Proof Logging for Projected Enumeration (and Counting?) Problems in VeriPB

Ciaran McCreesh
University of Glasgow, Scotland

Jakob Nordström
University of Copenhagen, Denmark
Lund University, Sweden

Andy Oertel
Lund University, Sweden
University of Copenhagen, Denmark

Yong Kiam Tan
Nanyang Technological University, Singapore
Institute for Infocomm Research (I^2R), A*STAR, Singapore

Abstract

When a certifying solver claims that a solution is optimal or that a problem is unsatisfiable, it demonstrates this convincingly by giving a proof log which can be checked by an independent (and ideally formally verified) proof checker. Such an approach should also be viable for enumeration problems ("I have listed all solutions explicitly") and counting problems ("there are exactly 42 solutions"), but the currently most popular proof logging systems contain several vital features which are incompatible with this goal. We explain how the VeriPB system can be modified for enumeration and counting proofs whilst retaining as much as possible of its powerful "strengthening" and "deletion" features. We implement this extension both inside VeriPB's user-friendly proof checker and elaborator and the formally verified CakePB backend, and use this to obtain formally verified enumerations of solutions for a range of constraint solving and graph problem instances.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Discrete optimization

Keywords and phrases Constraint programming, proof logging, auditable solving

Digital Object Identifier 10.4230/LIPIcs.CP.2026.54

Supplementary Material Source code for the tools described in this paper can be found here:
Software (Glasgow Constraint Solver): `https://zenodo.org/records/20089244`
Software (VeriPB): `https://gitlab.com/MIAOresearch/software/VeriPB`
archived at `swh:1:rev:6d38dab246af9c321b8f17cb5a187f2fbb9e491d`
Software (CakePB): `https://gitlab.com/MIAOresearch/software/cakepb`
archived at `swh:1:rev:a7593ef22de2fc0b47a688f2d4f08e6b742735af`

Funding *Ciaran McCreesh*: Supported by a Royal Academy of Engineering research fellowship, by the Engineering and Physical Sciences Research Council [grant number EP/X030032/1], and by the Advanced Research and Invention Agency.
Jakob Nordström: Swedish Research Council grant 2024-05801 and Independent Research Fund Denmark grant 9040-00389B
Andy Oertel: Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation
Yong Kiam Tan: Singapore NRF Fellowship Programme NRF-NRFF16-2024-0002

Acknowledgements We would like to thank the anonymous reviewers of *CP* for their helpful comments. We are grateful to Benjamin Bogø, Wietze Koops, and Matthew McIlree for valuable discussions. Also, we are thankful for stimulating conversations with and useful feedback from

participants of the extended reunion of the program *Satisfiability: Theory, Practice, and Beyond* at the Simons Institute for the Theory of Computing at UC Berkeley and the Dagstuhl workshop 25231 *Certifying Algorithms for Automated Reasoning*. The second and third authors wish to acknowledge that they have benefited greatly from being part of the Basic Algorithms Research Copenhagen (BARC) environment financed by the Villum Investigator grant 54451.

1 Introduction

Modern solvers for combinatorial and automated reasoning problems are able to solve many large and challenging problem instances quickly, but their complexity means that sometimes bugs lead to incorrect answers being produced. The most successful remedy to this issue is to make solvers *certifying*: that is, to have them output a mathematical *proof log* detailing how a conclusion was reached. This proof log can then be checked independently by a proof-checking tool, and ideally this tool will be sufficiently simple that it can be formally verified. Proof logging is a standard feature for Boolean satisfiability (SAT) solvers, and is now being adopted in other application areas that use richer data types and constraints.

In SAT proof logging systems operating on formulas in conjunctive normal form (CNF), a proof is a sequence of clausal constraints, each of which can easily be checked to maintain satisfiability when added to the input formula. Therefore, if we can derive an unsatisfiable constraint (in the form of the empty clause without literals), then we have shown that the original problem has no solutions. Ideally, a proof system should also make it easy for solver authors to generate proofs. For SAT solvers using conflict-driven clause learning (CDCL), a proof can simply be the list of inferred clauses, in the order they are derived [30, 33, 38].

For more general problems, such as those with non-Boolean variables or with non-clausal constraints, two kinds of approach have been proposed. The first involves extending the proof format with new rules directly corresponding to every kind of constraint and inference used by every individual solver (e.g. [5, 20, 21, 59]). The second instead adds only a few simple but very general rules to the proof format, in the hope that these are powerful enough to express all of the reasoning carried out by many different solvers. The VeriPB system we discuss in this paper adopts this latter approach, and it has been used to certify a range of constraint programming techniques [28, 44, 45, 46], dynamic programming [15], graph solving [25, 26, 27], advanced SAT and MaxSAT solving [6, 29, 36, 58], pseudo-Boolean solving [24, 41], and classical planning [17]. Critically, although VeriPB itself operates with 0–1 linear inequalities, or *pseudo-Boolean (PB) constraints*, the system can be used to certify solvers that carry out general constraint reasoning, without requiring these solvers to modify their reasoning or to be aware of the underlying pseudo-Boolean representation when solving. To enable this, VeriPB proof logs can include explicit derivations of auxiliary constraints that can be used to express non-trivial inferences, e.g., all-different consequences [18], and other complex forms of reasoning, without requiring new bespoke rules in the proof format.

In practice, solver users do not just care about unsatisfiable decision problem instances. Demonstrating *satisfiability* for NP-complete problems is not difficult: by definition, these problems will have an easily checkable witness. For proofs of optimality for single-objective optimisation problems, we can combine these two concepts by providing a solution achieving some objective value together with a proof of unsatisfiability when the requirement to produce a strictly better solution is added. However, for problems involving solution *enumeration* (explicitly listing all solutions) or *counting* (determining how many solutions there are, without requiring an explicit list), the situation is more intricate. These kinds of problems arise in areas such as computer algebra, graph theory, combinatorics, and

physics (e.g. [2, 12, 16, 35, 37, 40, 55, 57]).

Proof logs for enumeration and counting problems must reason carefully to preserve the full set of solutions. But the proof systems used in practice for SAT solving can only prove unsatisfiability, since they allow unrestricted deletions of constraints that can introduce spurious solutions. And even if deletions are handled more carefully, as in VeriPB, so-called *strengthening rules* used to justify without-loss-of-generality reasoning can change the set of solutions arbitrarily as long as at least one solution is preserved. To the best of our knowledge, there has not existed any proof logging system even for SAT that could support efficient proofs for enumeration problems.

In this work, we present, for the first time, an efficient and formally verified toolchain for certified solution enumeration, and show how this can be achieved while still supporting the strengthening and deletion rules in state-of-the-art proof logging. We start by elaborating in Section 2 on why enumeration proofs are not straightforward. We then describe in Section 3 an extension to the VeriPB system (including its formally verified CakePB backend) that supports such proofs, and evaluate our implementation on a range of problem instances. In Section 4, we introduce further extensions for preprocessing and for counting problems, after which we conclude with some final remarks in Section 5.

2 Proof Logging Beyond Implicational Proofs of Unsatisfiability

Extending a simple implicational proof system to support optimisation can be relatively simple, if the proof system has some natural way of representing integer inequalities (though most proof systems used for SAT solving cannot easily express this, since they are based on clausal constraints, and this is an important reason that proof logging for SAT-based optimisation is not yet well-developed). Perhaps the most straightforward approach is to add a "solution-improving" rule to the proof language. Such a rule takes a witness solution, which the proof checker validates satisfies all constraints, and introduces a new constraint saying that from now on the objective function must evaluate to a better value than that achieved by the witness. A proof of optimality ends by a derivation of contradiction, effectively certifying that "there is a solution with objective value X, and the instance is unsatisfiable if you want an objective value better than X".

A similar scheme could be used for enumeration proofs: we could add a "solution-excluding" proof rule with a witness solution, where the proof checker would first validate the witness and then introduce a *blocking constraint* forbidding this particular assignment. In this setting, a proof of unsatisfiability would be saying "there are no solutions, other than the ones listed by solution-excluding steps." Indeed, the very first version of VeriPB supported a limited form of such proofs, and this was used to check results for maximal clique enumeration problem instances that were misreported in the literature [25] (but the issue of spurious or repeated solutions was sidestepped by just assuming that the user would not delete any constraints that could trigger such a scenario).

An immediate limitation of this kind of enumeration proof is that there may be more solutions to an encoded problem seen by a solver or proof checker than there are to the real-world problem the user actually cares about. This can occur for, e.g. subgraph or constraint programming problems when the encoding of the high-level problem to pseudo-Boolean form is not *parsimonious* or fails to be *arc consistent* in the SAT sense of the word. For some encodings, this can be solved by *projection* [23]: informally, by specifying a *preserved (sub)set* of variables, and then asking for unique solutions with respect to that set. Inside a proof, the blocking constraint would then be introduced only with respect to those variables (although

for soundness, the witnessing assignment must still demonstrate satisfiability with respect to all variables and constraints). It turns out that dealing with projection will be helpful for the remainder of this paper, even for problems where there is a well-behaved encoding, and so we will assume we are always working in a projected setting (where the preserved set is "all of the variables" for simple enumeration problems).

So with this established, why do proofs for enumeration remain a problem? It turns out that most practical proof systems [34, 60], including newer versions of VeriPB [8, 29], have two additional features which are both vital for practical performance and efficiency, but which make a simple blocking-based approach unsound: strengthening and deletions.

2.1 Strengthening

Many applied solvers for combinatorial problems do not use purely "implicational" reasoning. Instead, in addition to deriving constraints that must be satisfied by any solution to the input formula, they also infer constraints that must be satisfied by *some* solutions, but not necessarily *all* solutions. As a simple example, with *pure literal elimination*, a SAT solver may observe that a literal only appears in positive form in all clauses [14]. If we are solving a decision problem, we can infer that such a literal can be set to true unconditionally, because if a solution exists to the problem then a solution necessarily exists where that literal is set to true. More generally, we might want to introduce symmetry-elimination constraints [1, 22, 51]: for example, if our entire problem is $X + Y = 1$ over integer variables X and Y with identical domains, then it is safe to consider only the case where $X \leq Y$. These inferences may exclude *some* solutions, and thus strengthen the constraints in the original formula, but will never turn a satisfiable instance into an unsatisfiable one.

For this reason, most modern proof logging formats weaken the invariant preserved by proof rules to being some notion like "without loss of satisfiability", rather than "implied". We will use the term "strengthening" to refer to proof rules that only give this weaker invariant. The exact form and power of these rules vary significantly between systems. However, the general intuition behind most of these rules is that they allow us to introduce a constraint C if we can show that given an assignment α which would satisfy the problem but does not satisfy C, we can somehow *patch* α to an assignment where we can easily verify that it would also satisfy C, whilst still satisfying the remaining constraints [10, 29, 38]. This patching rule is either given explicitly (such as by listing a set of substitutions), or is implicit in how the rule is defined, and must allow a proof checker to verify the patching conditions efficiently. For example, in our toy symmetry breaking example, we would show that $X \leq Y$ is safe by arguing that if we are given a solution where instead $X > Y$, we can patch this by swapping the values of X and Y. This notion also extends to optimisation problems, where we can reason "without loss of optimality" at each step and must verify that the patching procedure does not worsen the objective [8]. Any proof logging system for combinatorial solvers must support strengthening steps, since such reasoning is a crucial part of modern combinatorial solving. Clearly, however, strengthening of this kind cannot be used in enumeration proofs, because it allows a proof to exclude some solutions to the problem.

A further common use of strengthening is to define new variables in a proof. For example, suppose a proof author wants to introduce a *reification* constraint C saying $f \rightarrow (X + Y \geq 2)$, where the variable f is fresh. This is valid, because if we are given a solution to the original problem that violates C, we can "patch" this solution by setting f to false. Having done this, we could then also introduce another constraint C' saying $f \leftarrow (X + Y \geq 2)$, because we could patch a bad solution for C' by setting f to true (and note that this also satisfies C).

It turns out that the ability to introduce fresh variables can make proof systems exponen-

tially stronger theoretically [42, 56], and is vital in practice for efficient proofs of certain kinds of inference [15, 28]. However, fresh variables also cause problems for enumeration proofs. For example, introducing one direction of a reification might introduce new solutions, leading to overcounting (but it is hard to verify whether a given rule application *definitely* introduces new solutions or not). We might hope that, if we restrict ourselves to only introducing both directions of a reification simultaneously, then we would avoid this trap (at the expense of reasoning power): we discuss this further in what follows.

2.2 Deletions

A second vital feature of most proof systems is deletions [32]. Combinatorial solvers explore an exponentially large search space, but one important way of mastering this complexity is to avoid keeping too much information about the search in memory. For example, in a simple backtracking algorithm, once a solver has backtracked from a certain depth, it no longer needs to remember anything about the subtrees it explored below that depth. Similarly, solvers based around conflict-driven learning will periodically "forget" most of the clauses they have learned, in part to avoid memory exhaustion [43]. It is extremely helpful if a proof can issue the promise that "the remainder of the proof will not use the following constraints again, so if you are checking this proof you do not need to keep these constraints in memory".

Clearly, an unconditional deletion rule would allow non-solutions to be claimed as solutions, since a devious proof logger could simply delete all input constraints and claim that an arbitrary assignment is a "solution". Note, however, that this is not a problem if our proof system only allows for proofs of unsatisfiability, and has a separate external mechanism if we wish to certify satisfiability (which is the case for the most widely adopted SAT proof logging system DRAT [60]).

In a purely implicational proof system, deletions can be tamed by forbidding any constraint in the input from being deleted (although this is more restrictive than necessary). In order to ensure soundness of enumeration proofs, we could extend this to also forbid deleting solution-blocking constraints (so that previously discovered solutions cannot be reintroduced again), but this would come at the expense of requiring the proof checker to keep all solution-blocking constraints in memory rather than deleting them upon backtracking.

When combined with strengthening rules, deletions become even more complicated. For example, a problem may exhibit *conditional symmetries*, and we may wish to write a proof that first shows that the condition always holds, and then deletes parts of the input, allowing us to introduce a symmetry elimination constraint. The VeriPB system allows this when such reasoning can be shown to be sound, through a mechanism which we discuss in Section 3.

2.3 Enumeration Proofs with Strengthening and Deletions

Having now established why enumeration proofs are not straightforward, in the next section we will show the exact conditions needed to allow strengthening (and deletions) whilst preserving enumeration counts, or *equienumerability*, in the VeriPB proof format. It turns out that projection will be vital: the key observation is that, when combined with projection, the strengthening rule preserves equienumerability so long as the "patching" procedure does not touch any variable in the preserved set. Additionally, we show that VeriPB's existing support for deletions in optimisation problems can also be adapted to work for projected enumeration. Subsequently, we show how the proof system can allow the preserved set to be altered mid-proof, and that doing so provides interesting ways of generating proofs for solvers that count solutions in a way which is more efficient than simple enumeration.

We implement these features both in the user-friendly VeriPB checker and in the formally verified CakePB checker [4, 26, 36, 41]. As part of this, proofs of all three of the theorems that will follow have been formally verified inside a proof assistant.

3 Projected Enumeration Proofs in VeriPB

We will briefly review some basics about pseudo-Boolean (PB) reasoning and the VeriPB proof system, referring the reader to Bogaerts et al. [8] for more details on the VeriPB proof system and to Buss and Nordström [11] for a discussion of pseudo-Boolean reasoning. A *literal* ℓ of a *Boolean variable* x with domain 0 and 1 is either x itself or its negation $\overline{x} = 1 - x$. We often refer to 0 as *false* and 1 as *true*. A *pseudo-Boolean constraint* is a 0–1 integer linear inequality $\sum_i a_i \ell_i \geq A$ over literals ℓ_i, where in addition we can assume *normalised form* so that all integers a_i and A are positive and literals are over distinct variables. We note that for a PB constraint $C = \sum_i a_i \ell_i \geq A$ with $W = \sum_i a_i$ the negation $\neg C$ is another PB constraint $\sum_i a_i \ell_i \leq A - 1$ or $\sum_i a_i \overline{\ell}_i \geq W - A + 1$ in normalised form; the reification of C for a fresh variable f can be expressed with $f \rightarrow C \equiv A\overline{f} + \sum_i a_i \ell_i \geq A$ and $f \leftarrow C \equiv \overline{f} \rightarrow \neg C$.

Let F be a *pseudo-Boolean formula*, which is a conjunction $\bigwedge_i C_i$ of pseudo-Boolean constraints C_i. A *(partial) assignment* is a (partial) mapping σ from variables to $\{0, 1\}$, which is extended to literals by $\sigma(\overline{x}) = 1 - \sigma(x)$. A *solution* to F is an assignment σ such that each constraint in the formula is satisfied, i.e., $\sum_{\sigma(\ell_i)=1} a_i \geq A$ (again assuming normalised form). A pseudo-Boolean constraint C *propagates* a literal ℓ under a partial assignment σ if C is falsified by ℓ mapped to 0 and σ, i.e., $\sum_{\sigma(\ell_i)\neq 0 \wedge \ell_i \neq \ell} a_i < A$, where σ can be extended by mapping ℓ to 1. A constraint C is implied by *reverse unit propagation (RUP)* from a formula F if iteratively propagating constraints $F \wedge \neg C$ and updating an assignment σ results in a falsified constraint, i.e., such that $\sum_{\sigma(\ell_i)\neq 0} a_i < A$.

Given a *preserved set* of variables P, and denoting a restriction of the domain of a function by $|$, we say that the set $\{\sigma|_P : \sigma \text{ is a solution for } F\}$ is the *solution set of F projected onto P*. An *enumeration problem* is to explicitly list all the elements of this set exactly once each, whilst a *counting problem* is to determine the cardinality of this set. The *blocking constraint* for a given solution σ with respect to P is the constraint $\sum_{p\in P:\sigma(p)=0} p + \sum_{p\in P:\sigma(p)=1} \overline{p} \geq 1$, which says that at least one variable in P has to take a different value than in σ.

To support enumeration, we have extended VeriPB to allow its pseudo-Boolean input formula to specify a preserved set of variables; if projection is not desired, this set can be every variable appearing in the formula. This is similar to how optimisation is handled, where the input formula can come with an additional objective expression to minimise (which is $f = 0$ for decision problems).

▶ **Example 1.** Consider the following pseudo-Boolean problem instance, which we specify in the extended OPB format [48] supported by our modified version of VeriPB. To read this file, note that tilde denotes negation, and that each inequality is a weighted linear sum—here, each inequality is an "at least one" constraint, i.e., a disjunctive clause. Each inequality is also presented in standard math notation on the right hand side of each line.

`preserved: x1 x2 ;`	preserved set initialised to $\{x_1, x_2\}$
`1 x1 1 x2 1 x3 >= 1 ;`	$x_1 + x_2 + x_3 \geq 1$
`1 x1 1 x2 1 x4 >= 1 ;`	$x_1 + x_2 + x_4 \geq 1$
`1 ~x1 1 ~x2 >= 1 ;`	$\overline{x}_1 + \overline{x}_2 \geq 1$
`1 ~x1 1 ~x2 1 ~x3 >= 1 ;`	$\overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 1$
`1 x3 1 x4 >= 1 ;`	$x_3 + x_4 \geq 1$

A little thought shows that there are seven solutions to the problem without projection, but with projection onto x_1 and x_2 there are only three. *example continues below...*

We will now walk through a VeriPB enumeration proof. Most steps in a proof will introduce a new pseudo-Boolean constraint, based upon what is known so far. Initially the only things we know are the constraints in the input, which are taken as axioms.

▶ **Example 2** (continued). Continuing with our example above, we might want to start our proof by observing that x_4 is a pure literal, and so can always be set to true without changing the number of projected solutions. We use the *redundance-based strengthening rule* to do this, where the patching routine swaps x_4 with its negation. Recall that this means that "if you give me a solution which satisfies every constraint except $x_4 \geq 1$, then by negating x_4, i.e., by making it true, you can clearly see that we will now have a solution which satisfies every constraint, and also $x_4 \geq 1$". In the following VeriPB proof examples, the percent sign starts a comment.

```
pseudo-Boolean proof version 3.0
@purex4  red 1 x4 >= 1 : x4 -> ~x4 ; % x4 is pure
```

The second line starts with a `@label`, which gives a name to a constraint so we can refer to it later; constraints also automatically have numerical identifiers, but this is harder for a human to follow. *example continues below...*

For optimisation problems, the VeriPB proof system has the proof rule denoted `soli`, for "solution-improving", which takes a set of literals L as its argument. This rule says "check that L unit propagates to a solution α where every variable in the objective is assigned, and then introduce an objective-improving constraint $f \leq f(\alpha) - 1$ for this solution". For enumeration, we have introduced a similar rule `solx`, for "solution excluding", which checks that its argument literals unit propagate to a solution where every variable in the preserved set is assigned, and then introduces the blocking constraint for this solution with respect to the preserved set.

▶ **Example 3** (continued). Suppose now we find that $\{x_1, \overline{x}_2, \overline{x}_3\}$ is a solution. We can log this as follows.

```
@sol1    solx x1 ~x2 ~x3 ;          % log our first solution
```

Note here that we *do* need to specify an assignment for x_3, because it could be set either way, but x_4's value is obviously forced by the previous constraint. We also give a name to the blocking constraint that this proof step creates (which will be $\overline{x}_1 + x_2 \geq 1$).

Once we have this blocking constraint, it is clear that there are no more solutions where x_1 is true, so we can introduce the constraint $x_1 \leq 0$ (or $\overline{x}_1 \geq 1$ in normalised form) via reverse unit propagation. Having done so, we will never need the blocking constraint again, and so we can delete it; the `core` line in between will be explained below.

```
@x1false rup 1 ~x1 >= 1 ;          % no more solutions with x1 true
         core id @x1false ;        % ...and move this to core
         del id @sol1 ;            % no need to keep subtree constraints
```

Looking at what remains, x_3 is effectively a pure literal, because the fourth input constraint is the only place where it appears negatively, and this will always be satisfied by setting x_1 to false. After that, we can log the two solutions where x_1 is false, and use the two new blocking constraints to learn that x_1 must be true. We can then delete the blocking constraints.

```
@purex3  red 1 x3 >= 1 : x3 -> ~x3 ; % now x3 is effectively pure
@sol2    solx ~x1 x2 ;               % log our second solution
@sol3    solx ~x1 ~x2 ;              % and our third solution
@x1true  rup 1 x1 >= 1 ;             % no more solutions with x1 false
         core id @x1true ;           % ...and move this to core
         del id @sol2 @sol3 ;        % no need to keep subtree constraints
```

At this point, we have shown that x_1 must be both true and false, which is enough to establish a contradiction by reverse unit propagation. So, we can learn $0 \geq 1$, and then delete both constraints that talk about x_1, and for good measure we no longer care that x_3 and x_4 are pure either.

```
@contra  rup >= 1 ;                  % backtrack to root node
         core id @contra ;           % ...and move this to core
         del id @x1false @x1true ;   % no need to keep these constraints
         del id @purex3 @purex4 ;    % no need to remember purity
```

Now all that remains is to conclude that we have actually enumerated three solutions to completeness. *example continues below...*

A VeriPB proof ends with an output section and conclusion. For enumeration problems, we add support for three kinds of conclusions. The first is `ENUMERATION_COMPLETE` n `:` i, which states that we have witnessed and obtained blocking constraints for exactly n different projected solutions, and then proved unsatisfiability with constraint i. The second is `ENUMERATION_PARTIAL` n, which states that we have witnessed n different projected solutions, but that we do not claim to have found them all. The third is to extend the existing `NONE` conclusion with an `output EQUIENUMERABLE` option, which we discuss in the following section: this allows us to write proofs that alter a formula whilst preserving the number of solutions.

▶ **Example 4** (continued). We can finish our proof with the following:

```
output NONE ;
conclusion ENUMERATION_COMPLETE 3 : @contra ;
end pseudo-Boolean proof ;
```

This is now a complete proof, which is accepted by our enhanced VeriPB and CakePB checkers. ◀

With this example completed, we will now go into more detail on why our modified proof system remains sound. We will only sketch proofs of the formal claims that follow—one reason why this is in order is that formally machine-verified proofs of all of these claims are part of the CakePB backend. Firstly, we observe that our new `solx` rule allows us to count solutions if we are only using the implicational parts of the proof system.

▶ **Proposition 5.** *If a proof uses only implicational rules and* `solx`*, then each* `solx` *rule invocation excludes exactly one new projected solution.*

Proof sketch. Since the proof only uses implicational rules, the set of solutions does not change. By definition, the `solx` rule checks that the claimed solution is in fact valid. Then, we cannot log the same solution (under projection) twice, because the second time we attempt to do so, our witness will violate the blocking constraint introduced by the first attempt. ◀

We now need to establish that none of the remaining proof rules allow us to alter the number of solutions. The two non-implicational rules in the VeriPB proof system are strengthening and deletions.

▶ **Proposition 6.** *Applications of strengthening where the patching procedure does not touch the preserved set always preserve the projected solution set.*

Proof sketch. It is most illuminating to look at the two ways we have seen strengthening be used, and to see why they do not break this property.

For strengthening rules that introduce an implication under a fresh variable, the patching procedure only touches the fresh variable, which is not inside the preserved set. This new constraint does not exclude solutions, since the fresh variable can be assigned to satisfy the constraint. This rule may introduce additional solutions, but the fresh variable is not included in the preserved set, so these additional solutions will be equivalent under projection.

For strengthening rules that touch existing variables which are not in the preserved set, this may exclude (unprojected) solutions. However, the patching procedure leaves the value of the variables in the preserved set unchanged, so that the patched solution restricted to the preserved set is the same.

Note that neither of these arguments depend upon the detail of exactly how we justify the patching routine, and so they are valid for both redundance-based strengthening and the more powerful *dominance-based strengthening rule* [8]. Observe also that we are able to introduce both one-way and two-way reifications on fresh variables this way, but as a special case of the more general result, rather than needing an explicitly restricted proof rule. ◀

To understand deletions, we need to explain a little more about VeriPB's *checked deletion* mode. Within a proof, every constraint is considered to be either *core* or *derived*. Initially, everything in the input goes into the core set, whilst constraints inside the proof go into either the core set or the derived set depending upon how they are created. Additionally, any derived constraint can be moved into core by an explicit proof step. Constraints can always be deleted from the derived set, but to delete a constraint C from the core set, we must show that C can be rederived using redundance-based strengthening by just using other core constraints. Critically, once a constraint is in the core set, either it or stronger constraints remain there for the remainder of the proof. This system was developed to allow for optimisation proofs that use symmetry or dominance reasoning, and we remind readers that the full details are described in Bogaerts et al. [8].

▶ **Proposition 7.** *If all deletions are checked, then deletion will not allow a proof to claim non-solutions with respect to the preserved variables as a solution.*

Proof sketch. Since we show that the deleted constraint can be rederived by redundance strengthening from the other constraints in the core set, the core set with and without the deleted constraint have the same projected solution set by Proposition 6. ◀

The VeriPB proof system handles the solution-improving constraints for optimisation problems by putting them into the core set. We do the same for enumeration.

▶ **Proposition 8.** *If all deletions are checked, and if blocking constraints are created in the core set, then deletions will not allow projected solutions to be counted more than once.*

Proof sketch. If we are able to delete a blocking constraint from the core, then it must be possible to rederive it from the other core constraints, since all deletions are checked. So the blocking constraint always remains a consequence of the core set by Proposition 7. ◀

At this point, we wish to claim the following.

▶ **Theorem 9.** *If a VeriPB proof uses only checked deletions, does not strengthen using any patching procedure which touches a preserved variable, has blocking constraints created in the core set, and is able to derive a contradiction constraint, then the number of projected solutions to the original problem is precisely the number of times the* `solx` *rule is used.*

To do this, we are effectively claiming "there are no other elements of the VeriPB proof system that we have forgotten about that could break things". The most convincing demonstration we have that this is true is that we have formally verified the entire proof system inside CakePB, including the conclusions supported.

3.1 Formal Verification of Enumeration Proofs

CakePB is a formally verified proof checker that can handle a subset of the VeriPB proof rules. To support an end-to-end certified solving workflow, VeriPB can operate in *elaboration* mode, where it takes a proof written in a user-friendly *augmented* proof format and elaborates it to a simpler *kernel* format proof for CakePB to check. Intuitively, elaboration adds proof details which are easy to fill with an unverified search, while the kernel format contains a set of primitive pseudo-Boolean proof rules that can be efficiently implemented and formally verified in a proof assistant with reasonable effort.

We have extended the formal verification of CakePB (in the HOL4 proof assistant [52]) to support all of the new rules described in this paper. Let us recall CakePB's refinement-based verification methodology [26], and describe our extensions at each step.

1. The formalisation starts by defining the pseudo-Boolean problem semantics, and then defines the abstract PB proof system. We extended the semantics to support projected enumeration, namely:

$$\begin{aligned}&\mathsf{sem\_concl}\ \mathit{pbf}\ \mathit{obj}\ \mathit{pres}\ (\mathsf{EEnum}\ n\ \mathit{complete}) \stackrel{\mathsf{def}}{=}\\ &\quad n \leq \mathsf{card}\ (\mathsf{proj\_pres}\ \mathit{pres}\ \{\ w \mid \mathsf{satisfies}\ w\ \mathit{pbf}\ \}\ ) \wedge\\ &\quad (\mathit{complete} \Rightarrow \mathsf{card}\ (\mathsf{proj\_pres}\ \mathit{pres}\ \{\ w \mid \mathsf{satisfies}\ w\ \mathit{pbf}\ \}\ ) \leq n)\end{aligned}$$

   Here, we are defining the semantics of conclusions (sem_concl) that can be made about a PB problem with constraints *pbf*, objective *obj*, and preserved set *pres*. For the new conclusion type EEnum we allow a proof to claim either partial or complete enumeration of n projected solutions. The proj_pres function projects the set of satisfying solutions for *pbf*, $\{ w \mid \mathsf{satisfies}\ w\ \mathit{pbf} \}$, onto *pres*; card is the cardinality of a set in HOL.
2. The next step is to implement the kernel proof rules with an abstract proof checking algorithm. In this phase, we introduce the `solx` command (among others), and we also modify existing rules to be compatible with the preserved set. In essence, we have a machine-checked proof of Theorem 9 (and its constituent propositions) showing that whenever the checker successfully processes a proof with a formal conclusion (like EEnum), then that conclusion is indeed true of the input PB problem.
3. In the third phase, the abstract proof checker is refined into an optimised machine-code implementation using CakeML [31, 47, 53]. Thanks to the relatively small-scale change to the proof system, the updates here were straightforward. We ultimately obtain an end-to-end proof that the machine-code semantics (either x64 or ARMv8) is sound for our updated PB problem semantics.

The steps above result in what we refer to as the *backend* CakePB implementation for pseudo-Boolean problems. As discussed earlier, pseudo-Boolean proofs can also be used more broadly to certify solvers for more general combinatorial and automated reasoning paradigms.

For this, we need an additional step to set up a formally verified *frontend* which ensures that the original (non-pseudo-Boolean) problem is re-encoded into pseudo-Boolean format correctly. This yields a certification flow where a general combinatorial solver solves problems with its native constraint reasoning, while emitting pseudo-Boolean justifications that are checked by VeriPB and CakePB to establish valid claims about the input problem.

4. (Optional step) For a given combinatorial problem, e.g., maximal clique enumeration [55], we start by formalising the problem semantics as follows.

$$\begin{aligned}
\mathsf{is_clique}\ vs\ (v,e) &\stackrel{\mathsf{def}}{=} vs \subseteq \mathsf{count}\ v \wedge \forall x\ y.\ x \in vs \wedge y \in vs \wedge x \neq y \Rightarrow \mathsf{is_edge}\ e\ x\ y \\
\mathsf{is_maximal_clique}\ vs\ g &\stackrel{\mathsf{def}}{=} \mathsf{is_clique}\ vs\ g \wedge \forall vs'.\ \mathsf{is_clique}\ vs'\ g \wedge vs \subseteq vs' \Rightarrow vs = vs' \\
\mathsf{maximal_cliques}\ g &\stackrel{\mathsf{def}}{=} \{\ vs \mid \mathsf{is_maximal_clique}\ vs\ g\ \}
\end{aligned}$$

Here, a graph g is represented as a pair consisting of a natural number v for vertices labeled $\mathsf{count}\ v \stackrel{\mathsf{def}}{=} \{0,\ldots,v-1\}$ and edges e (looked up by the is_edge function). As usual, the set vs forms a clique if it is a set of vertices in g that mutually share an edge. A clique vs satisfies is_maximal_clique iff there is no larger clique containing vs.
Then, we prove the soundness for an encoder into PB:

$$\begin{aligned}
&\vdash \mathsf{good_graph}\ g \wedge \mathsf{enc}\ g = (pres,pbf) \wedge \mathsf{sem_concl}\ pbf\ \mathsf{None}\ pres\ (\mathsf{EEnum}\ n\ b) \Rightarrow \\
&\quad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{card}\ (\mathsf{maximal_cliques}\ g) = n\ \mathsf{else}\ n \leq \mathsf{card}\ (\mathsf{maximal_cliques}\ g)
\end{aligned}$$

This theorem says that running the problem encoder enc on the input graph g returns a preserved set $pres$ and a set of PB constraints pbf, such that whenever sem_concl holds with an enumeration conclusion for this PB problem, that conclusion correctly reports the number of enumerated maximal cliques. (The None argument to sem_concl indicates no objective function.) Here, enc uses the direct encoding for cliques, where a Boolean variable x_i is true iff vertex i is in the chosen clique. Accordingly, the preserved set $pres$ is automatically inferred to be the set of vertex variables for g. Putting this verified encoder frontend together with the verified backend PB proof checker CakePB yields an end-to-end verified proof checker for maximal clique enumeration, which we call CakePBmclique.

3.2 Experiments for Enumeration Proofs

We carried out two sets of experiments to validate our implementation. Firstly, we adapted the maximal clique enumeration implementation of Tomita et al.'s algorithm [55] by Gocht et al. [25] to use our new `solx` command, projection, and conclusion, and added support for deletions (which would not have been sound for enumeration problems in the very first version of VeriPB). We then successfully replicated all of Gocht et al.'s clique enumeration experiments using both VeriPB and CakePBmclique, except for one instance where CakePBmclique exceeded a 48GByte RAM limit when verifying the encoding. Note that CakePB manages its own memory in a fully verified manner and exits gracefully *without* making any claims about the input problem when an out-of-memory error occurs. Thus, the theoretical soundness guarantee still holds (trivially).

Secondly, we adapted the Glasgow Constraint Solver's test suite [28]. Part of the testing process for this solver is to create a large number of random constraint satisfaction problems, verify that the solver generates the same set of solutions as an extremely simple generate-and-test routine, and then to use VeriPB to check an enumeration proof for the same problem. Previously, these proofs relied upon underspecified solution-excluding behaviour as

a workaround, and could not formally claim a complete enumeration as a conclusion or make use of CakePB. Our extensions to VeriPB rectify this, and additionally allow us to pass the proofs through CakePB for formal verification. Additionally, we altered the solver to use projection to explicitly exclude auxiliary variables in the encoding from the solution count, meaning that its tests now guarantee that solution counts are in terms of the high-level constraint programming variables, not the low-level encoding. With these additions, all of the solver's tests still pass the stricter verification setup, and we have formally verified the results of tens of thousands of enumeration proofs for constraint satisfaction problems.

4 Proofs for Preprocessing and Counting Without Enumerating

Instead of checking a complete solving process, VeriPB (and CakePB) can also be used to check that a preprocessor preserves certain guarantees [36]. In order to make claims about the properties of a rewritten formula produced by a preprocessor or presolver, proofs use an *output section* with a heading such as `output EQUISATISFIABLE FILE`, followed by an empty *conclusion section* with the heading `conclusion NONE`. This particular output section heading tells VeriPB to check that the proof demonstrates that a pseudo-Boolean problem instance provided in a file named on the command line is *equisatisfiable* with respect to the original pseudo-Boolean problem instance (i.e., the output formula is unsatisfiable if and only if the input formula is unsatisfiable).

We have extended VeriPB and its proof system to also support `output EQUIENUMERABLE` as an output type. This feature can be used to check that two pseudo-Boolean problem instances have the same number of (projected) solutions, without actually listing those solutions. We do *not* require that these two pseudo-Boolean problem instances have the same preserved set, and instead allow variables to be added to and removed from the preserved set during the proof, so long as a justification is given—we explain how this is possible below.

Beyond preprocessing, this feature has a second potential use. In some situations, a user may only want to know how many solutions there are to a problem instance, without requiring an explicit list. In such cases, sometimes solvers can produce results more efficiently than carrying out an enumeration. One family of techniques to do this involves recompiling the problem into a new structure which is efficiently countable, such as some kind of decision diagram [13, 61]. Another is to decompose the problem, enumerate the solutions to each decomposition, and then use an inclusion-exclusion argument to generate the count [39, 49, 50].

We remark that specialised certification methods are available for both exact [19] and approximate [54] model counting. Of these, the CPOG framework [9] similarly enables a proof of equivalence between an input CNF and the output of a knowledge compiler. Our VeriPB-based approach could offer a generic means of certifying transformation-based tools.

▶ **Example 10.** Suppose someone wishes to convince you that there are exactly three (projected) solutions to our previous example. It should be sufficient for them to provide a VeriPB proof that ends

```
output EQUIENUMERABLE FILE ;
conclusion NONE ;
end pseudo-Boolean proof ;
```

where the output file provided could be

```
preserved: s1 s2 s3 ;
1 s1 1 s2 1 s3 = 1 ;                    exactly one s variable is true
```

Why is this? Recall that our original problem is entirely over x_i variables. Here, our output file is now over a fresh set of s_i variables, where we have a constraint saying that exactly one of these three variables is true. We do not have any other constraints and so clearly there are three solutions to this problem. *example continues below...*

This example on its own is probably too simple to convincingly illustrate the benefits of an alternative approach. However, suppose our output constraints only said that $\sum_{i=1}^{10} s_i = 1$, $\sum_{i=1}^{5} t_i = 1$, $\sum_{i=1}^{8} u_i = 1$, and $s_1 \rightarrow \overline{t}_1$. From this we could easily and efficiently determine that there are $(10 \times 5 \times 8) - (1 \times 8) = 392$ solutions using an inclusion-exclusion argument, with a proof that could potentially be of the order $10+5+8+1$ steps long. Such an approach mirrors how some subgraph-counting algorithms work [49]. We will now outline how such proofs could be produced (for the simpler example, although this technique generalises cleanly), together with the additional features we have implemented to make this possible.

4.1 Tabulating Solutions as Extension Variables

Rather than logging solutions using `solx`, our proofs will introduce an extension variable s_1 for the first solution found, reifying the negation of the blocking constraint that `solx` would introduce (resembling a proof for autotabulation [28]). We will then move the forward direction of this implication to the core.

▶ **Example 11** (continued). We might start a proof as follows. The first `red` line uses strengthening for a pure literal, as before, and the second and third `red` lines introduce the reification variable.

```
pseudo-Boolean proof version 3.0
@purex4  red 1 x4 >= 1 : x4 -> ~x4 ;              % x4 is pure
@sol1f   red 2 ~s1 1 x1 1 ~x2 >= 2 : s1 -> 0 ;    % log our first solution
@sol1b   red 1 s1  1 ~x1 1 x2 >= 1 : s1 -> 1 ;    % ...in both directions
         core id @sol1f ;                            example continues below...
```

Going forward, for every subsequent constraint we generate, we include $\overline{s}_1$ as an additional assumption, until we hit a second solution. We then introduce a second extension variable s_2 for this solution, and change our assumption to be $\overline{s}_1 \wedge \overline{s}_2$, and so on.

▶ **Example 12** (continued). So, our proof can continue like this.

```
@x1false rup 1 s1 1 ~x1 >= 1 ;                    % unless s1, x1 is false
@purex3  red 1 s1 1 x3 >= 1 : x3 -> ~x3 ;         % unless s1, x3 is pure
@sol2f   red 2 ~s2 1 ~x1 1 x2 >= 2 : s2 -> 0 ;    % log our second solution
@sol2b   red 1 s2  1 x1 1 ~x2 >= 1 : s2 -> 1 ;    % ...in both directions
         core id @sol2f ;
@sol3f   red 2 ~s3 1 ~x1 1 ~x2 >= 2 : s3 -> 0 ;   % log our third solution
@sol3b   red 1 s3  1 x1 1 x2 >= 1 : s3 -> 1 ;     % ...in both directions
         core id @sol3f ;
@x1true  rup 1 s1 1 s2 1 s3 1 x1 >= 1 ;           % otherwise, x1 is true
@al1sol  rup 1 s1 1 s2 1 s3 >= 1 ;                % no other solutions
         core id @al1sol ;
         del id @x1true @x1false ;                   example continues below...
```

We then continue our solving process until completion, at which point, instead of deriving contradiction, we have derived an at-least-one constraint over a set of s_i variables exactly corresponding to the number of solutions found. We move this to the core, and can now delete any remaining constraints from the search process.

Next, we want to turn this at-least-one constraint into an exactly-one constraint. This can be done efficiently using a standard technique in VeriPB proofs, by first deriving a not-both constraint for each pair of s_i variables: we refer to Gocht et al. [25] for an explanation. We also move this constraint to core.

▶ **Example 13** (continued). In this case, we can generate the at-most-one constraint using the following sequence of steps, where the reverse unit propagation succeeds because the s_i variables represent different projected solutions.

```
@s1ors2  rup 1 ~s1 1 ~s2 >= 1 ;                % s1 and s2 are exclusive
@s1ors3  rup 1 ~s1 1 ~s3 >= 1 ;                % s1 and s3 are exclusive
@s2ors3  rup 1 ~s2 1 ~s3 >= 1 ;                % s2 and s3 are exclusive
@am1sol  pol -3 -2 + -1 + 2 d ;                % recover the at-most-one
         del id @s1ors2 @s1ors3 @s2ors3 ;      % intermediate steps
         core id @am1sol ;                        example continues below...
```

At this point, our core set should consist precisely of all the original constraints, all the forward implications for our s_i variables, and the exactly-one constraint over the s_i variables. Our preserved set remains unchanged, which is what we will address next.

4.2 Extending the Preserved Set

We would like to be able to add our s_i variables to the preserved set. By careful thought, in this particular case, we can see that doing so would not change the number of solutions. To convince the proof checker of this, we need an argument such as the following.

▶ **Theorem 14.** *Given a pseudo-Boolean formula F with preserved set P, a variable v, and some constraint C only over variables in P, if at some point during a VeriPB proof it is possible to derive the constraints $v \rightarrow C$ and $v \leftarrow C$ using only core constraints, then adding v to P for the remainder of the proof will not affect the projected enumeration count.*

Proof sketch. Any solution to F projected on to P will either satisfy or falsify C, since C only contains variables in P, and thus v's value will be uniquely determined by this solution. Furthermore, because both directions of the reification are derivable from the core set, this property cannot be altered by any later step in the proof. ◀

We have therefore extended VeriPB with the proof rule `preserved_add` that takes a variable s and an associated constraint C over preserved variables, for which it can be proven that $s \leftrightarrow C$ holds. Just as for other advanced VeriPB rules, there is the possibility to provide explicit subproofs that show why this claim holds. However, our example here is simple enough that VeriPB will be able to figure out all required details with so-called *auto-proving* without any help from the proof logger, and so we instead refer the interested reader to the VeriPB technical documentation [3] for more details on how subproofs work.

▶ **Example 15** (continued). Each of our s_i variables is in fact defined by an if-and-only-if reification, so we can use this as the constraint.

```
preserved_add s1 1 x1 1 ~x2 >= 2 ;      % add s1 to preserved set
preserved_add s2 1 ~x1 1 x2 >= 2 ;      % add s2 to preserved set
preserved_add s3 1 ~x1 1 ~x2 >= 2 ;     % add s3 to preserved set
```

example continues below...

4.3 Shrinking the Preserved Set

Our next challenge is to remove the x_i variables from the preserved set. Again, we need a general justification of why it is safe to do this.

▶ **Theorem 16.** *Given a pseudo-Boolean formula F with preserved set P, a preserved variable $v \in P$, and some constraint C only over the variables in $P \setminus \{v\}$, if at some point during a VeriPB proof it is possible to derive the constraints $v \rightarrow C$ and $v \leftarrow C$ using only core constraints, then removing v from P for the remainder of the proof will not affect the projected enumeration count.*

Proof sketch. This holds since any solution to F projected on to P will either satisfy or falsify C, which will force v to 0 or 1. So all these conditions together mean that the value of v will always be uniquely determined by any assignment to the remaining variables in P. ◀

We have implemented this through the `preserved_rm` rule, which takes a variable and a constraint given explicitly. (Interestingly, this operation is reversible: a proof checker could in principle go over the proof backwards to efficiently recover solutions in terms of the original variables, without the need for a separate solution reconstruction stack.)

▶ **Example 17** (continued)**.** We can obtain the desired constraint by looking at precisely which s_i variables assign a variable to true.

```
preserved_rm x1 1 s1 >= 1 ;         % remove x1 from preserved set
preserved_rm x2 1 s2 >= 1 ;         % remove x2 from preserved set
```

example continues below...

4.4 Deleting Remaining Constraints

At this point, we have a convincing proof that there are *at most* three solutions, but we still retain all the original constraints over the x_i variables in the core set, so it is not clear that each s_i assignment would actually give a valid solution. (Nor is this fact trivially checkable, since the s_i variables do not force values for x_3 or x_4.) The final step, then is to delete each original constraint in turn.

To delete a constraint from core, we must show that the constraint can be reconstructed using only the other remaining core constraints. We should always be able to do this, using a fairly simple procedure. Suppose first that we were not producing a projection proof, and that our original preserved set contained every x_i variable. Consider any original constraint C. We will use a proof by contradiction, showing that the negation of C together with the reification constraints for each s_i variable implies unsatisfiability, as follows. For each s_i variable in turn, if that variable is true, then it implies an assignment of x_i variables that satisfy every original constraint, including C, which means that the negation of C is falsified. Then, we resolve over the at-least-one s_i constraint to reach our overall contradiction.

When we do have a preserved set, this procedure is potentially slightly more complicated. In our example, the constraint $x_3 + x_4 \geq 1$ is not contradicted by assigning true to any s_i variable, and we relied upon (conditional) pure literal reasoning to handle these variables.

This is not an insurmountable problem: we can solve this by temporarily moving the pure literal constraints into core, and then deleting them at the end of the proof using a substitution justification (we refer to the VeriPB documentation [3] for details, since this feature is the same as for optimisation proofs).

▶ **Example 18** (continued). In our example, the proof by contradiction is sufficiently simple in each case that it can be autoproved, so we only need the following.

```
core id @purex3 @purex4 ;
del id 1 2 3 4 5 ;
del id @purex3 : x3 -> ~x3 ;
del id @purex4 : x4 -> ~x4 ;
```

Finally, we can delete the forward implications of the s_i-variables using their corresponding solution for the patching procedure.

```
del id @sol1f : x1 -> 1 x2 -> 0 ;
del id @sol2f : x1 -> 0 x2 -> 1 ;
del id @sol3f : x1 -> 0 x2 -> 0 ;
```

At this point, we can terminate the proof using the desired output statement. ◀

This approach generalises to outputs over multiple sets of variables, potentially with further restrictions between them. With these new proof rules in place, we have enough reasoning power to handle certain inclusion-exclusion arguments. If we wished to develop a complete and general end-to-end verification system, we would also need to agree upon an efficiently-countable but expressive subset of pseudo-Boolean formulas, and to implement a formally-verified frontend for this subset. Choosing an appropriate and useful language would require a detailed study of commonly used decomposition and counting techniques.

5 Conclusion

Enhancing combinatorial solvers with *proof logging* to make them *certifying* is currently the most successful way of ensuring that such solvers produce trustworthy results. Proof logging technology is by now mature for decision problems in Boolean satisfiability (SAT) solving, and is also starting to be developed for more and more optimisation and automated reasoning paradigms. However, the most widely adopted proof logging systems are inherently unable to support proof logging for problems such as enumerating or counting the number of solutions. Instead, proof logging support for such reasoning problems has so far only been available in bespoke tools with limited applicability.

In this work, we demonstrated how the VeriPB proof logging system, which has previously been shown to provide a unified proof logging format for a wide range of combinatorial solving and optimisation problems, can also be extended to support also solution enumeration proofs in the same proof format. In contrast to previous approaches, for VeriPB this is possible without sacrificing any of the advanced features that make the proof system so powerful, such as strengthening rules and checked deletion. This opens up the future possibility of also providing efficient proofs for counting problems (where it is crucial to not have to enumerate all solutions), by supporting proofs of solution-count-preserving transformations for the input formula to a format in which counting solutions is trivial.

References

1 Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization*, pages 449–481. Springer, 2013.

2 Özgür Akgün, Martín Mereb, and Leandro Vendramin. Enumeration of set-theoretic solutions to the yang-baxter equation. *Math. Comput.*, 91(335):1469–1481, 2022. URL: `https://doi.org/10.1090/mcom/3696`, `doi:10.1090/MCOM/3696`.

3 Markus Anders, Bart Bogaerts, Benjamin Bogø, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Adrián Rebola-Pardo, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT competition 2026. Available at `https://satcompetition.github.io/2026/output.html`, March 2026.

4 Markus Anders, Bart Bogaerts, Benjamin Bogø, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Adrián Rebola-Pardo, and Yong Kiam Tan. Faster certified symmetry breaking using orders with auxiliary variables. In *Proceedings of the 40th AAAI Conference on Artificial Intelligence (AAAI '26)*, pages 14140–14148, January 2026.

5 Haniel Barbosa, Clark Barrett, Byron Cook, Bruno Dutertre, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Cesare Tinelli, and Yoni Zohar. Generating and exploiting automated reasoning proof certificates. *Communications of the ACM*, 66(10):86—95, October 2023.

6 Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, July 2023.

7 Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition, February 2021.

8 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. Preliminary version in *AAAI '22*.

9 Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to verified model counting. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT '23)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:20, July 2023.

10 Sam Buss and Neil Thapen. DRAT and propagation redundancy proofs without new variables. *Logical Methods in Computer Science*, 17(2):12:1–12:31, April 2021. Preliminary version in *SAT '19*.

11 Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [7], chapter 7, pages 233–350.

12 Michael Codish, Alice Miller, Patrick Prosser, and Peter J. Stuckey. Constraints for symmetry breaking in graph representation. *Constraints An Int. J.*, 24(1):1–24, 2019. URL: `https://doi.org/10.1007/s10601-018-9294-5`, `doi:10.1007/S10601-018-9294-5`.

13 Adnan Darwiche. New advances in compiling CNF to decomposable negation normal form. In *Proc. of ECAI*, pages 328–332, 2004.

14 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

15 Emir Demirović, Ciaran McCreesh, Matthew McIlree, Jakob Nordström, Andy Oertel, and Konstantin Sidorov. Pseudo-Boolean reasoning about states and transitions to certify dynamic programming and decision diagram algorithms. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, September 2024.

16 Andreas Distler, Christopher Jefferson, Tom W. Kelsey, and Lars Kotthoff. The semigroups of order 10. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 883–899. Springer, 2012. doi:10.1007/978-3-642-33558-7_63.

17 Simon Dold, Malte Helmert, Jakob Nordström, Gabriele Röger, and Tanja Schindler. Pseudo-Boolean proof logging for optimal classical planning. In *Proceedings of the 35th International Conference on Automated Planning and Scheduling (ICAPS '25)*, pages 54–63, November 2025.

18 Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.

19 Johannes Klaus Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, Haifa, Israel, August 2-5, 2022*, volume 236 of *LIPIcs*, pages 30:1–30:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.SAT.2022.30.

20 Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirovic. A multi-stage proof logging framework to certify the correctness of CP solvers. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, Girona, Spain, September 2-6, 2024*, volume 307 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.11.

21 Graeme Gange, Geoffrey Chu, and Peter J. Stuckey. Certifying optimality in constraint programming. Unpublished manuscript, 2017. URL: https://people.eng.unimelb.edu.au/pstuckey/papers/certified-cp.pdf.

22 Maria Garcia de la Banda, Peter J. Stuckey, Pascal Van Hentenryck, and Mark Wallace. The future of optimization technology. *Constraints*, 19(2):126–138, April 2014.

23 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Solution enumeration for projected boolean search problems. In Willem-Jan van Hoeve and John N. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 71–86, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

24 Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Certified CNF translations for pseudo-Boolean solving. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.

25 Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

26 Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, February 2024.

27 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

28 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022.

29 Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

30 Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pages 886–891, March 2003.

31 Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *Proceedings of the 26th European Symposium on Programming (ESOP '17)*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, April 2017.

32 Marijn J. H. Heule, Warren A. Hunt, and Nathan Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Softw. Test. Verif. Reliab.*, 24(8):593–607, September 2014.

33 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, August 2017.

34 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Encoding redundancy for satisfaction-driven clause learning. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '19)*, volume 11427 of *Lecture Notes in Computer Science*, pages 41–58. Springer, April 2019.

35 Ruth Hoffmann, Özgür Akgün, and Christopher Jefferson. Composable constraint models for permutation enumeration. *Discrete Mathematics & Theoretical Computer Science*, vol. 26:1, Permutation Patterns 2023, Jan 2025. doi:10.46298/dmtcs.12620.

36 Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. Certified MaxSAT preprocessing. In *Proceedings of the 12th International Joint Conference on Automated Reasoning (IJCAR '24)*, volume 14739 of *Lecture Notes in Computer Science*, pages 396–418. Springer, July 2024.

37 Avraham Itzhakov and Michael Codish. Incremental symmetry breaking constraints for graph search problems. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1536–1543. AAAI Press, 2020. URL: https://doi.org/10.1609/aaai.v34i02.5513, doi:10.1609/AAAI.V34I02.5513.

38 Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR '12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, June 2012.

39 Richard M. Karp. Dynamic programming meets the principle of inclusion and exclusion. *Operations Research Letters*, 1(2):49–51, 1982. doi:10.1016/0167-6377(82)90044-X.

40 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation and enumeration. *ACM Trans. Comput. Log.*, 25(3):1–30, 2024. doi:10.1145/3670405.

41 Wietze Koops, Daniel Le Berre, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Yong Kiam Tan, and Marc Vinyals. Practically feasible proof logging for pseudo-Boolean optimization. In *Proceedings of the 31st International Conference on Principles and Practice of Constraint Programming (CP '25)*, volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:27, August 2025.

42 Jan Krajíček. *Proof Complexity*, volume 170 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, March 2019.

43 Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Biere et al. [7], chapter 13, pages 509–570.

44 Matthew McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP '23)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, August 2023.

45 Matthew McIlree and Ciaran McCreesh. Certifying bounds propagation for integer multiplication constraints. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI '25)*, pages 11309–11317, February-March 2025.

46 Matthew McIlree, Ciaran McCreesh, and Jakob Nordström. Proof logging for the circuit constraint. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14743 of *Lecture Notes in Computer Science*, pages 38–55. Springer, May 2024.

47 Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2–3):284–315, January 2014.

48 Olivier Roussel and Vasco M. Manquinho. Input/output format and solver requirements for the competitions of pseudo-Boolean solvers. Revision 2324. Available at `http://www.cril.univ-artois.fr/PB16/format.pdf`, January 2016.

49 Jessica Laurette Ryan. *Parameterised algorithms for counting subgraphs, matchings, and monochromatic partitions*. PhD thesis, University of Glasgow, 2023. URL: `https://theses.gla.ac.uk/83568/3/2023RyanPhD.pdf`.

50 Herbert John Ryser. *Combinatorial mathematics*, volume 14. American Mathematical Soc., 1963.

51 Karem A. Sakallah. Symmetry and satisfiability. In Biere et al. [7], chapter 13, pages 509–570.

52 Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, pages 28–32, 2008. `doi:10.1007/978-3-540-71067-7\_6`.

53 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2:1–e2:57, February 2019.

54 Yong Kiam Tan, Jiong Yang, Mate Soos, Magnus O. Myreen, and Kuldeep S. Meel. Formally certified approximate model counting. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 153–177. Springer, 2024. `doi:10.1007/978-3-031-65627-9\_8`.

55 Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006. URL: `https://doi.org/10.1016/j.tcs.2006.06.015`, `doi:10.1016/J.TCS.2006.06.015`.

56 Grigori Tseitin. On the complexity of derivation in propositional calculus. In A. O. Silenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. Consultants Bureau, New York-London, 1968.

57 Daimy Van Caudenberg, Bart Bogaerts, and Leandro Vendramin. Incremental sat-based enumeration of solutions to the yang-baxter equation. In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II*, volume 15697 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2025. `doi:10.1007/978-3-031-90653-4\_1`.

58 Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.

59 Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '10)*, pages 204–209, July 2010.

60 Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International*

Conference on Theory and Applications of Satisfiability Testing (SAT '14), volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

61 Suwei Yang and Kuldeep S. Meel. Towards projected and incremental pseudo-boolean model counting. In Toby Walsh, Julie Shah, and Zico Kolter, editors, *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA*, pages 11399–11407. AAAI Press, 2025. URL: https://doi.org/10.1609/aaai.v39i11.33240, doi:10.1609/AAAI.V39I11.33240.