₂) : Prop :=
  ∃ f : sol_set csp₁ → sol_set csp₂, Function.Bijective f
```

For practical reasons when developing proofs of equivalence between specific CSPs, we introduce a characterization of equivalence. Given two CSPs $\mathcal{P}$ and $\mathcal{P}'$, and a projection π that maps assignments of $\mathcal{P}'$ to assignments of $\mathcal{P}$, we say that $\mathcal{P}$ and $\mathcal{P}'$ are π-equivalent if:

1. Solutions of $\mathcal{P}'$ map to solutions of $\mathcal{P}$,
2. Every solution of $\mathcal{P}$ has a preimage, and
3. π is injective on solutions.

If we prove that two CSPs $\mathcal{P}$ and $\mathcal{P}'$ are π-equivalent, then they are equivalent via the bijection $\pi|_{\text{sol}(\mathcal{P}')}$. The π-equivalence definition is slightly more convenient to use when reasoning on concrete CSPs, since it is usually easier to define a map between the assignment sets explicitly than between the solution sets, where constraint satisfaction is implicit.

Under the natural assumption that the assignment type of $\mathcal{P}$ is non-empty, i.e., every variable has at least one domain value, we prove that two CSPs are equivalent if and only if they are π-equivalent.

```
theorem equivalent_iff_pi_equivalent [Nonempty (Assignment VarIndex₁ DomainType₁)]
    (csp₁ : CSP VarIndex₁ DomainType₁) (csp₂ : CSP VarIndex₂ DomainType₂) :
    equivalent csp₂ csp₁ ↔ ∃ π, pi_equivalent csp₁ csp₂ π
```

On the other hand, the definition of equisatisfiability that we use is the natural one: two CSPs are equisatisfiable if both are satisfiable or both are unsatisfiable.

```
def equisatisfiable (csp₁ : CSP VarIndex₁ DomainType₁)
    (csp₂ : CSP VarIndex₂ DomainType₂) : Prop :=
  is_satisfiable csp₁ ↔ is_satisfiable csp₂
```

Finally, we prove that equivalence implies equisatisfiability, as it is natural to expect:

```
theorem equivalent_implies_equisatisfiable (csp₁ : CSP VarIndex₁ DomainType₁)
    (csp₂ : CSP VarIndex₂ DomainType₂) :
  equivalent csp₁ csp₂ → equisatisfiable csp₁ csp₂
```

2.3 Symmetries and symmetry-breaking constraints

Symmetry in CSPs has been widely studied, and many definitions have appeared in the literature [8]. We define a *symmetry* as a permutation on the variables or the domain sets that preserves solutions, and a *symmetry-breaking constraint* as a constraint such that, given a solution of the original problem, there exists a symmetry that transforms it into a solution of the extended CSP that includes the new constraint.

We distinguish between *variable symmetries*, which are permutations of the variable indices, and *domain symmetries*, which are permutations of the values in the domains. A variable symmetry β is a bijection on the variable index set such that for any solution, permuting the variables according to β yields another solution.

```
def VariableSymmetry (csp : CSP VarIndex DomainType) (β : Equiv VarIndex VarIndex)
    (h_types : DomainTypePreserving (DomainType := DomainType) β) : Prop :=
  ∀ assignment, is_solution csp assignment →
    is_solution csp (applyVariablePerm β assignment h_types)
```

Note that, since the value domains may be different for each variable, `DomainTypePreserving` ensures that for every variable `v`, `DomainType (β v) = DomainType v`, while `applyVariablePerm` applies the variable permutation to the assignment, i.e., intuitively it computes the composition of the assignment with the variable permutation β.

Domain symmetries are defined similarly, but instead of permuting variable indices, they permute the values in the domains. Since, in the case of domains, they are not necessarily the same for all variables, we need to provide permutations for the domain of each variable. In Lean, we do this by defining a domain equivalence (or permutation) family:

```
def DomainEquivFamily (VarIndex : Type) (DomainType : VarIndex → Type) : Type :=
  (v : VarIndex) → Equiv (DomainType v) (DomainType v)
```

In order to apply a domain permutation family δ to an assignment α, we need to map each variable v to the value $\delta(v)(\alpha(v))$ (this is what the `apply` function does in Lean). A domain symmetry is then a domain equivalence family δ such that for any solution, applying δ to it yields another solution.

```
def DomainSymmetry (csp : CSP VarIndex DomainType)
    (δ : DomainEquivFamily VarIndex DomainType) : Prop :=
  ∀ assignment, is_solution csp assignment → is_solution csp (δ.apply assignment)
```

Once the variable and domain symmetries on CSPs are defined, we can define symmetry-breaking constraints. A symmetry-breaking constraint is a constraint such that, given a solution of the original problem, there exists a symmetry (variable or domain) such that applying it to the solution results in a solution of the CSP that includes the old plus the new constraint. We define domain symmetry-breaking constraints in the following way (the variable case is analogous):

```
def domainSymmetryBreakingConstraint (csp : CSP VarIndex DomainType)
    (c : DynamicConstraint VarIndex DomainType) : Prop :=
  ∀ assignment : Assignment VarIndex DomainType, is_solution csp assignment →
    ∃ δ : DomainEquivFamily VarIndex DomainType,
      DomainSymmetry csp δ ∧
      is_solution (extended_csp csp c) (δ.apply assignment)
```

Finally, we can prove that adding a symmetry-breaking constraint to a CSP preserves satisfiability, since the underlying symmetry transforms solutions of the original problem into solutions of the extended problem:

```
theorem domainSymmetryBreaking_equisatisfiability (csp : CSP VarIndex DomainType)
    (c : DynamicConstraint VarIndex DomainType)
    (h_sbc : domainSymmetryBreakingConstraint csp c) :
  equisatisfiable csp (extended_csp csp c)
```

2.4 Translation backends

To bridge our formalization with external solvers, we provide translation backends to the MiniZinc and SMT-LIB formats. This feature allows us to efficiently solve the problems we reason about, for instance, CSPs that include symmetry-breaking constraints that are formally verified. The backends operate on the particular case of CSPs where every variable shares the integer domain $\mathbb{Z}$, and the lower and upper bound delimiting each variable's search space are encoded as ordinary *bound constraints*, that is, unary constraints on the single variable they bound. A broad class of combinatorial problems fits this fragment naturally, including all the case studies of Section 3, and we leave support for richer value types as future work.

We provide a fixed library of over 40 *constraint patterns*, organized into five families: comparison constraints (`eq`, `ne`, `lt`, `le`, ...), arithmetic constraints (linear combinations,

products, absolute values), global constraints (`alldifferent`, cardinality, `element`), Boolean gates and their reified variants, and the bound constraints already mentioned. Each pattern has a target-specific encoding in both backends. For instance, `alldifferent` maps to the MiniZinc global of the same name and to `distinct` in SMT-LIB. Bound constraints become variable declarations `var lb..ub: x` in MiniZinc and range assertions in SMT-LIB, while arithmetic and Boolean patterns are emitted compositionally.

Translation is performed on instantiated CSPs rather than on parameterized problem families: once concrete values for the possibly open parameters have been supplied, the CSP can be exported as a single string in the target format and passed to a MiniZinc or SMT solver.

3 Case studies

We illustrate the library on five combinatorial problems: n-queens, graph coloring, Latin squares, Sudoku, and the Schur numbers problem. For each, we prove that two CSP formulations are equivalent in the sense of Section 2.2: a *compact* formulation that uses fewer variables over larger domains, and a *one-hot* formulation that uses Boolean-valued integer variables (with `false` and `true` represented by `0` and `1`), closer to a classical SAT encoding. The equivalence proof goes through the π-equivalence characterization: we define a projection π from one-hot assignments to compact ones, and discharge the forward direction (π preserves solutions), the backward direction (every compact solution lifts to a one-hot solution), and the injectivity of π.

For each problem, we also define symmetry-breaking constraints on the compact encoding and prove their correctness. We first define a function on assignments and prove it is a variable or domain symmetry of the base CSP. We then prove that an additional constraint is symmetry-breaking by fixing an arbitrary solution of the base CSP and exhibiting a symmetry that maps it to a solution of the extended CSP: if the solution already satisfies the new constraint, the identity suffices; otherwise, we apply the symmetry function defined earlier and check that it preserves all original constraints (because it is a symmetry) and now satisfies the new one as well.

We work through n-queens in detail. The compact encoding (`nqueens_csp1D`) uses n integer variables ranging over $\{0, \ldots, n-1\}$, where x_i is the row of the queen in column i, with all-different constraints on rows and on positive and negative diagonals. The one-hot encoding (`nqueens_csp2D`) uses n^2 Boolean variables together with cardinality constraints (exactly one queen per row and column, at most one per diagonal). The projection π sets $x_i := j$ whenever $x_{ij} = 1$; this is well-defined on solutions because the cardinality constraints force exactly one such j for every i. We obtain:

```
theorem nqueens_equivalent (hn_pos : 0 < n) :
    equivalent (nqueens_csp2D n) (nqueens_csp1D n)
```

For symmetry breaking on the compact encoding, we use the horizontal reflection $d \mapsto n - 1 - d$ on the row domain, prove it is a domain symmetry, and add the constraint `sb_constraint` that the first queen's row is at most $\lfloor (n+1)/2 \rfloor$. The case distinction described above then yields the correctness of this constraint as a symmetry-breaking constraint for the base CSP:

```
theorem sb_constraint_is_domain_symmetry_breaking (n : ℕ) (h_n : 0 < n) :
    domainSymmetryBreakingConstraint (nqueens_csp n) (sb_constraint n h_n)
```

By applying the last theorem in Section 2.3, this gives equisatisfiability between the base CSP and the extended one with the symmetry-breaking constraint added. Note that each of the theorems above is proved once and for all for every $n > 0$, with no instance-specific reasoning.

The same methodology applies to the other four problems with problem-specific variations. *Graph coloring* compares a one-variable-per-vertex encoding with a $|V| \times C$ Boolean matrix, breaking domain symmetry by fixing the first vertex's color. *Latin squares* relate n^2-variable and n^3-Boolean formulations and break column-permutation symmetry by forcing the first row to be sorted. *Sudoku* extends Latin squares with box-uniqueness constraints and reuses much of the proof infrastructure. The *Schur numbers* problem partitions the set $\{1, \ldots, n\}$ into c color classes avoiding monochromatic solutions to $i + j = k$, contrasting an n-variable compact encoding with an $n \times c$ Boolean one-hot variant, and breaks domain symmetry by fixing the first integer's color.

4 End-to-end pipeline on Schur numbers

In this section, we illustrate how our library on CSPs can be combined with external CP and SMT solvers to obtain end-to-end verified mathematical results in Lean, with its kernel as the only trusted component of the pipeline. The parametric properties established in earlier sections are proved once and for all inside Lean. After a CSP is instantiated with concrete parameter values, the translation backends of Section 2.4 export the CSP to MiniZinc or SMT-LIB, an external solver is invoked, and the returned certificate is re-checked back inside Lean. At present, we only handle the satisfiable case, where the certificate is an explicit witness. This already suffices for natural decision questions such as the Schur-number lower bound $S(c) \geq n$, which we use to illustrate the workflow.

Schur's theorem [10] states that for every $c > 0$ there is a largest integer $S(c)$ for which $\{1, \ldots, S(c)\}$ admits a c-colouring with no monochromatic solution to $i + j = k$. We define a CSP `schur_csp` parametrised by the size of the set $\{1, \ldots, n\}$ and the number of colours c. It uses n integer variables ranging over $\{0, \ldots, c-1\}$ and, for each triple (i, j, k) such that $i + j = k$, the disjunction $x_i \neq x_j \vee x_j \neq x_k \vee x_k \neq x_i$, representing that they do not all share the same color. A Lean theorem `schur_csp_iff_colorable`, parameterized by n and c, bridges this CSP with the mathematical claim: it states that the CSP is satisfiable if and only if $\{1, \ldots, n\}$ admits a c-coloring with no monochromatic $i + j = k$, which is exactly $S(c) \geq n$. In Lean, this high-level mathematical property is the one expressed in the following theorem:

```
theorem S4_ge_44 :
    ∃ χ : Fin 44 → Fin 4, ∀ (i j k : Fin 44),
      ¬(k.val = i.val + j.val + 1 ∧ χ i = χ j ∧ χ j = χ k)
```

The proof of this result is divided into the following steps: (i) by `schur_csp_iff_colorable`, it suffices to show that the base CSP `schur_csp 44 4` is satisfiable; (ii) by equisatisfiability with the extended CSP that adds the symmetry-breaking constraint of Section 3, it then suffices to show that the extended CSP is satisfiable; (iii) we translate the extended CSP (with an additional symmetry-breaking constraint to speed up solving) to MiniZinc or SMT-LIB via the backends of Section 2.4 and pass it to an external CP or SMT solver (we used Gecode, Chuffed, Z3 and CVC5 independently), which returns a candidate assignment; (iv) we parse the obtained solution back into Lean and check it against the extended CSP, finally proving that it is satisfiable. Following this workflow, we obtained Lean-verified proofs of the bounds $S(2) \geq 4$, $S(3) \geq 13$, and $S(4) \geq 44$.

5 Trust assumptions and proof effort

All the reformulation theorems of Section 3, namely the equivalence of compact and one-hot encodings, the correctness of each candidate symmetry-breaking constraint, and the equisatisfiability under symmetry-breaking, are stated and Lean-checked against the CSP semantics of Section 2. They depend only on `mathlib` and the standard Lean environment and introduce no project-specific axioms. Although the translation backends to MiniZinc and SMT-LIB of Section 2.4 only emit text and are not themselves verified, in the presented end-to-end pipeline in Section 4, however, the solver-produced assignment is parsed back and checked inside Lean against the original CSP, so that the translator and the external solver act only as untrusted oracles and stay outside the trusted base.

Each equivalence and symmetry proof is carried out once per problem family and holds for every instance of that family. For example, the Schur development of Section 3 is reused unchanged for all three bounds $S(2) \geq 4$, $S(3) \geq 13$, and $S(4) \geq 44$ of Section 4, independently of the values of n and c. Setting proofs aside, the definitions alone of the formalization, both at the library and at the per-problem level, sum up to about 2000 lines of Lean code. On top of these, and across the five case studies, the equivalence proofs range from roughly 370 lines of Lean code for Sudoku, which reuses lemmas from Latin squares, to about 1400 lines for n-queens, where the diagonal constraints require global geometric reasoning to match an offset `alldifferent` against a cardinality constraint over the diagonals. The symmetry-breaking correctness proofs are noticeably shorter, in the range of 100 to 230 lines per problem, and the bridge theorem `schur_csp_iff_colorable` used in Section 4 adds fewer than 200 additional lines. Most of this effort lies in constraint-specific lemmas that establish how each constraint behaves under the projection π and under the candidate symmetry.

6 Conclusions and future work

We have presented LeanCSP, a Lean 4 library for formalizing constraint satisfaction problems and reasoning about their properties at the specification level. Equivalence, equisatisfiability, and symmetry-breaking correctness are established parametrically for problem families, and all generated instances inherit these guarantees automatically. We demonstrated this approach by reasoning on five different problems: n-queens, graph coloring, Latin squares, Sudoku, and Schur numbers, proving the equivalence of compact and one-hot formulations and the correctness of symmetry-breaking constraints for each family. Moreover, we show that CSPs and the concepts formalized in our library can be applied to the verification of more general results in Lean, using the introduced end-to-end pipeline. It translates CSPs to external formats that solvers can understand, and the produced witnesses can be loaded back into Lean and checked, with only the Lean kernel in the trusted base.

Our current end-to-end guarantees cover the satisfiable case, where a solution can be checked directly within Lean. Extending the pipeline to unsatisfiable instances will require checking solver-produced unsatisfiability certificates (such as DRAT, VeriPB, or similar proof formats) inside Lean. Recent work on verified proof checkers in Lean [4, 17] provides a foundation for this direction, and combining it with our parametric specification-level proofs would yield end-to-end formal guarantees for both the satisfiable and unsatisfiable cases. On the modelling side, the current formalization covers variable and domain symmetries, while variable-value symmetries are left for future work.

References

1 Mohammad Abdulaziz and Friedrich Kurz. Formally verified SAT-based AI planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(12):14665–14673, Jun. 2023. URL: https://ojs.aaai.org/index.php/AAAI/article/view/26714, doi:10.1609/aaai.v37i12.26714.

2 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.7. Technical report, Department of Computer Science, The University of Iowa, 2025. Available at www.SMT-LIB.org.

3 Christian Bessiere, Emmanuel Hebrard, George Katsirelos, Zeynep Kiziltan, Nina Narodytska, and Toby Walsh. Reasoning about constraint models. In Duc-Nghia Pham and Seong-Bae Park, editors, *PRICAI 2014: Trends in Artificial Intelligence*, pages 795–808, Cham, 2014. Springer International Publishing.

4 Henrik Böving and Josh Clune. Interactive bit vector reasoning using verified bitblasting. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2025.

5 Marco Cadoli and Toni Mancini. Using a theorem prover for reasoning on constraint problems. In Stefania Bandini and Sara Manzoni, editors, *AI*IA 2005: Advances in Artificial Intelligence*, pages 38–49, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

6 Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. A certified constraint solver over finite domains. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 116–131, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

7 Cayden R. Codel, Jeremy Avigad, and Marijn Heule. Verified encodings for SAT solvers. *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pages 141–151, 2023. URL: https://api.semanticscholar.org/CorpusID:265859217.

8 David Cohen, Peter Jeavons, Christopher Jefferson, Karen Petrie, and Barbara Smith. Symmetry definitions for constraint satisfaction problems. volume 11, pages 17–31, 10 2005. doi:10.1007/11564751_5.

9 Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Formally verifying the solution to the boolean pythagorean triples problem. *J. Autom. Reason.*, 63(3):695–722, October 2019. doi:10.1007/s10817-018-9490-4.

10 Ronald L. Graham and Bruce L. Rothschild. *Ramsey theory (2nd ed.)*. Wiley-Interscience, USA, 1990.

11 The mathlib Community. The Lean mathematical library. https://github.com/leanprover-community/mathlib4, 2020–.

12 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-79876-5_37.

13 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

14 Francesca Rossi, P. V. Beek, and Toby Walsh. Handbook of constraint programming. In *Handbook of Constraint Programming*, 2006. URL: https://api.semanticscholar.org/CorpusID:14174798.

15 Barbara M. Smith. Modelling. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, pages 377–406. Elsevier, Amsterdam, 2006.

16 Bernardo Subercaseaux, Wojciech Nawrocki, James Gallicchio, Cayden Codel, Mario Carneiro, and Marijn J. H. Heule. Formal verification of the empty hexagon number, 2024. URL: https://arxiv.org/abs/2403.17370, arXiv:2403.17370.

17 Stefan Szeider. PBLean: Pseudo-boolean proof certificates for Lean 4, 2026. arXiv:2602.08692.