Practically Feasible Proof Logging for Pseudo-Boolean Optimization

Wietze Koops
Lund University, Sweden
University of Copenhagen, Denmark

Daniel Le Berre
Univ. Artois, CNRS, UMR 8188 CRIL, France

Magnus O. Myreen
Chalmers University of Technology, Sweden
University of Gothenburg, Sweden

Jakob Nordström
University of Copenhagen, Denmark
Lund University, Sweden

Andy Oertel
Lund University, Sweden
University of Copenhagen, Denmark

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

Marc Vinyals
University of Auckland, New Zealand

Abstract

Certifying solvers have long been standard for decision problems in Boolean satisfiability (SAT), allowing for proof logging and checking with very limited overhead, but developing similar tools for combinatorial optimization has remained a challenge. A recent promising approach covering a wide range of solving paradigms is pseudo-Boolean proof logging, but this has mostly consisted of proof-of-concept works far from delivering the performance required for real-world deployment.

In this work, we present an efficient toolchain based on *VeriPB* and *CakePB* for formally verified pseudo-Boolean optimization. We implement proof logging for the full range of techniques in the state-of-the-art solvers *RoundingSat* and *Sat4j*, including core-guided search and linear programming integration with Farkas certificates and cut generation. Our experimental evaluation shows that proof logging and checking performance in this much more expressive paradigm is now quite close to the level of SAT solving, and hence is clearly practically feasible.

2012 ACM Subject Classification Theory of computation → Logic and verification; Mathematics of computing → Combinatorial optimization

Keywords and phrases proof logging, certifying algorithms, combinatorial optimization, certification, pseudo-Boolean solving, 0–1 integer linear programming

Digital Object Identifier 10.4230/LIPIcs.CP.2025.42

Supplementary Material *Software (RoundingSat)*: `https://gitlab.com/MIAOresearch/software/roundingsat`
archived at `swh:1:rev:7116cbc7c228335d7a828fc9086639e4321fd5d1`
Software (Sat4j): `http://www.sat4j.org/`
archived at `swh:1:rev:adda8a7c5414b89b6f8ca1b5452c17526296104f`
Software (VeriPB): `https://gitlab.com/MIAOresearch/software/VeriPB`
archived at `swh:1:rev:9dbb658ffb8d88815c3f3d8ff7fcefd6c6fc43db`

Software (CakePB): `https://gitlab.com/MIAOresearch/software/cakepb`
archived at `swh:1:rev:b29b4e967413ee4e8c3dcb1bbc78bd0d508d6817`
Dataset (Experiments): `https://zenodo.org/records/15628604`

Funding *Wietze Koops*: Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation
Daniel Le Berre: BLaSST ANR-21-CE25-0010
Magnus O. Myreen: Swedish Research Council grant 2021-05165
Jakob Nordström: Swedish Research Council grants 2016-00782 and 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
Marc Vinyals: UoA FoS RDF 3729031

Acknowledgements We would like to acknowledge earlier work on *RoundingSat* by Jan Elffers, Jo Devriendt, and Stephan Gocht, and on *Sat4j* by Anthony Blomme and Romain Wallon, including preliminary contributions to proof logging. We are most grateful to Benjamin Bogø for useful feedback and for helping us to fix a bug in the proof logging code developed for this paper. Different subsets of the authors wish to thank the participants of the Dagstuhl workshops 22411 *Theory and Practice of SAT and Combinatorial Solving* and 23261 *SAT Encodings and Beyond* for several stimulating discussions. The first, fourth, and fifth authors also gratefully acknowledge that they have benefitted greatly from being part of the *Basic Algorithms Research Centre (BARC)* environment in Copenhagen financed by the Villum Investigator grant 54451. The computational experiments were enabled by resources provided by LUNARC at Lund University.

1 Introduction

Combinatorial search and optimization is a major success story in computer science, and the dramatic improvements in performance of combinatorial solvers over the last couple of decades allow them to be used for solving large-scale real-world problems in model checking [10], cryptanalysis [54], planning [64], kidney allocation for transplants [11, 12], protein design [2, 44], and many other application domains. But modern solvers are highly complex pieces of software, using sophisticated combinations of intelligent inference algorithms driven by intricate heuristics, and it is well documented that even the most mature state-of-the-art combinatorial solvers sometimes return “solutions” that do not satisfy the constraints, or erroneously claim infeasibility or optimality [1, 14, 20, 37]. Such errors could potentially have serious consequences for applications where correctness is a non-negotiable demand, and in more complex settings, where combinatorial solvers are used to solve subproblems, seemingly innocuous off-by-one mistakes can snowball into huge overall errors.

The Boolean satisfiability (SAT) community has pioneered the concept of *certifying solvers* as the most successful approach to date to address this problem. Certifying solvers do not only produce an answer, but also use *proof logging* to generate a machine-verifiable mathematical proof that this answer is correct. Many different SAT proof logging formats such as *RUP* [43], *TraceCheck* [9], *GRIT* [22], and *LRAT* [21] have been developed, with *DRAT* [46, 47, 71] established as the de facto standard; for the past decade, proof logging has been compulsory in the main track of the SAT competitions [63].

A big part of the success of SAT proof logging is that it is at the same time very easy to implement and very efficient to check. In the *DRAT* format, all that is required for proof logging is (essentially) for the solver to print the clauses it learns, meaning that the overhead of proof generation can be kept very small (say, less than 10% of solving time). Yet proof checkers are so fast that the time spent on verifying solver claims is within an order of magnitude of solving time, and can sometimes be made as efficient as solving at the price of adding additional proof details as in *LRAT* proofs [61]. Many SAT proof formats also come with formally verified proof checking backends [21, 22, 51, 66], meaning that the correctness of the final verdict is guaranteed by state-of-the-art formal verification techniques.

Inspired by the success of SAT proof logging, over the years there have been numerous attempts to develop proof logging in other combinatorial solving communities such as constraint programming [28, 60, 69], mixed integer programming [18, 30], satisfiability modulo theories (SMT) solving [3, 4, 8, 48, 65], model counting [15, 17, 36], and automated planning [33, 34, 58]. However, most of these attempts have run into the problem that it is hard to design simple proof systems that can support the full range of solver reasoning techniques used. And if one instead provides specialized proof rules for different techniques, then proof systems quickly become very complex, making efficient proof checking challenging.

In the last few years, *pseudo-Boolean (PB) reasoning* with 0–1 linear constraints has emerged as a simple and yet expressive proof logging approach for a wide range of solving paradigms and techniques such as SAT-based optimization (MaxSAT) [5, 6, 49], subgraph solving [38, 39, 40], constraint programming [31, 41, 55, 56, 57], automated planning [27], and dynamic programming [23]. Although having a single, unified proof logging method for a large number of seemingly very different combinatorial paradigms appears quite attractive, the actual performance in terms of proof generation and checking has remained at a proof-of-concept stage, far from what would be required for deployment in production-grade software. It seems fair to say that no combinatorial optimization paradigm has been able to deliver proof logging with the efficiency corresponding to what is available for SAT decision problems.

In this work, we present—to the best of our knowledge, for the first time—highly efficient and practical certified solving for a combinatorial optimization problem, thus going beyond the decision problems considered in SAT solving. Using the pseudo-Boolean proof system *VeriPB* together with the formally verified backend *CakePB*, we provide a toolchain for linear pseudo-Boolean optimization with fully formally verified conclusions. We show how to implement proof logging for the state-of-the-art pseudo-Boolean solvers *RoundingSat* [25, 26, 32] and *Sat4j* [52], covering the full range of advanced techniques used in these solvers. For the main bottleneck in previous pseudo-Boolean proof logging works, namely proof checking, we can now provide formally verified conclusions within a factor 20 of the solving time, getting close to the overhead factor of 9 traditionally required in the SAT competitions—this is orders of magnitude faster than even the unverified proof checking in prior work.

To provide some perspective on why our work is a nontrivial contribution, let us briefly explain how SAT proof logging works and why a similar approach is not possible for pseudo-Boolean solving. The workhorse of SAT proof logging is so-called *reverse unit propagation (RUP)* [43, 68]. If SAT conflict analysis produces a clause, say, $C \doteq x_1 \vee x_2 \vee \overline{x}_3$, from the current constraint database $\mathcal{C}$, then in order to verify the correctness of this derivation it is sufficient to assert the negation $\neg C \doteq \overline{x}_1 \wedge \overline{x}_2 \wedge x_3$ and check that so-called *unit propagation* on the constraint database $\mathcal{C}$ leads to contradiction.

The concept of *RUP* can be generalized to pseudo-Boolean constraints [31], where it means achieving integer bounds consistency, but RUP checks are not sufficient to certify pseudo-Boolean solving. At the risk of getting a bit technical, consider the constraints

$$C_1 \doteq 2x + 2y + 2z + u + v \geq 5 \tag{1}$$

$$C_2 \doteq 2\overline{x} + 2\overline{y} + 2\overline{z} + u + v \geq 5 \tag{2}$$

and suppose that a pseudo-Boolean solver decides to set $x = 0$. Then the constraint C_1 propagates $y = z = 1$, which leads to a conflict with C_2. Division-based pseudo-Boolean conflict analysis now weakens away u and v from C_1 to get $2x + 2y + 2z \geq 3$, divides by 2 to obtain $x + y + z \geq 2$, and finally multiplies this constraint by 2 and adds to C_1 to learn

$$C_3 \doteq u + v \geq 3\,. \tag{3}$$

It is easy to see that this learned constraint can never be satisfied for $\{0, 1\}$-valued variables, and so the solver has derived contradiction and can terminate.

The details of the conflict analysis above are not too important, but the crucial observation is that the learned constraint C_3 cannot be verified by RUP, because none of the constraints C_1, C_2, or $\neg C_3 \doteq \overline{u} + \overline{v} \leq 2$ causes any unit propagations. A more expensive check would be to compute if the linear programming relaxation of these constraints yields an empty polytope—this would be another way of showing that C_3 follows from C_1 and C_2. But such a check also fails here, since $x = y = z = \frac{1}{2}$ and $u = v = 1$ is a fractional solution satisfying C_1, C_2, and $\neg C_3$. The point of this technical detour is to illustrate that SAT-style RUP proof logging cannot be expected to work for pseudo-Boolean solvers, but instead we have to provide much more explicit, syntactic, information about how solvers infer constraints.

Implementing proof logging for the state-of-the-art solvers *RoundingSat* and *Sat4j* presents further challenges. In addition to pseudo-Boolean conflict analysis, which maps very naturally to pseudo-Boolean reasoning steps, *RoundingSat* uses a number of more advanced techniques for which it is much less obvious how to express the reasoning in terms of pseudo-Boolean proof rules. In other cases, the natural proof logging approach turns out to lead to performance issues, which have to be circumvented by using more complex solutions. *Sat4j* does not cause as many problems in terms of different reasoning techniques, but has a much larger codebase developed over more than two decades. Adding proof generation to this codebase, and finding all the different places where the solver might, for instance, perform minor simplifications of constraints that do not really change anything in terms of semantics but cause syntactic checks to detect unexplained changes and fail, is highly nontrivial.

Our formally verified proof checking is performed in two stages, as is also commonly done for SAT solving. In the first stage, the unverified checker *VeriPB elaborates* the proof to a more restrictive format for which no search or propagation is required. This elaborated proof is then checked by the formally verified backend *CakePB*. We have made significant efforts to identify bottlenecks in proof elaboration and formal checking, and this work is a big part of the explanation for why the whole verified proof checking workflow is now so efficient.

The rest of this paper is organized as follows. After reviewing the basics of pseudo-Boolean reasoning and proof logging in Section 2, we discuss proof logging for advanced solving techniques in Sections 3 and 4. In Section 5 we report on our work on optimizing the proof logging and checking pipeline. We present our experimental evaluation in Section 6 and provide some concluding remarks in Section 7. Some additional technical details regarding our proof logging derivations are provided in Appendices A and B.

2 Preliminaries

We start with a condensed review of pseudo-Boolean reasoning, referring the reader to [13, 42] for more details on *VeriPB* and to [16] for more on proof systems in general. This is followed by a brief description of the concrete syntax and semantics of *VeriPB* proofs.

2.1 Pseudo-Boolean Reasoning and the Cutting Planes Proof System

We work exclusively with *Boolean variables*, i.e., variables taking values 0 or 1. A literal ℓ over a variable x is either x or its negation $\overline{x} = 1 - x$. By a *pseudo-Boolean (PB) constraint* we mean an inequality of the form $C \doteq \sum_i a_i \ell_i \geq A$, where a_i and A are integers (and where we write $\doteq$ to denote syntactic equality). We can assume without loss of generality that constraints are written in *normalized form* [45] with all *coefficients* a_i and the *degree* A being non-negative, and all literals being over distinct variables. A *pseudo-Boolean formula* is a conjunction $F \doteq \bigwedge_j C_j$ of pseudo-Boolean constraints.

A *substitution* ω is a function from variables to literals or 0 or 1. We extend the domain of substitutions to literals in the natural way by respecting the meaning of negation. For a constraint $C \doteq \sum_i a_i \ell_i \geq A$, we define $C\restriction_\omega \doteq \sum_i a_i \omega(\ell_i) \geq A$ as the constraint where we perform the substitution specified by ω (and simplify the resulting constraint). For a formula F, we define $F\restriction_\omega \doteq \bigwedge_j C_j\restriction_\omega$. We sometimes write $\ell_1 \mapsto \ell_2$ for $\omega(\ell_1) = \ell_2$ when the substitution ω is clear from context or immaterial.

An *assignment* α is a substitution that maps literals to $\{0, 1\}$ only. A *partial assignment* can also leave literals ℓ_i untouched by mapping ℓ_i to itself. A partial assignment ρ *extends* another partial assignment σ if ρ agrees with σ on all literals that σ maps to $\{0, 1\}$, i.e., if $\sigma(\ell) \in \{0, 1\}$ implies $\rho(\ell) = \sigma(\ell)$. A (partial) assignment α *satisfies* a constraint $C \doteq \sum_i a_i \ell_i \geq A$ if $\sum_{\alpha(\ell_i)=1} a_i \geq A$ (note that we assume here that C is written in normalized form). An assignment satisfies a formula $F \doteq \bigwedge_j C_j$ if it satisfies all its constraints, in which case we call it a *solution*. The *decision problem* for the pseudo-Boolean formula F is to determine whether F has a solution. In an *optimization problem*, we are additionally given an integer linear *objective* $f \doteq \sum_i w_i \ell_i$ that we want to minimize over all solutions to F.

We view the *cutting planes proof system* [19] as operating on pseudo-Boolean constraints in normalized form. Our derivation rules, which all preserve solutions, are as follows. We can always introduce the *literal axiom* $\ell_i \geq 0$ for any literal ℓ_i. Given two constraints $C_1 \doteq \sum_i a_i \ell_i \geq A$ and $C_2 \doteq \sum_i b_i \ell_i \geq B$, we can add them to obtain $C_1 + C_2 \doteq \sum_i (a_i + b_i)\ell_i \geq A + B$. For a positive integer c, we can multiply $C_1 \doteq \sum_i a_i \ell_i \geq A$ by c to get $c \cdot C_1 \doteq \sum_i (ca_i)\ell_i \geq cA$, or divide by c and round to get $\sum_i \left\lceil \frac{a_i}{c} \right\rceil \ell_i \geq \left\lceil \frac{A}{c} \right\rceil$. Finally, we can *saturate* a constraint $\sum_i a_i \ell_i \geq A$ to obtain $\sum_i \min\{a_i, A\}\ell_i \geq A$.

The *slack* of a normalized constraint $C \doteq \sum_i a_i \ell_i \geq A$ with respect to a partial assignment ρ is the sum of the coefficients of all non-falsified literals minus the degree, i.e.,

$$slack\left(\textstyle\sum_i a_i \ell_i \geq A; \rho\right) = \textstyle\sum_{\rho(\ell_i) \neq 0} a_i - A\,, \tag{4}$$

and measures how close ρ is to falsifying C. If $slack(C; \rho) < 0$, then C is *conflicting* under ρ and no extension of ρ can satisfy C. If ρ does not assign ℓ_i but $0 \leq slack(C; \rho) < a_i$, then we say that C *propagates* ℓ_i under ρ, since any extension of ρ that satisfies C must set $\ell_i = 1$.

A *unit constraint* (or just *unit*) is a constraint $\ell \geq 1$. During *unit propagation* on F under ρ, we iteratively extend ρ by assigning to 1 all literals ℓ_i that are propagated by a constraint in F until either no further literals propagate or some constraint in F is conflicting. In the latter case, we say that F propagates to *conflict* under ρ. The negation of a constraint $C \doteq \sum_i a_i \ell_i \geq A$ is $\sum_i a_i \ell_i \leq A - 1$, which rewritten to normalized form becomes

$$\neg C \doteq \textstyle\sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1\,. \tag{5}$$

It is easy to verify that adding C and $\neg C$ yields the contradictory constraint $0 \geq 1$. We say that F implies C by *reverse unit propagation (RUP)* if $F \wedge \neg C$ propagates to conflict (in which case it is clear that any solution to F must also satisfy C).

2.2 *VeriPB* Syntax and Semantics

We next discuss the syntax and semantics of the *VeriPB* implementation [70] in more detail. The reader should be advised, however, that the presentation is streamlined and simplified to cover precisely the subset of the *VeriPB* system that is needed for this paper.

VeriPB maintains a database of constraints $\mathcal{C}$, where each constraint can be referred to using a *constraint ID* that is a positive integer (assigned consecutively as the derivation proceeds, with the first IDs $1, 2, \ldots, S$ identified with the S pseudo-Boolean constraints in the input formula). In our *VeriPB* code examples we will use displayed math equation labels as constraint IDs for ease of reference, so that the constraint

$$C_6 \doteq 2x_1 + 5x_2 + x_3 \geq 4 \tag{6}$$

is referred to using constraint ID `6`. To make it easier to refer to constraints, they can also be assigned @-labels that refer to their integer IDs when they are derived.

Cutting planes derivations are written in reverse Polish notation prefixed by `pol`. *VeriPB* uses `~` to represent negation, `x`*i* and `~x`*i* for the literal axioms $x_i \geq 0$ and $\overline{x}_i \geq 0$, and `+`, `*`, `d`, and `s` for addition, multiplication, division, and saturation, respectively. In addition, *VeriPB* supports literal *weakening* (denoted by `w`), which eliminates a term $a\ell$ from a constraint by adding $a \cdot (\overline{\ell} \geq 0)$ (where we note that ℓ and $\overline{\ell}$ cancel to leave an additive 1 when added since $\ell + \overline{\ell} = 1$ must always hold). To illustrate all of what has been said so far, the line

```
@newconstr pol 6 s ~x2 2 * + x3 w 2 d
```
$$x_1 + x_2 \geq 1 \tag{7}$$

first saturates constraint C_6 (yielding $2x_1 + 4x_2 + x_3 \geq 4$), then takes the literal axiom $\overline{x}_2 \geq 0$, multiplies it by 2 and adds it (yielding $2x_1 + 2x_2 + x_3 \geq 2$), then weakens on x_3 by adding the literal axiom $\overline{x}_3 \geq 0$ (yielding $2x_1 + 2x_2 \geq 1$), and finally divides by 2. The end result is the constraint $x_1 + x_2 \geq 1$, which is stored in the constraint database with the next available ID `7`, and the (optional) label `@newconstr` functions as an alias for this number in the rest of the proof. In our example derivations, we write out the derived constraint on the right for clarity, but stating this constraint is not part of the *VeriPB* syntax for `pol` lines.

Reverse unit propagation (RUP) steps can be *annotated* with information about which constraints propagate to contradiction, as in *LRAT* [21], or stated without such information, as in *DRAT* [71]. Thus, the line

```
rup +1 x2 >= 1 ; 6
```
$$x_2 \geq 1 \tag{8}$$

derives $x_2 \geq 1$ after successful unit propagation to conflict on $\{2x_1 + 5x_2 + x_3 \geq 4, \overline{x}_2 \geq 1\}$, i.e., the constraint `6` together with the negation of $x_2 \geq 1$. Without the annotation `6`, *VeriPB* would instead propagate on the full constraint database and the negation of $x_2 \geq 1$ (which will always work, but might be less efficient). *VeriPB* also supports simple *syntactic implication* `ia`, which checks whether a specified constraint C can be derived from a single constraint in the database by adding literal axioms and saturating. A constraint ID can be added as an annotation to specify which concrete constraint syntactically implies C.

All derivation rules described so far are convenient shorthands for cutting planes derivations, but *VeriPB* also has two *strengthening rules* for deriving non-implied constraints that are guaranteed not to change the satisfiability status or optimal value of the input. We will not need the *dominance-based strengthening rule* in this paper, and so we ignore it and focus instead on the *redundance-based strengthening*, or *redundance* for short. A constraint C is inferred by redundance by specifying a *witness* substitution ω and proving the implication

$$\mathcal{C} \cup \{\neg C\} \vDash (\mathcal{C} \cup \{C\})\restriction_\omega \cup \{f\restriction_\omega \leq f\} \tag{9}$$

by providing cutting planes subproofs of all constraints on the right-hand side (which we refer to as *proof obligations*). In practice, many of these subproofs can be omitted since *VeriPB* will infer them automatically by, e.g, RUP or syntactic implication. As an example, if y_1 is a fresh variable that has not appeared in the derivation before, then we can define y_1 to imply $2x_1 + x_2 + x_3 \geq 3$ by the derivation step

```
red +3 ~y1 +2 x1 +1 x2 +1 x3 >= 3 ; y1 -> 0
```

$$3\overline{y}_1 + 2x_1 + x_2 + x_3 \geq 3 \tag{10}$$

introducing the constraint (10), which has the intended meaning since $y_1 = 1$ clearly forces the inequality $2x_1 + x_2 + x_3 \geq 3$ to hold. This derivation step is valid since the witness $\omega = \{y_1 \mapsto 0\}$ only affects the new constraint $3\overline{y}_1 + 2x_1 + x_2 + x_3 \geq 3$, which it fixes to true, and all other proof obligations are identical to the premises on the left-hand side or vacuous. In general, any subproofs required are provided after the witness specification delimited by `begin` and `end`. In case only one proof obligation needs an explicit proof and this proof only uses one constraint from $\mathcal{C}$, we will refer to that constraint as the *justification*.

When the witness is empty, the redundance rule allows to derive a constraint C using proof by contradiction. Namely, in this case one of the required implications to prove is $\mathcal{C} \cup \{\neg C\} \vDash C$, which since $C + \neg C \doteq 0 \geq 1$ is equivalent to deriving contradiction.

Solutions to the input formula F can be specified using the `sol` rule, which checks if the specified assignment is a valid solution. For an optimization problem with objective function $f \doteq \sum_i w_i \ell_i$ to be minimized, the `soli` rule also adds a *solution-improving constraint*

$$\textstyle\sum_i w_i \ell_i \leq -1 + \sum_i w_i \cdot \alpha(\ell_i) \tag{11}$$

to the constraint database (but in normalized form), which enforces that all future solutions will yield a better objective value than α.

A *VeriPB* proof for the decision problem for F should establish either satisfiability by logging a solution to F or else unsatisfiability by deriving contradiction (in the form of a constraint that has negative slack with respect to the empty assignment). For a (feasible) optimization problem, a proof that the optimal value of the objective f lies in the interval $[LB, UB]$ should log a solution α to F achieving $f(\alpha) = UB$ and derive a constraint $f \geq LB$.

3 Proof Logging for Linear Programming Integration

Proof logging for the basic conflict-driven search in the pseudo-Boolean solvers *RoundingSat* and *Sat4j* is relatively straightforward, since both solvers use the cutting planes proof system for their conflict analysis. However, *RoundingSat* is tightly integrated with a linear programming (LP) solver [25], which adds extra complications as explained in this section.

Most aspects of the LP reasoning in *RoundingSat* can be expressed in terms of (positive) linear combinations of constraints and can therefore be logged directly with `pol` lines, and Chvátal-Gomory cut generation is the same as cutting planes division. However, *mixed-integer rounding (MIR)* cuts [53] are not natively supported by the *VeriPB* proof system, and the cut generation procedure turns inequalities into equalities by introducing non-Boolean *integral slack variables*, which cannot be directly expressed in the proof system.

As shown in [25], the MIR cut with divisor $d \in \mathbb{N}^+$ applied to $\sum_i a_i \ell_i \geq A$ yields

$$\textstyle\sum_i \Big(\min\{a_i \bmod d, A \bmod d\} + \Big\lfloor \frac{a_i}{d} \Big\rfloor (A \bmod d)\Big)\, \ell_i \geq \Big\lceil \frac{A}{d} \Big\rceil (A \bmod d)\,, \tag{12}$$

which is in general stronger than the constraint

$$\textstyle\sum_i \Big(\Big\lceil \frac{a_i}{d} \Big\rceil (A \bmod d)\Big)\, \ell_i \geq \Big\lceil \frac{A}{d} \Big\rceil (A \bmod d) \tag{13}$$

that we obtain by cutting planes division from $\sum_i a_i\ell_i \geq A$ with divisor d and then multiplying by $A \bmod d$, which we call the *multiplier* of the MIR cut. To see this, note that when $0 < a_i \bmod d < A \bmod d$, (12) has a smaller coefficient for the literal ℓ_i than (13).

We illustrate with an example how MIR cuts are generated in *RoundingSat*. As input, consider the two constraints

$$C_{14} \doteq 4x_1 + 7x_2 + 5x_3 + 3x_5 \geq 9\,, \tag{14}$$

$$C_{15} \doteq 3x_1 + 2x_2 + 2x_4 \geq 3\,, \tag{15}$$

and suppose that the multipliers are $\lambda_{14} = 1$ and $\lambda_{15} = 4$, the divisor is $d = 5$, and $P = \{x_1, x_4\}$ is the set of variables on which we *partially weaken* (i.e., add literal axioms to reduce the coefficients to the largest multiple of the divisor d). For purposes of computation, we introduce an integral slack variable $s_j \geq 0$ for each constraint C_j to obtain the equalities

$$C'_{14} \doteq 4x_1 + 7x_2 + 5x_3 + 3x_5 - s_{14} = 9\,, \tag{16}$$

$$C'_{15} \doteq 3x_1 + 2x_2 + 2x_4 - s_{15} = 3\,. \tag{17}$$

We take the linear combination

$$\textstyle\sum_j \lambda_j C'_j \doteq 16x_1 + 15x_2 + 5x_3 + 8x_4 + 3x_5 - s_{14} - 4s_{15} = 21 \tag{18}$$

and continue working on the greater-than-or-equal part of this equality. Partially weakening on x_1 and x_4 by adding $\overline{x}_1 \geq 0$ and $3 \cdot (\overline{x}_4 \geq 0)$ yields

$$15x_1 + 15x_2 + 5x_3 + 5x_4 + 3x_5 - s_{14} - 4s_{15} \geq 17\,. \tag{19}$$

Next, we apply a MIR cut with divisor $d = 5$ to get the inequality

$$6x_1 + 6x_2 + 2x_3 + 2x_4 + 2x_5 - s_{15} \geq 8\,. \tag{20}$$

For later use, we note that the multiplier of this MIR cut is $H = (-3) \bmod 5 = 2$. To obtain the final cut, we *subtract* C'_{15} to cancel the integral slack variable s_{15}, which yields

$$3x_1 + 4x_2 + 2x_3 + 2x_5 \geq 5\,. \tag{21}$$

Our formal *VeriPB* derivation of the MIR cut (21) is a proof by contradiction starting from the negation of this constraint, which is

$$3\overline{x}_1 + 4\overline{x}_2 + 2\overline{x}_3 + 2\overline{x}_5 \geq 7\,. \tag{22}$$

We first add $\lambda_{14} = 1$ times C_{14}, but we postpone adding C_{15}. Intuitively, this is done in order to be able to compensate for the subtraction later. In general, we add $\lambda_j C_j$ for all j such that $\lambda_j \leq H$, where H is the multiplier of the MIR cut. This results in the constraint

$$x_1 + 3x_2 + 3x_3 + x_5 \geq 5\,. \tag{23}$$

Next, we partially weaken on the variables in P, to reduce their coefficients to the largest smaller multiple of $d - H = 3$. In this case, we add $\overline{x}_1 \geq 0$, which yields

$$3x_2 + 3x_3 + x_5 \geq 4\,. \tag{24}$$

Now we divide by $d - H = 3$, add $H = 2$ times this to (22), and finally add C_{15}, which yields the sequence of constraints

$$x_2 + x_3 + x_5 \geq 2 \tag{25}$$

$$3\overline{x}_1 + 2\overline{x}_2 \geq 5 \tag{26}$$

$$2x_4 \geq 3 \tag{27}$$

ending with contradiction, establishing correctness of the cut (21). In general, we add $(d - \lambda_j)C_j$ for all j such that $\lambda_j > H$. The code fragment

`red +3 x1 +4 x2 +2 x3 +2 x5 >= 5 ; ; begin` $\quad 3\overline{x}_1 + 4\overline{x}_2 + 2\overline{x}_3 + 2\overline{x}_5 \geq 7 \quad (28)$

`  pol 28 14 +` $\quad x_1 + 3x_2 + 3x_3 + x_5 \geq 5 \quad (29)$

`  pol 29 ~x1 + 3 d` $\quad x_2 + x_3 + x_5 \geq 2 \quad (30)$

`  pol 28 30 2 * + 15 +` $\quad 2x_4 \geq 3 \quad (31)$

`end` $\quad 3x_1 + 4x_2 + 2x_3 + 2x_5 \geq 5 \quad (32)$

shows how the MIR cut example can be formalized as a valid *VeriPB* derivation.

In Appendix A, we describe how proof logging for MIR cuts is done in full generality.

4 Proof Logging for Pseudo-Boolean Core-Guided Optimization

RoundingSat solves optimization problems using linear search, core-guided search, or a "hybrid" combination of the two [26]. Proof logging for linear search is straightforward, since the solution-improving constraint added by the solver is what is derived by the `soli` rule. For core-guided optimization the proof logging is more complicated, but we explain by example below. The overall approach is similar to core-guided optimization in MaxSAT [6], from which the technique has been imported, but the details differ significantly since the solver deals with general pseudo-Boolean constraints rather than clauses. Pseudo-Boolean constraints in the *VeriPB* code examples of this section are mathematically formatted to improve clarity, which is not correct *VeriPB* syntax.

Consider a minimization problem with objective function $f_o \doteq x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6$. We initially have an implicit lower bound $f_o \geq 0$.

We begin by running the solver with an additional assumption that $f_o = 0$, equivalently $x_1 = \ldots = x_6 = 0$. We either find a solution, which must be optimal, or our assumption is too strong. In that case the solver finds a conflict, say $3x_2 + 2x_3 + x_4 + x_5 \geq 4$, which we derive with some proof line

`pol (...)` $\quad 3x_2 + 2x_3 + x_4 + x_5 \geq 4 \quad (33)$

as done for conflict analysis. Such a *core constraint* shows that the assumptions are inconsistent with the input formula. We then turn the core constraint (33) into a cardinality constraint, also called a *cardinality core*. The most obvious way to do so is to divide by the largest coefficient in the constraint, but it is possible to obtain stronger cardinality constraints through a more complicated algorithm. All of these algorithms can be logged as a sequence of weakening, saturation, and division steps. In our case, the derivation step

`pol 33 3 d` $\quad x_2 + x_3 + x_4 + x_5 \geq 2 \quad (34)$

divides by 3 to get $x_2 + x_3 + x_4 + x_5 \geq 2$. We proceed to rewrite the objective using the cardinality core (34). To do so, we introduce two new slack variables y_3 and y_4 so that we can write $x_2+x_3+x_4+x_5 = 2+y_3+y_4$. Equivalently, writing $\Delta_1 \doteq x_2+x_3+x_4+x_5-(2+y_3+y_4)$, we have $\Delta_1 = 0$. We log the two sides of this equality using the redundance rule. We first log the *at-most* constraint $\Delta_1 \leq 0$ with witness $y_3 = y_4 = 1$ and trivial justification. Then we log the *at-least* constraint $\Delta_1 \geq 0$ in two steps, with witnesses $y_3 = 0$ and $y_4 = 0$, and justifications the cardinality core (34) and the previous step (36):

`red` $x_2 + x_3 + x_4 + x_5 \leq 2 + y_3 + y_4$ `; y3 -> 1 y4 -> 1` (35)

`red` $x_2 + x_3 + x_4 + x_5 \geq 2 + y_3$ `; y3 -> 0` (36)

`red` $x_2 + x_3 + x_4 + x_5 \geq 2 + y_3 + y_4$ `; y4 -> 0` (37)

To speed up search we give slack variables the additional meaning that y_j is true if and only if $x_2 + x_3 + x_4 + x_5 \geq j$. We do so with an ordering constraint of the form $y_3 \geq y_4$, which we derive using the redundance rule and witness swapping y_3 and y_4:

red $y_3 \geq y_4$; `y3 -> y4 y4 -> y3` (38)

We then proceed to rewrite the objective to $f_r \doteq f_o - \Delta$, where $\Delta \doteq 2\Delta_1$ is the difference between the reformulated and original objectives. The multiplier 2 is the largest number that keeps all coefficients of f_r positive. The new objective is $f_r \doteq x_1 + x_3 + 2x_4 + 3x_5 + 6x_6 + 2y_3 + 2y_4 + 4$, and, because $\Delta = 0$, it is equivalent to the original objective. Furthermore, the new objective contains a constant term 4, which gives us a lower bound of 4. We log the *global reformulation constraint* $\Delta \geq 0$ as 2 times (37), then log the *objective lower bound* $f_o \geq 4$ by first logging the trivial constraint $f_r \geq 4$ and then adding $\Delta \geq 0$:

`pol 37 2 *` $\quad 2x_2 + 2x_3 + 2x_4 + 2x_5 \geq 4 + 2y_3 + 2y_4$ (39)

rup $x_1 + x_3 + 2x_4 + 3x_5 + 6x_6 + 2y_3 + 2y_4 \geq 0$ (40)

`pol 40 39 +` $\quad x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6 \geq 4$ (41)

At this point we run the solver with assumptions $x_i = y_i = 0$ again. Suppose that we now obtain a conflict constraint $x_4 + x_5 + x_6 + y_3 \geq 1$, which is already a cardinality core.

We now have $\Delta_2 \doteq x_4 + x_5 + x_6 + y_3 - (1 + z_2 + z_3 + z_4)$. We introduce at-most, at-least, and ordering constraints:

red $x_4 + x_5 + x_6 + y_3 \leq 1 + z_2 + z_3 + z_4$; `z2 -> 1 z3 -> 1 z4 -> 1` (42)

red $x_4 + x_5 + x_6 + y_3 \geq 1 + z_2$; `z2 -> 0` (43)

red $x_4 + x_5 + x_6 + y_3 \geq 1 + z_2 + z_3$; `z3 -> 0` (44)

red $x_4 + x_5 + x_6 + y_3 \geq 1 + z_2 + z_3 + z_4$; `z4 -> 0` (45)

red $z_2 \geq z_3$; `z2 -> z3 z3 -> z2` (46)

red $z_3 \geq z_4$; `z2 -> z4 z3 -> z2 z4 -> z3` (47)

We rewrite the objective again, with $\Delta \doteq 2\Delta_1 + 2\Delta_2$. The new objective is $f_r \doteq x_1 + x_3 + x_5 + 4x_6 + 2y_4 + 2z_2 + 2z_3 + 2z_4 + 6$. We derive the global reformulation constraint $\Delta \geq 0$ from the previous global reformulation constraint, and then we log the lower bound $f_o \geq 6$:

`pol 39 45 2 * +` $\quad 2x_2 + 2x_3 + 4x_4 + 4x_5 + 2x_6 \geq 6 + 2y_4 + 2z_2 + 2z_3 + 2z_4$ (48)

rup $x_1 + x_3 + x_5 + 4x_6 + 2y_4 + 2z_2 + 2z_3 + 2z_4 \geq 0$ (49)

`pol 49 48 +` $\quad x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6 \geq 6$ (50)

We run the solver with assumptions $x_i = y_i = z_i = 0$ yet another time. Suppose that these assumptions extend to a solution. We log the solution using `soli` and obtain the objective upper bound $f_o \leq 6$, completing the proof.

Expert readers may note that in practice it is too slow to introduce all the slack variables when we reformulate the objective, and we need to introduce variables lazily instead. This means that when we find the first cardinality core, we only introduce y_3. Therefore, we define $\Delta_1^{(1)} \doteq x_2 + x_3 + x_4 + x_5 - (2 + y_3)$, where the notation $\Delta_i^{(j)}$ refers to the i-th cardinality core after j slack variables have been introduced. We log constraints encoding that y_3 is true if and only if $x_2 + x_3 + x_4 + x_5 \geq 3$ is true. We first log the at-most constraint encoding that $x_2 + x_3 + x_4 + x_5 \geq 3$ implies y_3

red $x_2 + x_3 + x_4 + x_5 \leq 2 + 2y_3$; `y3 -> 1` (51) with witness $y_3 = 1$ and trivial justification, and then the at-least constraint encoding that y_3 implies $x_1 + x_3 + x_4 + x_5 \geq 3$, or equivalently $\Delta_1^{(1)} \geq 0$,

`red` $x_2 + x_3 + x_4 + x_5 \geq 2 + y_3$ `; y3 -> 0` (52)

with witness $y_3 = 0$ and justification the cardinality core (34).

We rewrite the objective to $f_r \doteq f_o - \Delta$ with $\Delta \doteq 2\Delta_1^{(1)}$. Observe that we no longer have $\Delta = 0$ but only $\Delta \geq 0$, therefore all we can say about the objective is $f_o \leq f_r$. Nevertheless, this is enough to derive a lower bound on the objective.

Suppose that our next core constraint is $x_4 + x_5 + x_6 + y_3 \geq 1$. We now have $\Delta_2^{(1)} \doteq x_4 + x_5 + x_6 + y_3 - (1 + z_2)$. We introduce at-most and at-least constraints:

`red` $x_4 + x_5 + x_6 + y_3 \leq 1 + 3z_2$ `; z2 -> 1` (53)

`red` $x_4 + x_5 + x_6 + y_3 \geq 1 + z_2$ `; z2 -> 0` (54)

Since y_3 appears in this cardinality core and will disappear from the objective, we introduce y_4 and extend $\Delta_1^{(1)}$ to $\Delta_1^{(2)} \doteq x_2 + x_3 + x_4 + x_5 - (2 + y_3 + y_4)$. We first log the at-most constraint with witness $y_4 = y_3$ and justification the previous at-most constraint (51),

`red` $x_2 + x_3 + x_4 + x_5 \leq 2 + y_3 + y_4$ `; y4 -> y3` (55)

and then the at-least constraint with witness $y_4 = 0$ and justification the previous at-least constraint (52).

`red` $x_2 + x_3 + x_4 + x_5 \geq 2 + y_3 + y_4$ `; y4 -> 0` (56)

We derive the ordering constraint from previous constraints instead of by redundance, as there might be constraints containing y_3 which are not obvious to derive under the witness:

`pol 56 51 +` $y_3 \geq y_4$ (57)

We rewrite the objective to $f_r \doteq f_o - \Delta$ with $\Delta \doteq 2\Delta_1^{(2)} + 2\Delta_2^{(1)}$. As $\Delta_1^{(1)}$ has been extended to $\Delta_1^{(2)}$, we cannot obtain the new global reformulation constraint from the previous global reformulation constraint, therefore we rederive it from scratch.

`pol 56 2 * 54 2 * +` $2x_2 + 2x_3 + 4x_4 + 4x_5 + 2x_6 \geq 6 + 2y_4 + 2z_2$ (58)

Assuming that we next find an optimal solution, we complete the proof analogously to the eager case.

We provide proofs of correctness and some further constructions in Appendix B. Most importantly, we touched on eager and lazy sum encodings, but we also support a third reified encoding. Additionally, if we have upper bounds on the objective, we can use that information to derive unit constraints by hardening, and to derive upper bounds on cores, which in turn can be used to derive stronger at-most constraints and stop extending lazy variables earlier.

5 Implementation

In this section, we discuss our work on implementing proof logging in the pseudo-Boolean solvers *Sat4j* and *RoundingSat*, and how we have optimized the proof checkers *VeriPB* and *CakePB* to get close to the levels of SAT proof logging performance.

5.1 Proof Logging in *Sat4j*

The solver *Sat4j* [52] for SAT and pseudo-Boolean problems was designed more than 20 years ago, when proof logging was not a standard feature, but the fine-grained event-driven design allows for precise visualization and remote control of solver behaviour [7]. As such, when *DRUP* proof logging became mandatory in the SAT 2013 competition, including support for this was very fast. In 2024 incremental proof logging with *IDRUP* [35] was added. Such support for incremental solver calls also paved the way for proof logging for optimization.

The *Sat4j* optimization engine uses linear solution-improving search, for which proof logging is straightforward. However, for the conflict analysis the solver has two versions *Sat4j Resolution* with SAT-style, clausal, reasoning and *Sat4j Cutting Planes* with cutting-planes-based reasoning. Proof logging for SAT conflict analysis is easy to implement with RUP statements and is robust to syntactic changes of the underlying constraints, but dealing with cutting planes derivations requires exact control of the syntactic representation of constraints.

Sat4j inherits from *MiniSat* [29] that the constraints are simplified during parsing, so that, e.g., satisfied constraints are ignored and variables with fixed values are removed from constraints. In addition, unit propagation and saturation are applied. RUP proofs can essentially be written as if these simplifications were not performed, but explicit cutting planes proofs quickly fail when there is a mismatch between the original input constraint and the simplified constraint stored in the constraint database. Dealing with this turns out to be a formidable challenge. *Sat4j* has more than 30 different implementations of constraints with individual simplification rules, and any changes to the 15-20-year-old code must be limited so that none of the many software packages that depend on the solver breaks.

The @-labels for constraints discussed in Section 2.2 make book-keeping easier, in that we can represent falsified literals $x_j = 0$ by unit constraints $\overline{x}_j \geq 1$, which we can derive by RUP and give labels `@xj`. Removing a falsified literal then amounts to adding `@xj` to the constraint, whereas satisfied literals can simply be removed by weakening.

To illustrate how this works, suppose that the Mth input constraint read by *Sat4j* is $6x_1 + 2x_2 + x_3 + x_4 \geq 5$ and that $x_4 = 0$ is already known at this point. Then the falsified literal x_4 is removed to derive $6x_1 + 2x_2 + x_3 \geq 5$; saturation is applied to yield $5x_1 + 2x_2 + x_3 \geq 5$; and finally $x_1 = 1$ is propagated. The first two steps yields proof lines

`@x4 rup 1 ~x4 >= 1` $\qquad \overline{x}_4 \geq 1 \quad (59)$

`@M pol` M `@x4 + s` $\qquad 5x_1 + 2x_2 + x_3 \geq 5 \quad (60)$

while the unit $x_1 \geq 1$ is only derived later when needed. Note that because of the intermediate derivations the simplified version of the Mth input constraint will *not* get constraint ID M, but by defining the label `@M` we can avoid having to keep track of the exact number.

Continuing our example, suppose that the solver later reads the Nth input constraint $3\overline{x}_1 + 2x_2 + 2x_3 + \overline{x}_4 + 2x_5 \geq 3$. Then the solver simplifies this to $2x_2 + 2x_3 + 2x_5 \geq 2$, which is semantically equivalent to the clause $x_2 \vee x_3 \vee x_5$, and justifies this with the proof steps

`@x1 rup 1 x1 >= 1` $\qquad x_1 \geq 1 \quad (61)$

`@N pol` N `@x1 3 * + x4 w` $\qquad 2x_2 + 2x_3 + 2x_5 \geq 2 \quad (62)$

Using labels for input constraints and fixed literals make proof generation in *Sat4j* significantly simpler in that we just use a flag to keep track of whether the Nth constraint was simplified or not, and to refer to this constraint by `@N` or N accordingly. Another helpful *VeriPB* feature during implementation was that the proof can be terminated at any time and checked for correctness with respect to the upper and lower bound on the objective known at that time.

5.2 Proof Logging in *RoundingSat*

An important part of making the proof logging in *RoundingSat* efficient is to deal with fixed variable assignments $x \geq 1$ or $x \leq 0$ inferred by the solver (i.e., unit constraints). As in several previous works (e.g., [6, 30, 38, 50]), our work on implementing proof logging helped us find and eliminate bugs which previous extensive testing had failed to detect.

RoundingSat simplifies all derived constraints $C \doteq \sum_i a_i\ell_i \geq A$ by removing unit constraints. Satisfied literals $\ell_i \geq 1$ can just be weakened away, but falsified literals $\ell_j \leq 0$ require a justification. Adding $a_j \cdot (\ell_j \leq 0)$ to C for all falsified literals causes performance issues during proof checking. A better solution turns out to be to derive the fully simplified version of C in one go using a RUP statement annotated with the units $\ell_j \leq 0$ used.

In contrast to constraints learned during conflict analysis, unit constraints are typically discovered during propagation before the solver has made any decisions (i.e., at *decision level* 0). To see why proof logging for this scenario requires care, consider the constraints

$$C_1 \doteq x_2 + x_3 + 2x_4 + 2x_5 + 4x_6 \geq 7 \tag{63}$$

$$C_2 \doteq \overline{x}_1 + \overline{x}_2 + 2\overline{x}_3 + 2\overline{x}_4 + 4\overline{x}_5 + 4\overline{x}_6 \geq 7 \tag{64}$$

unit propagating $x_6 = 1$, $x_5 = 0$, $x_4 = 1$, $x_3 = 0$, $x_2 = 1$, and $x_1 = 0$ in that order. Explicit cutting planes derivations of these units would require adding all previous unit constraints to either C_1 or C_2, yielding a quadratic number of steps overall. And this is not just a theoretical pathological case—we observed instances where constraints propagated many units and where writing proofs for such units dominated the cost of proof logging. What we do instead is to derive such units using annotated RUP, explicitly mentioning which constraints are needed for propagation. While checking such RUP steps can in theory incur a linear overhead for "ping-ponging" propagations as between constraints (63) and (64) above, this does not seem to be a problem in practice.

The fact that cutting planes proof depends on the exact syntactic representation of constraints is not only a source of complications, as discussed in Section 5.1, but can also help detect bugs, including mistakes that do not affect soundness but makes solver reasoning weaker than it should. When generating MIR cuts as in (12), *RoundingSat* computed the coefficient $\left(\frac{a_i}{d} + 1\right)(A \bmod d)$ for ℓ_i when $d \mid a_i$ and $a_i < 0$ instead of the correct value $\frac{a_i}{d}(A \bmod d)$. Since the resulting coefficient is always larger, this is not unsound, but yields a weaker constraint than intended. This was discovered since the constraint derived in the proof did not match what was in the solver constraint database.

Another example, which is related to the discussion at the end of Section 4, is that when an objective upper bound $f \leq UB$ is known and X is a linear form such that $f - wX \geq 0$ holds, then the solver derived $X \leq \lfloor UB/w \rfloor$. However, in this scenario the constraint database in fact contains the solution-improving constraint $f \leq UB - 1$, and so the solver should infer the stronger constraint $X \leq \lfloor (UB-1)/w \rfloor$ instead. This was again discovered thanks to a mismatch between proof log and constraint database.

5.3 Optimizations in *VeriPB*

We have done substantial work on *VeriPB* to improve performance and robustness of proof checking and elaboration. Much of this concerns technical implementation details that are hard to explain briefly, but we try to give two examples below.

To make it convenient to log solutions with the `sol` and `soli` rules, *VeriPB* is guaranteed to apply unit propagation before checking that the provided assignment satisfies all constraints. Such propagation can cause substantial overhead, especially when solutions are logged frequently during optimization. Since pseudo-Boolean solvers can easily specify the full assignment to all variables, we modified the default behaviour of *VeriPB* to check first if the provided assignment satisfies all constraints before applying unit propagation.

We have also improved *autoproving*, i.e., the ability of *VeriPB* to automatically infer constraints, by changing the syntactic implication check so that it first applies unit propagation to the premises. Such autoproving is crucial to be able to minimize the size of the generated proofs, especially for strengthening rules, but involves delicate trade-offs with performance.

5.4 Optimizations in *CakePB*

Significant effort has also been invested to speed up *CakePB* and to update the formal verification following our code changes. The verification shows that, upon successful proof checking by *CakePB*, the conclusions of the proof log are sound with respect to the semantics of the input decision or optimization problem. Formally verified compilation [67] extends this guarantee all the way to the *CakePB* machine code (see [39, Section 3.3] for more details).

The *CakePB* updates include implementing and verifying standard code optimizations such as introducing tail recursion and carefully choosing data structures, but below we discuss some more interesting changes specific to pseudo-Boolean proof checking.

1. **RUP with annotations:** In prior work [49], *VeriPB* elaborated RUP steps into cutting planes proofs to be checked by *CakePB*. This meant that *VeriPB* expanded RUP statements to cutting planes derivations, which *CakePB* then checked using its general procedure for cutting planes. A much more efficient approach is for *CakePB* to support annotated RUP steps, which include a list of constraints in the order they propagate. The verified RUP implementation in *CakePB* tracks these propagations with a mutable bitset, which allows units to be calculated in linear passes over the annotated constraints.
2. **Simplifications for cutting planes proofs:** *CakePB* now makes a syntactic simplification pass on `pol` lines, where adjacent literal weakening steps are merged into a single simultaneous step internally, and adjacent additions of literal axioms (possibly multiplied by constants) are also merged. In this way *CakePB* can use a single pass over the resulting merged step to carry out all these operations.
3. **Space-efficient occurrence mapping:** *CakePB* tracks a mapping from variables to the IDs of all constraints containing them [49]; this is used as an efficient heuristic for skipping proof obligations in strengthening rules. However, such a mapping is space-inefficient as many of the variables are never looked up. Our new heuristic instead tracks only a fixed number of constraints per variable. We also changed the mapping to use an in-place array-based representation as opposed to the previous, purely functional implementation.

6 Experiments

We have run experiments on the 397 decision instances and 478 optimization instances in the Pseudo-Boolean Competition 2024 [62] on hardware with i5-1145G7 CPUs, 14GB of available RAM, a 100GB solid state drive as storage, and Rocky Linux 8.10 as operating system. We first ran the solvers without proof logging to establish a baseline, and only kept instances solved within a one-hour time limit. We then ran solvers with proof logging, had *VeriPB* elaborate the proofs, and fed them to the formally verified checker *CakePB*. Noise due to very small running times was smoothed by computing all ratios as $(1 + x)/(1 + y)$.

Sat4j Cutting Planes (Figure 1) solves 199 decision and 123 optimization instances. The worst overhead for proof generation is 51.7%; 95% of the instances have an overhead

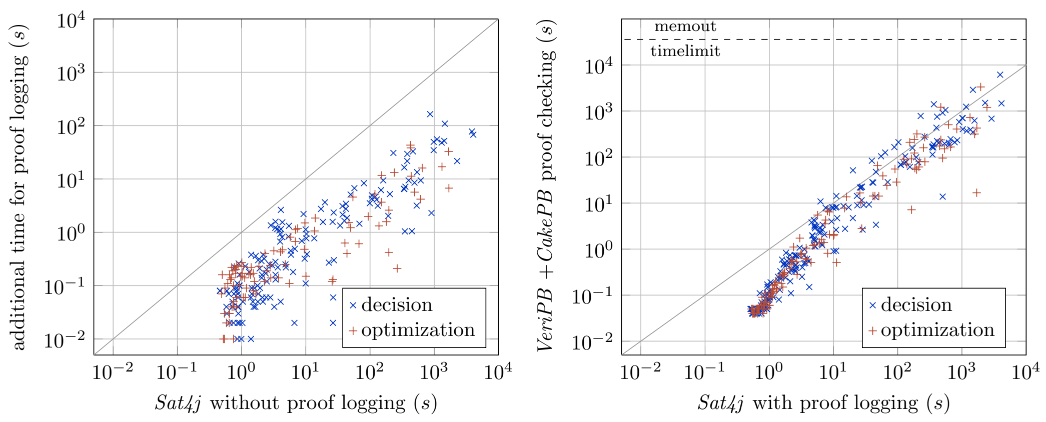

Figure 1 *Sat4j Cutting Planes* overhead for proof logging and logging versus checking time.

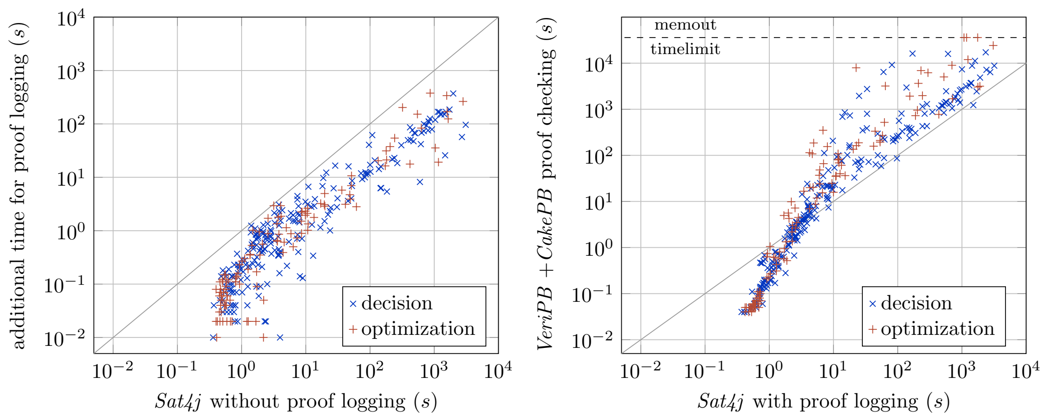

Figure 2 *Sat4j Resolution* overhead for proof logging and logging versus checking time.

below 14.2%; and the median is 2.6%. All instances are successfully checked within 6,158s. The worst ratio of checking to solving time is 3.83; 95% of the instances are checked within a factor 1.60, and the median is a factor 0.54, which is even faster than solving.

Sat4j Resolution (Figure 2) solves 243 decision and 123 optimization instances. The worst proof logging overhead is 71%; 95% of the instances have an overhead below 42%; and the median is 9.7%. For 3 instances, *VeriPB* times out after 10h. The worst ratio of proof checking time compared to solving time is a whopping 338. However, 95% of the instances are checked within a factor 20.9; and the median is a factor 1.33. The main reason for the much larger overheads for checking is that *Sat4j Resolution* essentially writes SAT-style *DRAT* proofs using RUP steps without annotations. This means that *VeriPB* has to perform propagation for every clause learned, but pseudo-Boolean propagation is more expensive than clausal propagation and *VeriPB* has no special handling of clausal proofs.

RoundingSat (Figure 3) solves 297 decision and 258 optimization instances. The worst proof logging overhead is 46.2%; 95% of the instances have an overhead below 21.1%; and the median is 2.7%. One optimization instance cannot be elaborated by *VeriPB* because the proofs exceed the 100GB of available disk space. On 3 decision and 4 optimization instances *VeriPB* runs out of memory, and on a further 1 decision and 1 optimization instance *CakePB* does. All remaining instances are checked within 11,730s. The worst ratio of checking to

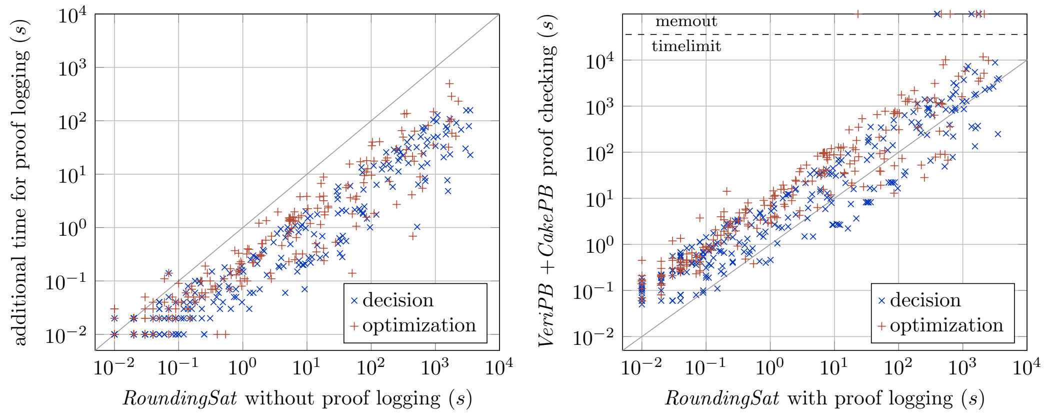

Figure 3 *RoundingSat* overhead for proof logging and logging versus checking time.

solving time is 19.17; 95% of the instances are checked within a factor 9.22; and the median is a factor 1.43. This is arguably quite close to what would be expected of SAT proof logging.

7 Concluding Remarks

In this work, we present a practically feasible and fully formally verified toolchain based on *VeriPB* and *CakePB* for certified pseudo-Boolean solving and optimization. Our work covers the full range of techniques in the state-of-the-art solvers *RoundingSat* and *Sat4j*, and so should be eminently possible to adapt to other pseudo-Boolean solvers.

From a solver author perspective, however, a key concern for adoption of proof logging is ease of use. One of the lessons learned from our work is that proof logging for basic constraint simplifications such as removing fixed variables is both common and surprisingly tricky to implement correctly and efficiently. Further investigations are warranted into how such simplifications could be handled more smoothly in the proof system. Also, adding a proof rule for mixed integer rounding (MIR) cuts would greatly simplify the proof logging for solvers that employ MIR cut generation techniques from mixed integer programming.

Regarding *VeriPB* performance, a substantial amount of time is spent on writing and parsing files. Introducing a binary file format for pseudo-Boolean formulas and proofs would alleviate this issue. Checking reverse unit propagation (RUP) steps is another bottleneck. For *Sat4j Resolution* the obvious remedy would be to implement RUP with annotations. In general, more sophisticated propagation algorithms [24, 59] would probably be helpful. However, for SAT-like proofs with *DRAT*-style, non-annotated, RUP steps, implementing a dedicated clausal checker along the lines of [71] inside *VeriPB* appears to be the best way to get truly efficient proof checking, but this seems like an engineering rather than a research problem.

For verified proof checking with *CakePB*, working on PB solving has led to several improvements on top of earlier optimizations for subgraph solving [39] and MaxSAT preprocessing [49]. Profiling shows the potential for further efficiency gains in checking strengthening steps and cutting planes derivations. Parallelizing the proof elaboration by *VeriPB* and checking by *CakePB* also seems worth exploring, especially since both tools support streaming proof files.

To the best of our knowledge, we present the first certified solving toolchain for an optimization problem beyond SAT with performance approaching that of SAT proof logging tools. We are hopeful that our work could point the way towards practically feasible certified solving also for stronger paradigms such as constraint programming and mixed integer programming.

References

1 Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, August 2018.

2 Katherine I. Albanese, Sophie Barbe, Shunsuke Tagami, Derek N. Woolfson, and Thomas Schiex. Computational protein design. *Nature Reviews Methods Primers*, 5(13), February 2025.

3 Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. Carcara: An efficient proof checker and elaborator for SMT proofs in the alethe format. In *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '23)*, volume 13993 of *Lecture Notes in Computer Science*, pages 367–386. Springer, April 2023.

4 Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark Barrett. Flexible proof production in an industrial-strength SMT solver. In *Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR '22)*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, August 2022.

5 Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesande. Certifying without loss of generality reasoning in solution-improving maximum satisfiability. 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 4:1–4:28, September 2024.

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 Daniel Le Berre and Stéphanie Roussel. Sat4j 2.3.2: on the fly solver configuration: System description. *Journal on Satisfiability, Boolean Modelling and Computation*, 8(3/4):197–202, August 2012.

8 Frédéric Besson, Pascal Fontaine, and Laurent Théry. A flexible proof format for SMT: a proposal. In *Proceedings of the 1st Workshop on Proof eXchange for Theorem Proving (PxTP '11)*, pages 15–26, August 2011.

9 Armin Biere. Tracecheck. `http://fmv.jku.at/tracecheck/`, 2006.

10 Armin Biere and Daniel Kröning. SAT-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, chapter 10, pages 277–303. Springer, December 2018.

11 Péter Biró, Joris van de Klundert, David F. Manlove, William Pettersson, Tommy Andersson, Lisa Burnapp, Pavel Chromy, Pablo Delgado, Piotr Dworczak, Bernadette Haase, Aline Hemke, Rachel Johnson, Xenia Klimentova, Dirk Kuypers, Alessandro Nanni Costa, Bart Smeulders, Frits C. R. Spieksma, María O. Valentín, and Ana Viana. Modelling and optimisation in European kidney exchange programmes. *European Journal of Operational Research*, 291(2):447–456, June 2021.

12 Péter Biró, Bernadette Haase-Kromwijk, Tommy Andersson, Eyjólfur Ingi Ásgeirsson, Tatiana Baltesová, Ioannis Boletis, Catarina Bolotinha, Gregor Bond, Georg Böhmig, Lisa Burnapp, Katarína Cechlárová, Paola Di Ciaccio, Jiri Fronek, Karine Hadaya, Aline Hemke, Christian Jacquelinet, Rachel Johnson, Rafal Kieszek, Dirk R. Kuypers, Ruthanne Leishman, Marie-Alice Macher, David Manlove, Georgia Menoudakou, Mikko Salonen, Bart Smeulders, Vito Sparacino, Frits C. R. Spieksma, María Oliva Valentín, Nic Wilson, and Joris van der Klundert. Building kidney exchange programmes in Europe—an overview of exchange practice and activities. *Transplantation*, 103(7):1514–1522, June 2019.

13 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*.

14 Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT '09)*, pages 1–5, August 2009.

15 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, 2023.

16 Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In 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*, chapter 7, pages 233–350. IOS Press, 2nd edition, February 2021.

17 Florent Capelli. Knowledge compilation languages as proof systems. In Mikolás Janota and Inês Lynce, editors, *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, volume 11628 of *Lecture Notes in Computer Science*, pages 90–99. Springer, 2019.

18 Kevin K. H. Cheung, Ambros M. Gleixner, and Daniel E. Steffy. Verifying integer programming results. In *Proceedings of the 19th International Conference on Integer Programming and Combinatorial Optimization (IPCO '17)*, volume 10328 of *Lecture Notes in Computer Science*, pages 148–160. Springer, June 2017.

19 William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

20 William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

21 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, August 2017.

22 Luís Cruz-Filipe, João P. Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, volume 10205 of *Lecture Notes in Computer Science*, pages 118–135. Springer, April 2017.

23 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.

24 Jo Devriendt. Watched propagation of 0-1 integer linear constraints. 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 160–176. Springer, September 2020.

25 Jo Devriendt, Ambros Gleixner, and Jakob Nordström. Learn to relax: Integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search. *Constraints*, 26(1–4):26–55, October 2021. Preliminary version in *CPAIOR '20*.

26 Jo Devriendt, Stephan Gocht, Emir Demirović, Jakob Nordström, and Peter Stuckey. Cutting to the core of pseudo-Boolean optimization: Combining core-guided search with cutting planes reasoning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3750–3758, February 2021.

27 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)*, November 2025. To appear.

28 Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining alldifferent. In *Proceedings of the 35th Australasian Computer Science Conference (ACSC '12)*, pages 115–124, January 2012.

29 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03), Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

30 Leon Eifler and Ambros Gleixner. A computational status update for exact rational mixed integer programming. *Mathematical Programming*, 197(2):793–812, February 2023.

31 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.

32 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, pages 1291–1299, July 2018.

33 Salomé Eriksson, Gabriele Röger, and Malte Helmert. Unsolvability certificates for classical planning. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS '17)*, pages 88–97, June 2017.

34 Salomé Eriksson, Gabriele Röger, and Malte Helmert. A proof system for unsolvable planning tasks. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS '18)*, pages 65–73, June 2018.

35 Katalin Fazekas, Florian Pollitt, Mathias Fleury, and Armin Biere. Certifying incremental SAT solving. In *Proceedings of the 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2024)*, volume 100 of *EPiC Series in Computing*, pages 321–340, May 2024.

36 Johannes Klaus Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. 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 30:1–30:24, August 2022.

37 Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, October 2019.

38 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.

39 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.

40 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.

41 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.

42 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.

43 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.

44 Mark A. Hallen and Bruce Randall Donald. Protein design by provable algorithms. *Communications of the ACM*, 62(10):76–84, October 2019.

45 Peter L. Hammer and Sergiu Rudeanu. *Boolean Methods in Operations Research and Related Areas*. Springer Verlag, New York, 1968.

46 Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pages 181–188, October 2013.

47 Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, June 2013.

48 Jochen Hoenicke and Tanja Schindler. A simple proof format for SMT. In *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories (SMT '22)*, volume 3185 of *CEUR Workshop Proceedings*, pages 54–70, August 2022.

49 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.

50 Sonja Kraiczy and Ciaran McCreesh. Solving graph homomorphism and subgraph isomorphism problems faster through clique neighbourhood constraints. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI '21)*, pages 1396–1402, August 2021.

51 Peter Lammich. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, 64(3):513–532, March 2020. Extended version of paper in *CADE* 2017.

52 Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, July 2010.

53 Hugues Marchand and Laurence A. Wolsey. Aggregation and mixed integer rounding to solve MIPs. *Operations Research*, 49(3):325–468, June 2001.

54 Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24:165–203, February 2000.

55 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.

56 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.

57 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.

58 Esther Mugdan, Remo Christen, and Salomé Eriksson. Optimality certificates for classical planning. In *Proceedings of the 33rd International Conference on Automated Planning and Scheduling (ICAPS '23)*, pages 286–294, July 2023.

59 Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Rui Zhao. Speeding up pseudo-Boolean propagation. In *Proceedings of the 27th International Conference on Theory*

and Applications of Satisfiability Testing (SAT '24), volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:18, August 2024.

60 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, January 2009.

61 Florian Pollitt, Mathias Fleury, and Armin Biere. Faster LRAT checking than solving with CaDiCaL. 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 21:1–21:12, July 2023.

62 Pseudo-Boolean competition 2024. `https://www.cril.univ-artois.fr/PB24/`, August 2024.

63 The International SAT Competitions web page. `https://satcompetition.github.io`.

64 Dominik Schreiber. Lilotane: A lifted SAT-based approach to hierarchical planning. *Journal of Artificial Intelligence Research*, 70:1117–1181, March 2021.

65 Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In *Proceedings of the 7th Workshop on Proof eXchange for Theorem Proving (PxTP '21)*, volume 336 of *Electronic Proceedings in Theoretical Computer Science*, pages 49–54, July 2021.

66 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in CakeML. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12652 of *Lecture Notes in Computer Science*, pages 223–241. Springer, March-April 2021.

67 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.

68 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *10th International Symposium on Artificial Intelligence and Mathematics (ISAIM '08)*, 2008. Available at `http://isaim2008.unl.edu/index.php?page=proceedings`.

69 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.

70 VeriPB: Verifier for pseudo-Boolean proofs. `https://gitlab.com/MIAOresearch/software/VeriPB`.

71 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.

Algorithm 1 MIR cut generation in *RoundingSat*.

```
   input : Set of indices $J$, list $\vec{C}$ of constraints $C_j \doteq \sum_j a_{ij}x_j \geq A_j$ for $j \in J$,
           list $\vec{\lambda}$ of multipliers $\lambda_j \in \mathbb{Z}$ for $j \in J$, divisor $d \in \mathbb{N}^+$, variable set $P$
 1 CutGeneration$(J, \vec{C}, \vec{\lambda}, d, P)$ :
 2 for $j \in J$ do
 3     define $C'_j \doteq \sum_j a_{ij}x_j - s_j = A_j$ ;          // Introduce slack variables
 4 $D \leftarrow \sum_j \lambda_j C'_j$;
 5 partially weaken $x_i$ for all $i \in P$;
 6 $B \leftarrow \mathsf{RHS}(D)$;
 7 $D \leftarrow \mathsf{MIR}(D, d)$;
 8 for $j \in J$ do
 9     add $M(-\lambda_j, d, B \bmod d)$ times $C'_j$ to $D$ ;       // Cancel slack variables
10 return $D$;
```

A Proof Logging for MIR Cut Generation

We start by analyzing the cut computed by *RoundingSat*, as summarized in Algorithm 1. Let inequalities $C_j \doteq \sum_i a_{ij}x_i \geq A_j$, multipliers λ_j, and a divisor d be given. We introduce an integer-valued *integral slack variable* $s_j \geq 0$ for each constraint C_j to get the equality $C'_j \doteq \sum_i a_{ij}x_i - s_j = A_j$. The constraint $D \doteq \sum_j \lambda_j C'_j$ can then, after grouping terms by variable, be written as

$$\sum\nolimits_i \left(\sum\nolimits_j \lambda_j a_{ij}\right) x_i - \sum\nolimits_j \lambda_j s_j = \sum\nolimits_j \lambda_j A_j. \tag{65}$$

Write $b'_i = \sum_j \lambda_j a_{ij}$ and let P be the set of variables on which we *partially weaken* (which means that we add literal axioms to reduce the coefficients of the variables in P to the largest smaller multiple of the divisor d). Partially weakening on the variables in P yields

$$\sum\nolimits_{i \notin P} b'_i x_i + \sum\nolimits_{i \in P} d\left\lfloor \frac{b'_i}{d} \right\rfloor x_i + \sum\nolimits_j (-\lambda_j)s_j = \sum\nolimits_j \lambda_j A_j - \sum\nolimits_{i \in P}(b'_i \bmod d). \tag{66}$$

Write $B = \sum_j \lambda_j A_j - \sum_{i \in P}(b'_i \bmod d)$, and let $b_i = b'_i$ for $i \notin P$ and $b_i = d\lfloor b'_i/d \rfloor$ for $i \in P$. Next, we apply a MIR cut with divisor d. Write $H = B \bmod d$, and write $M(a, d, H) = \lfloor a/d \rfloor H + \min(a \bmod d, H)$. This yields:

$$\sum\nolimits_i M(b_i, d, H)x_i + \sum\nolimits_j M(-\lambda_j, d, H)s_j \geq \lceil B/d \rceil H. \tag{67}$$

We now cancel the integral slack variables by adding

$$\sum\nolimits_j M(-\lambda_j, d, H) \left(\sum\nolimits_i a_{ij}x_i - s_j\right) = \sum\nolimits_j M(-\lambda_j, d, H)A_j. \tag{68}$$

Write $f_i = \sum_j M(-\lambda_j, d, H)a_{ij}$ and $F = \sum_j M(-\lambda_j, d, H)A_j$. Then this yields

$$\sum\nolimits_i \left(M(b_i, d, H) + f_i\right)x_i \geq \lceil B/d \rceil H + F, \tag{69}$$

which is the final cut derived by *RoundingSat* and returned by Algorithm 1.

We note that the cut only depends on the values of the multipliers modulo d. Hence, we may assume without loss of generality that $0 \leq \lambda_j \leq d - 1$ for all j. We now show how to proof log the cut. We prove (69) by contradiction. The negation of (69) is given by

$$\sum\nolimits_i \left(-M(b_i, d, H) - f_i\right)x_i \geq -\lceil B/d \rceil H - F + 1. \tag{70}$$

Write $\delta = d - H$ and $\mu_j = -M(-\lambda_j, d, H)$. We now want to add constraint C_j with multiplier $\lambda_j - \mu_j - \delta \mathbf{1}_{\mu_j>0}$, which we show to be nonnegative in Lemma 1. Intuitively, the $-\mu_j$ cancels out the f_i and F in (70), while we need the $-\delta \mathbf{1}_{\mu_j>0}$ later on to cancel the f_i again.

▶ **Lemma 1.** *We have* $\lambda_j - \mu_j - \delta \mathbf{1}_{\mu_j>0} \geq 0$.

Proof. If $\lambda_j = 0$, then $\mu_j = 0$ and hence the claim follows.

Otherwise, we have $0 \leq d - \lambda_j \leq d - 1$, and hence $(-\lambda_j) \bmod d = d - \lambda_j$.

If $0 < \lambda_j \leq \delta$, then $d - \lambda_j \geq H$, so $\mu_j = M(-\lambda_j, d, H) = -H + \min(d - \lambda_j, H) = 0$ and hence $\lambda_j - \mu_j - (d - H)\mathbf{1}_{\mu_j>0} = \lambda_j \geq 0$, as required.

If $\lambda_j > \delta$, we have $M(-\lambda_j, d, H) = -H + \min(d - \lambda_j, H) = -H + d - \lambda_j > 0$. Then we have $\lambda_j - \mu_j - \delta \mathbf{1}_{\mu_j>0} = \lambda_j - H + d - \lambda_j - (d - H) = 0$, as required. ◀

Let $T = \{j : \mu_j > 0\}$. Since $\sum_j \lambda_j C_j \doteq \sum_i \left(\sum_j \lambda_j a_{ij}\right) x_i \geq \sum_j \lambda_j A_j \doteq \sum_i b'_i x_i \geq \sum_j \lambda_j A_j$ and similarly $-\sum_j \mu_j C_j \doteq \sum_i \left(\sum_j (-\mu_j) a_{ij}\right) x_i \geq \sum_j (-\mu_j) A_j \doteq \sum_i f_i x_i \geq F$, we can write the constraint $\sum_j (\lambda_j - \mu_j - \delta \mathbf{1}_{\mu_j>0}) C_j$ as

$$\sum_i (b'_i + f_i) x_i - \delta \sum_{j \in T} \sum_i a_{ij} x_i \geq \sum_j \lambda_j A_j + F - \delta \sum_{j \in T} A_j. \tag{71}$$

We partially weaken this on variables in P by adding $\sum_{i \in P} (b'_i \bmod d) \overline{x}_i \geq 0$, to obtain

$$\sum_i (b_i + f_i) x_i - \delta \sum_{j \in T} \sum_i a_{ij} x_i \geq B + F - \delta \sum_{j \in T} A_j. \tag{72}$$

Hence, adding this to Equation (70) yields

$$\sum_i \left(b_i - M(b_i, d, H)\right) x_i - \delta \sum_{j \in T} \sum_i a_{ij} x_i \geq B - \lceil B/d \rceil H + 1 - \delta \sum_{j \in T} A_j. \tag{73}$$

To analyze this, we write $b_i = g_i d + h_i$ with $0 \leq h_i \leq d - 1$. Note that $M(g_i d + h_i, d, H) = g_i H + \min(h_i, H)$. Also, we have $B - \lceil B/d \rceil H = Gd + H - (G+1)H = G(d - H) = G\delta$. Hence, we can write Equation (73) as

$$\sum_i \left(g_i \delta + h_i - \min(h_i, H)\right) x_i - \delta \sum_{j \in T} \sum_i a_{ij} x_i \geq G\delta + 1 - \delta \sum_{j \in T} A_j. \tag{74}$$

We now apply the division rule with divisor $\delta = d - H$ (which is possible since $H < d$). To compute the result, write $S = \{i : h_i > H\}$. If $i \in S$, then

$$g_i \delta < g_i \delta + h_i - \min(h_i, H) < (g_i + 1)\delta,$$

since $h_i < d$. Hence, $\left\lceil \frac{g_i \delta + h_i - \min(h_i, H)}{\delta} \right\rceil = g_i + 1$. On the other hand, if $i \notin S$, then $h_i - \min(h_i, H) = 0$ so $\left\lceil \frac{g_i \delta + h_i - \min(h_i, H)}{\delta} \right\rceil = g_i$. Hence, the result is

$$\sum_i g_i x_i + \sum_{i \in S} x_i - \sum_{j \in T} \sum_i a_{ij} x_i \geq G + 1 - \sum_{j \in T} A_j. \tag{75}$$

Here, we use that all coefficients in $-\delta \sum_{j \in T} \sum_i a_{ij} x_i$ are divisible by δ, and hence do not affect the division of the x_i, even though the variables overlap. However, this division potentially not in normalized form. Nevertheless, any division which is not in normalized form can be simulated in *VeriPB* by applying partial weakening before dividing.

We now proceed by showing how we can combine Equation (70) with Equation (75) to derive a contradiction. In the newly introduced notation, we can write Equation (70) as

$$-\sum_i \left(g_i H + \min(h_i, H)\right) x_i - \sum_i f_i x_i \geq -(G+1)H - F + 1, \tag{76}$$

where we have also rewritten the terms involving the f_i in variable form.

Multiplying Equation (75) with H and adding it to Equation (76) yields

$$-\sum_{i \notin S} h_i x_i + \sum_i \left(-f_i - H \sum_{j \in T} a_{ij}\right) x_i \geq -F - H \sum_{j \in T} A_j + 1. \tag{77}$$

Note that we can write $f_i = -\sum_j a_{ij}\mu_j$ and $F = -\sum_j A_j \mu_j$, which yields

$$-\sum_{i \notin S} h_i x_i + \sum_i \sum_{j \in T} (\mu_j - H) a_{ij} x_i \geq \sum_{j \in T} (\mu_j - H) A_j + 1. \tag{78}$$

Since $\mu_j \leq H$, we have $H - \mu_j \geq 0$. Hence, we can add C_j with multiplier $H - \mu_j \geq 0$ for all j with $\mu_j > 0$ to Equation (78), which yields $-\sum_{i \notin S} h_i x_i \geq 1$. Since $h_i \geq 0$, this yields a contradiction by adding literal axioms $x_i \geq 0$.

B Core-Guided PB Logging

In this section use X_S as a shortcut to denote $\sum_S x_i$. We do not normalise constraints. We assume without loss of generality that all weights and literals in the objective are positive, so we can write $f_o \doteq \sum w_i x_i$.

During optimization we repeatedly call the solver with assumptions. Learned constraints are logged in the standard way. The solver might return in either of two states. If the solver found a solution, we log it using the `soli` rule and obtain a solution-improving constraint $f_o \leq u - 1$. Otherwise the solver finds the assumptions are contradictory. We process the contradiction into a cardinality constraint $\sum_{i \in S} x_i \geq d$, or $X_S \geq d$ for short, which we log and call the "core lower bound" constraint. We use this to reformulate the objective.

▶ **Lemma 2.** *The core lower bound is derivable with standard conflict analysis.*

To do this we introduce counting variables y_j for $j \in [0, |S|]$ with intended semantics $y_j = [\![X_S \geq j]\!]$. Then we can reformulate the objective to $f_r \doteq f_o - w\Delta_K$ where $\Delta_K \doteq X_S - Y$. Observe that $\Delta_K = 0$, hence the objective does not change. We call a triple (S, d, w) a core, and denote the set of all cores by $\mathcal{K}$.

Taking all cores into account, we have $\Delta \doteq \sum_{K \in \mathcal{K}} \Delta_K = 0$, which we can use to log objective lower bounds $f_o \geq \ell$ and to translate solution-improving constraints $f_o \leq u$ for the original objective into solution-improving constraints $f_r \leq u$ for the reformulated objective. Observe that we only use the fact that the reformulated objective lower bounds the original objective, in other words $f_r \leq f_o$ or $\Delta \geq 0$. We call such constraint the "global reformulation constraint", and we discuss how to derive it later.

In principle we might have $|S| + 1$ counting variables, but fewer variables are enough. Since $X_S \geq d$, we already know that $y_i = 1$ for $i \leq d$. Similarly, if we have an upper bound on the core of the form $u \geq X_S$, then we also know that $Y_{>u} = 0$, i.e., $y_i = 0$ for $i > u$.

We always have a trivial upper bound of the form $|S| \geq X_S$, but sometimes we can obtain a better bound from the solution-improving constraint. Let the "objective-to-core" constraint be $f_o - wX_S \geq 0$. If we have $f_o \leq u$, we have $X_S \leq \lfloor u/w \rfloor$. If $\lfloor u/w \rfloor < |S|$ then we log this constraint as the "core upper bound" constraint.

▶ **Lemma 3.** *The objective-to-core constraint is derivable from the global reformulation constraint.*

Proof. We have $f_r - wX_S \geq 0$ trivially. We add $\Delta \geq 0$ and obtain $f_o - wX_S \geq 0$. ◀

It is worth mentioning that, as an optimization, we delay logging this constraint until the time it is needed, if at all. This means that the reformulated objective may have changed in the meantime, and the actual way in which we log the objective-to-core constraint is by marking it as syntactically implied from $\Delta \geq 0$.

▶ **Lemma 4.** *The core upper bound is derivable from the solution-improving constraint and the objective-to-core constraint.*

Proof. We add $f_o \leq u$ and $f_o - wX_S \geq 0$ and obtain $wX_S \leq u$. Then we divide by w. ◀

Another use of the global reformulation constraint is hardening. Given the objective improving constraint $f_r \leq u$, any literal appearing in f_r with coefficient larger than u can be fixed to 0. We log such constraints by RUP.

How to proceed depends on how we encode $y_j = [\![X_S \geq j]\!]$.

B.1 Eager Encoding

With an eager encoding we reformulate the objective to $f_r \doteq f_o - w\Delta_K$ where $\Delta_K \doteq X_S - Y \doteq X_S - Y_{[d+1,u]} - d$. We need to derive an upper and a lower bound on Δ_K.

▶ **Lemma 5.** *The at-most constraint $\Delta_K \leq 0$ is derivable from the core upper bound.*

Proof. We derive $Y_{[d+1,u]} + d \geq X_S$ by redundance with witness $Y_{[d+1,u]} \mapsto 1$. Previously derived constraints are syntactically untouched by the witness and do not need proving. The conclusion becomes $u \geq X_S$, which is syntactically identical to the core upper bound. ◀

▶ **Lemma 6.** *The at-least constraint $\Delta_K \geq 0$ is derivable from the core lower bound.*

Proof. For $j = d+1, \dots, u$, we derive $X_S \geq Y_{[d+1,j]} + d$ by redundance with witness $y_j \mapsto 0$. Previous constraints are untouched. The at-most constraint becomes $Y_{[d+i,u]\setminus\{j\}} + d \geq X_S$, which is a weakening of the negated conclusion $Y_{[d+1,j]} + d - 1 \geq X_S$. Intermediate at-least constraints are untouched. The conclusion becomes $X_S \geq Y_{[d+1,j)} + d$, which is either syntactically identical to the core lower bound if $j = d+1$, or to the previous intermediate at-least constraint otherwise.

We delete all intermediate at-least constraints at the end. ◀

Note that we do not use a single application of the redundance rule with witness $Y_{[d+1,u]} \mapsto 0$ because the negation of the at-least constraint does not necessarily imply the at-most constraint after applying the witness substitution.

▶ **Lemma 7.** *The ordering constraints $y_j \geq y_{j+1}$ are derivable.*

Proof. For $j = d+1, \dots, u-1$ we derive $y_j \geq y_{j+1}$ by redundance with witness $y_{j+1} \mapsto y_j \mapsto y_{j-1} \mapsto \cdots \mapsto y_{d+1} \mapsto y_{j+1}$. Its negation propagates $y_j = 0$ and $y_{j+1} = 1$. The at-least and at-most constraints are mapped to themselves. $y_{d+1} \geq y_{d+2}$ is mapped to $y_{j+1} \geq y_{d+1}$, which is satisfied by unit propagation. Other previous ordering constraints are mapped to (different) previous ordering constraints. The conclusion is also mapped to a previous ordering constraint. ◀

B.2 Reified Encoding

Since introducing many variables at once may slow down the solver, an alternative to an eager encoding is to use a lazy encoding, where we introduce counting variables one by one on demand.

That is, instead of Δ_K, we subtract $\Delta_K^{(j)} \doteq X_S - Y_{[d+1,d+j]} - d$ from the objective. Observe that $\Delta_K^{(j)} \geq 0$, hence the reformulated objective still lower bounds the original objective, even though they might no longer be equal.

We begin by introducing a new variable y_{d+1}, and two constraints $y_{d+1} \rightarrow [\![X_S \geq d+1]\!]$ and $[\![X_S \geq d+1]\!] \rightarrow y_{d+1}$. As linear constraints, these correspond to $X_S \geq y_{d+1} + d$ ("at least") and $X_S \leq (u-d)y_{d+1} + d$ ("at most") respectively.

If we remove the counting variable from the objective during a subsequent objective reformulation, then the core would become useless and we would not be able to prove an exact lower bound on the objective. Therefore, in that case, we introduce a new counting variable and the relevant constraints, namely $X_S \geq jy_{d+j} + d$ and $X_S \leq (u-d-j+1)y_{d+j} + d + j - 1$. We write $\xi = (u-d-j+1)$.

▶ **Lemma 8.** *The at-most constraint is derivable from the core upper bound.*

Proof. We derive $\xi y_{d+j} - X_S \geq -d - j + 1$ by redundance with witness $y_{d+j} \mapsto 1$. Previous constraints are untouched. The conclusion becomes $u - X_S \geq 0$, which is trivial for $u = |S|$ and syntactially identical to the core upper bound otherwise. ◀

▶ **Lemma 9.** *The at-least constraint is derivable from the core lower bound.*

Proof. We derive $X_S - jy_{d+j} \geq d$ by redundance with witness $y_{d+j} \mapsto 0$. Previous constraints are untouched. The at-most constraint becomes $-X_S \geq -d - j + 1$, which is a weakening of the negated conclusion $-X_S + jy_{d+1} \geq -d + 1$. The conclusion becomes $X_S \geq d$, which is syntactically identical to the core lower bound. ◀

▶ **Lemma 10.** *The ordering constraint $y_j \geq y_{j+1}$ is derivable from the at-least and previous at-most constraints.*

Proof. Add the at-least and the previous at-most constraints. Then divide by a large enough number such as $|S|$. ◀

The local reformulation constraint changes each time that we introduce a variable. The first constraint is $\Delta_K^{(1)} \doteq X_S - y_{d+1} - d \geq 0$, which is syntactically identical to the at-least constraint. Subsequent reformulation constraints need to be derived.

▶ **Lemma 11.** *The local reformulation constraint $\Delta_K^{(j)}$ is derivable from the at-most constraint and $\Delta_K^{(j-1)}$.*

Proof. Take a linear combination of $(j-1)\Delta_K^{(j-1)}$ and the at-most constraint. Then divide by j. ◀

Observe that on a lazy encoding, bounds on the core need not stay constant. If we find better solutions, we may end up with better upper bounds. In that case we need to rederive the core upper bound. Note that the objective might have changed in the meantime, but the objective-to-core constraint already takes that into account by using the reformulation constraint from the time the core was found.

In general, a better core upper bound will be taken automatically into account when a new variable is introduced. However, it may be that the new upper bound makes introducing new variables unnecessary. In that case we have $d + j = u$, and we may improve the last at-most constraint to $X_S \leq y_{d+j} + d + j - 1 = y_{d+j} + u - 1$ instead.

▶ **Lemma 12.** *The improved at-most constraint is derivable from the core upper bound and the at-most constraint.*

Proof. Take a linear combination of $(\xi - 1)(X_S \leq u)$ and the at-most constraint. Then divide by ξ. ◀

B.3 Lazy Sum Encoding

The lazy sum encoding follows the same principles as the reified encoding, but we use stronger at-most and at-least constraints for subsequent counter variables. Instead of $X_S \geq jy_{d+j} + d$ we have $X_S \geq Y_{[d+1,d+j]} + d$, and instead of $X_S \leq (u - d - j + 1)y_{d+j} + d + j - 1$ we have $X_S \leq (u - d - j + 1)y_{d+j} + Y_{[d+1,d+j-1]} + d$. Observe that we can derive the reified at-least constraint as a linear combination of its lazy sum counterpart and ordering constraints, and that the reified at-most constraint is a weakening of the lazy-sum at-most constraint.

The first at-least and at-most constraints are derived identically to the reified case. Subsequent constraints are derived as follows.

▶ **Lemma 13.** *The at-most constraint is derivable from the previous at-most constraint.*

Proof. We derive $\xi y_{d+j} + Y_{[d+1,d+j-1]} - X_S \geq -d$ by redundance with witness $y_{d+j} \mapsto y_{d+j-1}$. Previous constraints are untouched. The conclusion becomes syntactically identical to the previous at-most constraint. ◀

▶ **Lemma 14.** *The at-least constraint is derivable from the previous at-least constraint.*

Proof. We derive $X_S - Y_{d+1,d+j} \geq d$ by redundance with witness $y_{d+j} \mapsto 0$. Previous constraints are untouched. The at-most constraint becomes $Y_{[d+1,d+j-1]} - X_S \geq -d$, which is a weakening of the negated conclusion $-X_S + Y_{d+1,d+j} \geq -d + 1$. The conclusion becomes syntactically identical to the previous at-least constraint. ◀

The ordering constraint $y_j \geq y_{j+1}$ is derivable from the at-least and previous at-most constraints identically to the reified case. The local reformulation constraint is always identical to the at-least constraint and does not need to be derived.

We need to handle updated core upper bounds in a different way since logging the at-most constraint depends on the previous at-most constraint, which is based on a different core upper bound. What we do instead is to first derive an improvement of the previous at-most constraint, and then log the current at-most constraint as usual.

▶ **Lemma 15.** *The improved at-most constraint is derivable from the core upper bound, the ordering constraints, and the at-most constraint.*

Proof. Let d be the core lower bound. Let $u = \xi + d$ be the previous core upper bound (so ξ is the previous expected number of counting variables). Let $u' = \eta + d$ be the new core upper bound (so η is the new expected number of counting variables).

We derive $X_S \leq (\eta - j + 1)y_{d+j} + Y_{[d+1,d+j-1]} + d$ by redundance with an empty witness. We add the current at-most constraint and the negation of the conclusion to obtain $(u - u')y_{d+j} \geq 1$, which we saturate to $y_{d+j} \geq 1$. The ordering constraints then propagate $y_{d+1} = \cdots = y_{d+j} = 1$. We take a linear combination of these and $X_S \leq u$ to obtain the conclusion. ◀