Certified CNF Translations for Pseudo-Boolean Solving

Stephan Gocht
Lund University, Lund, Sweden
University of Copenhagen, Copenhagen, Denmark

Ruben Martins
Carnegie Mellon University, Pittsburgh, Pennsylvania, USA

Jakob Nordström
University of Copenhagen, Copenhagen, Denmark
Lund University, Lund, Sweden

Andy Oertel
Lund University, Lund, Sweden
University of Copenhagen, Copenhagen, Denmark

Abstract

The dramatic improvements in Boolean satisfiability (SAT) solving since the turn of the millennium have made it possible to leverage state-of-the-art conflict-driven clause learning (CDCL) solvers for many combinatorial problems in academia and industry, and the use of proof logging has played a crucial role in increasing the confidence that the results these solvers produce are correct. However, the conjunctive normal form (CNF) format used for SAT proof logging means that it has not been possible to extend guarantees of correctness to the use of SAT solvers for more expressive combinatorial paradigms, where the first step is to translate the input to CNF.

In this work, we show how cutting-planes-based reasoning can provide proof logging for solvers that translate pseudo-Boolean (a.k.a. 0-1 integer linear) decision problems to CNF and then run CDCL. To support a wide range of encodings, we provide a uniform and easily extensible framework for proof logging of CNF translations. We are hopeful that this is just a first step towards providing a unified proof logging approach that will also extend to maximum satisfiability (MaxSAT) solving and pseudo-Boolean optimization in general.

2012 ACM Subject Classification Theory of computation → Program verification; Hardware → Theorem proving and SAT solving; Theory of computation → Logic and verification

Keywords and phrases pseudo-Boolean solving, 0-1 integer linear program, proof logging, certified translation, CNF encoding, cutting planes

Digital Object Identifier 10.4230/LIPIcs...

Funding *Stephan Gocht*: Swedish Research Council grant 2016-00782.
Ruben Martins: National Science Foundation award CCF-1762363 and Amazon Research Award.
Jakob Nordström: Swedish Research Council grant 2016-00782 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.

1 Introduction

Boolean satisfiability (SAT) has witnessed striking improvements over the last couple of decades, starting with the introduction of *conflict-driven clause learning (CDCL)* SAT solvers [36, 39], and this has lead to a wide range of applications including large-scale problems in both academia and industry [8]. The conflict-driven paradigm has also been

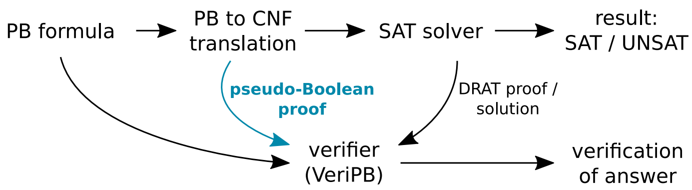


Figure 1 Proof logging workflow for pseudo-Boolean solving (our contribution in boldface).

successfully exported to other areas such as *maximum satisfiability (MaxSAT)*, *pseudo-Boolean (PB) solving*, *constraint programming (CP)*, and *mixed integer linear programming (MIP)*. As modern combinatorial solvers are used to attack ever more challenging problems, and employ ever more sophisticated optimizations and heuristics to do so, the question arises whether we can trust the results they produce. Sadly, it is well documented that state-of-the-art CP and MIP solvers can return incorrect solutions [1, 14, 24]. For SAT solvers, however, analogous problems [9] have been successfully addressed by the introduction of *proof logging*, requiring that solvers should be *certifying* [37] in the sense that they output machine-verifiable proofs of their claims that can be verified by a stand-alone *proof checker*.

A number of different proof logging formats have been developed for SAT, including *RUP* [28], *TraceCheck* [7], *DRAT* [29, 30, 50], *GRIT* [16], and *LRAT* [15], and since 2013 the SAT competitions [45] require solvers to be certifying, with *DRAT* established as the standard format. It would be highly desirable to have such proof logging also for stronger combinatorial solving paradigms, but while methods such as *DRAT* are extremely powerful in theory, the fact that they are limited to a clausal format makes it hard to capture more advanced forms of reasoning in a succinct way, and it is not even clear how to deal with input that is not in conjunctive normal form (CNF). One way to address this problem could be to allow extensions to the *DRAT* format [2], but another approach pursued in recent years is to develop stronger proof logging methods based on binary decision diagrams [4], algebraic reasoning [33, 44], pseudo-Boolean reasoning [21, 25, 26], or integer linear programming [12, 19].

Our Contribution In this work, we consider the use of CDCL for pseudo-Boolean solving, where the pseudo-Boolean input (i.e., a 0-1 integer linear program) is translated to CNF and passed to a SAT solver, as pioneered in *MiniSat+* [18]. The two solvers *Open-WBO* [41] and *NaPS* [40] using this approach were among the top performers in the latest pseudo-Boolean evaluation [43]. While *DRAT* proof logging can be used to certify unsatisfiability of the translated formula, it cannot prove the correctness of the translation, not only since there is no known method of carrying out PB reasoning efficiently in *DRAT* (except for constraints with small coefficients [10]), but also, and more fundamentally, because the input is not in CNF.

We demonstrate how to instead use the *cutting planes* method [13], enhanced with a rule allowing to introduce extension variables [27], to certify the correctness of translations of pseudo-Boolean constraints into CNF. Since this method is a strict extension of *DRAT*, we can combine the proof of the translation with the SAT solver *DRAT* proof log (with appropriate syntactic modifications) to achieve end-to-end verification of the pseudo-Boolean solving process using the proof checker *VeriPB* [48], as shown in Figure 1.

One challenge when certifying PB-to-CNF translations is that there are many different ways of encoding pseudo-Boolean constraints into CNF (as catalogued in, e.g., [42]), and it is time-consuming (and error-prone) to code up proof logging for every single encoding. However, many of the encodings can be understood as first designing a circuit to evaluate whether the PB constraint is satisfied, and then writing down a CNF encoding of the computation of this circuit. An important part of our contribution is that we develop a general framework to provide proof logging for a wide class of such circuits in a uniform way. The pseudo-Boolean format used for proof logging makes it easy to derive 0-1 linear inequalities describing the computations in the circuit, and once this has been done the desired clauses in the CNF translation can simply be obtained by so-called *reverse unit propagation (RUP)* [28, 47]. We have applied this method to the *sequential counter* [46], *totalizer* [3], *generalized totalizer* [32] and *adder network* [18, 49] encodings, and report results from an empirical evaluation.

Outline of This Paper After discussing preliminaries in Section 2, we illustrate our method for the sequential counter encoding in Section 3. Section 4 presents the general framework, and we briefly discuss how to apply it to adder networks in Section 5. (Due to space constraints, details for the totalizer and generalized totalizer encodings are omitted.) We report experimental data for proof logging and verification in Section 6 and conclude with a discussion of some possible directions for future research in Section 7.

2 Preliminaries

Let us start with a review of some standard material that can also be found in, e.g., [27] or in more detail in [11]. A *literal* ℓ over a Boolean variable x is x itself or its negation $\overline{x} = 1 - x$, where variables can be assigned values 0 (false) or 1 (true). For notational convenience, we define $\overline{\overline{x}} \doteq x$ (where we use $\doteq$ to denote syntactic equality). We sometimes write $\vec{x} = \{ x_1, \ldots, x_m \}$ to denote a set of variables. A *pseudo-Boolean (PB) constraint* is a 0-1 linear inequality

$$C \doteq \textstyle\sum_i a_i \ell_i \geq A \ , \tag{1}$$

which without loss of generality we always assume to be in *normalized form* [5]; i.e., all literals ℓ_i are over distinct variables and the coefficients a_i and the *degree (of falsity)* A are non-negative integers. The normalized form of the *negation* of C in (1) is

$$\neg C \doteq \textstyle\sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1 \ . \tag{2}$$

An equality constraint $C \doteq \sum_i a_i \ell_i = A$ is just syntactic sugar for the pair of inequalities $C^{\text{geq}} \doteq \sum_i a_i \ell_i \geq A$ and $C^{\text{leq}} \doteq \sum_i -a_i \ell_i \geq -A$ (rewritten in normalized form). Summing two equality constraints $C + D$ means taking the two sums $C^{\text{geq}} + D^{\text{geq}}$ and $C^{\text{leq}} + D^{\text{leq}}$. We write $\sum_i a_i \ell_i \bowtie A$ for $\bowtie \in \{ \geq, \leq, = \}$ for constraints that are either inequalities or equalities. A *pseudo-Boolean formula* is a conjunction $F \doteq \bigwedge_j C_j$ of PB constraints. Note that a *clause* $\ell_1 \vee \cdots \vee \ell_k$ is equivalent to the constraint $\ell_1 + \cdots + \ell_k \geq 1$, so CNF formulas are just special cases of PB formulas. A *cardinality constraint* is a PB constraint with all coefficients equal to 1.

A *(partial) assignment* ρ is a (partial) function from variables to $\{ 0, 1 \}$. Applying ρ to a constraint C as in (1), denoted $C{\upharpoonright_\rho}$, yields the constraint obtained by substituting values for all assigned variables, shifting constants to the right-hand side, and adjusting the degree appropriately, and for a formula F we define $F{\upharpoonright_\rho} \doteq \bigwedge_j C_j{\upharpoonright_\rho}$. The constraint C is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq A$ (or, equivalently, if the restricted constraint has a non-positive degree and is thus trivial). An assignment ρ satisfies $F \doteq \bigwedge_j C_j$ if it satisfies all C_j, in which case F is *satisfiable*. A formula without satisfying assignments is *unsatisfiable*. Two formulas are *equisatisfiable* if they are both satisfiable or both unsatisfiable.

Cutting planes as defined in [13] is a method for iteratively deriving new constraints C implied by a PB formula F. If C and D are previously derived constraints, or are *axiom constraints* in F, then any positive integer *linear combination* of these constraints can be added. We can also add *literal axioms* $\ell_i \geq 0$ at any time. Finally, from a constraint in normalized form $\sum_i a_i \cdot \ell_i \geq A$ we can use *division* by a positive integer d to derive $\sum_i \lceil a_i/d \rceil \ell_i \geq \lceil A/d \rceil$, dividing and rounding up the degree and coefficients.

For PB formulas F, F' and constraints C, C', we say that F *implies* or *models* C, denoted $F \models C$, if any assignment satisfying F must also satisfy C, and we write $F \models F'$ if $F \models C'$ for all $C' \in F'$. It is clear that any collection of constraints F' derived (iteratively) from F by cutting planes are implied in this sense.

A constraint C is said to *unit propagate* the literal ℓ under ρ if $C{\restriction}_\rho$ cannot be satisfied unless ℓ is satisfied. During *unit propagation* on F under ρ, we extend ρ iteratively by assignments to any propagated literals until an assignment ρ' is reached under which no constraint $C \in F$ is propagating, or under which some constraint C propagates a literal that has already been assigned to the opposite value. The latter scenario is called a *conflict*, since ρ' *violates* the constraint C in this case. We say that F implies C by *reverse unit propagation (RUP)*, and that C is a *RUP constraint* with respect to F, if F and the negation of C unit propagates to conflict under the empty assignment. It is not hard to see that $F \models C$ holds if C is a RUP constraint.

In addition to deriving constraints C that are implied by F, we will also need a rule for adding so-called *redundant* constraints D having the property that F and $F \wedge D$ are equisatisfiable. For this purpose we will use the *reification* rules described below, which are shown in [27] to be special cases of the redundance rule in that paper. Provided that z is a *fresh variable* that is not in the formula and has not appeared previously in the derivation, we can introduce the *reified constraints*

$$z \Rightarrow \textstyle\sum_i a_i \ell_i \geq A \quad \doteq \quad A\bar{z} + \sum_i a_i \ell_i \geq A \tag{3a}$$

and

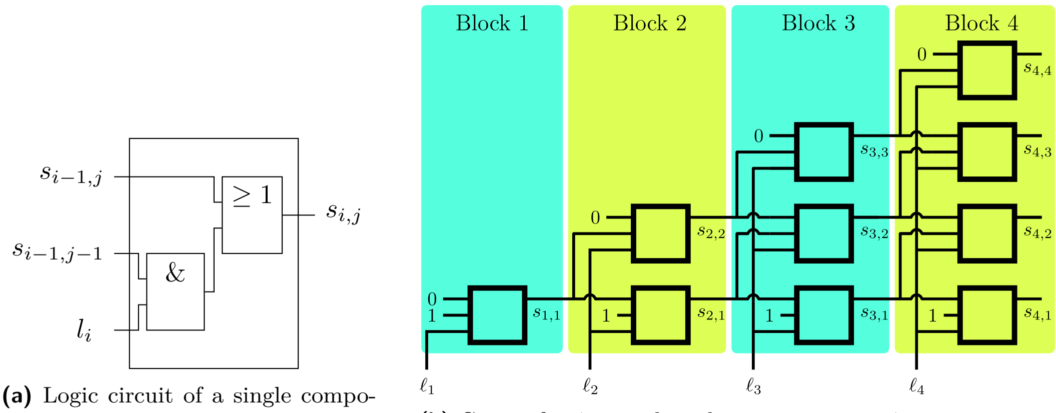
$$z \Leftarrow \textstyle\sum_i a_i \ell_i \geq A \quad \doteq \quad \left(\sum_i a_i - A + 1\right) \cdot z + \sum_i a_i \bar{\ell}_i \geq \sum_i a_i - A + 1 \ . \tag{3b}$$

A moment of thought reveals that the constraint (3a) says that if z is true, then $\sum_i a_i \ell_i \geq A$ has to hold, and this explains the notation $z \Rightarrow \sum_i a_i \ell_i \geq A$ introduced for this constraint. In an analogous fashion, the constraint (3b) says that if $\sum_i a_i \ell_i \geq A$ holds, then z has to be true. We will write $z \Leftrightarrow \sum_i a_i \ell_i \geq A$ for the conjunction of (3a) and (3b). It is easy to see that adding such reification constraints to a formula F preserves equisatisfiability, since any satisfying assignment to F can be extended by setting z as required to satisfy the implications.

3 Certified Translation for the Sequential Counter Encoding

To encode a cardinality constraint of the form $\sum_{i=1}^{n} \ell_i \bowtie k$ we can use the *sequential counter encoding* [46]. This encoding is designed after a circuit accumulating the sum of input bits using the intermediate fresh variables $s_{i,j}$ for $i \in [n], j \in [i]$, where $s_{i,j}$ is true if and only if the first i literals sum up to j. The variable $s_{i,j}$ is computed as in Figure 2a, i.e.,

$$s_{i,j} \leftrightarrow ((\ell_i \wedge s_{i-1,j-1}) \vee s_{i-1,j}) \ , \tag{4}$$



(a) Logic circuit of a single component.

(b) Circuit for 4 input literals counting up to 4.

Figure 2 Circuit representation of the sequential counter encoding.

that is either the first $i-1$ variables add up to $j-1$ and the i-th literal is true, or the first $i-1$ variables already add up to j. The resulting circuit is shown in Figure 2b and can be divided into multiple blocks, where the i-th block accumulates the i-th input literal and the variables $s_{i-1,j}$ for $j \in [i-1]$. We will use this block structure later as an abstract way to represent the encoding. The clausal encoding is given by translating the circuit into clausal form, i.e., via the clauses

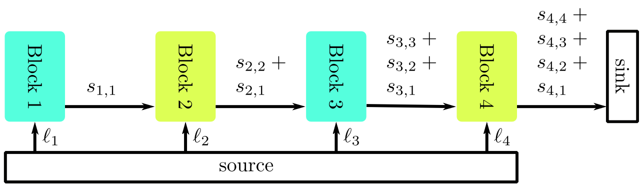
$$\bar{\ell}_i + \bar{s}_{i-1,j-1} + s_{i,j} \geq 1 \tag{5a}$$
$$\bar{s}_{i-1,j} + s_{i,j} \geq 1 \tag{5b}$$
$$\ell_i + s_{i-1,j} + \bar{s}_{i,j} \geq 1 \tag{5c}$$
$$s_{i-1,j-1} + \bar{s}_{i,j} \geq 1 \ , \tag{5d}$$

where $i \in [n]$ and $j \in [i]$. To cover corner cases we always replace $s_{i,j}$ for $j > i$ with 0 and $s_{i,j}$ for $j \leq 0$ with 1 and simplify the constraints accordingly. For example, for $i = j = 1$ we only get the clauses $\bar{\ell}_1 + s_{1,1} \geq 1$ and $\ell_1 + \bar{s}_{1,1} \geq 1$, since $s_{0,0}$ is replaced by 1 and hence the variable disappears from (5a) while (5d) is satisfied, and $s_{0,1}$ is replaced by 0 and thus disappears from (5c) and satisfies (5b). To enforce a greater-or-equal-k constraint it is only necessary to add the clause $s_{n,k} \geq 1$. Analogously, a less-or-equal-k constraint is enforced using the clause $\bar{s}_{n,k+1} \geq 1$. A common optimization, known as k-simplification, is to not add the clauses for variable $s_{i,j}$ if $j > k+1$, as these variables have no influence on the satisfiability of the clausal encoding.

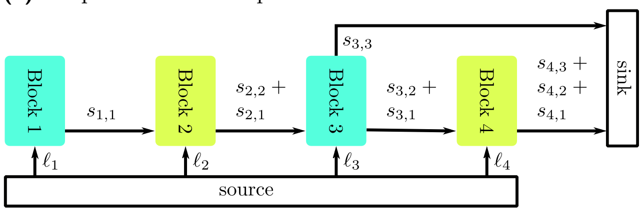
Before discussing the proof logging, let us study the encoding in more detail, ignoring k-simplification for now. Remember that the variable $s_{i,j}$ should be true if and only if the first i literals sum up to j and hence can be understood as a unary representation, where we want that $\sum_{j=1}^{i} \ell_j = \sum_{j=1}^{i} s_{i,j}$ for $i \in [n]$. However, the circuit is only using the variables from the previous block $s_{i-1,j}$ and the literal ℓ_i as input to compute the $s_{i,j}$ variables and hence it will instead be more convenient to consider the equality

$$\ell_i + \textstyle\sum_{j=1}^{i-1} s_{i-1,j} = \sum_{j=1}^{i} s_{i,j} \quad i \in [n] \ . \tag{6}$$

We can use this insight to get a more abstract representation of the circuit in Figure 2b, by thinking of blocks as nodes with two input edges labelled ℓ_i and $\sum_{j=1}^{i-1} s_{i-1,j}$ and an output edge labelled $\sum_{j=1}^{i} s_{i,j}$ as shown in Figure 3a. Additionally, for each inner node



(a) Graph without k-simplification.



(b) Graph with k-simplification for $k = 1$.

Figure 3 Graph representation of the sequential counter encoding.

the sum of all input labels should be equal to the sum of all output labels as enforced by (6), which we will call a *preserving equality*. This graph representation will be helpful to generalize the presented proof logging approach for other encodings.

Note that the sum of input variables coming from the source equals the sum of output variables on the edges going to the sink because each node preserves equality between incoming and outgoing values. That is we have $\sum_{j=1}^{n} \ell_i = \sum_{j=1}^{n} s_{n,j}$, which can also be obtained mathematically by summing all equalities of the form (6). Based on this equality, it is clear that a bound on the input variables $k \bowtie \sum_{j=1}^{n} \ell_i$ also implies a bound on the output variables, which can be seen by summing $k \bowtie \sum_{j=1}^{n} \ell_i$ and $\sum_{j=1}^{n} \ell_i = \sum_{j=1}^{n} s_{n,j}$ to get

$$k \bowtie \textstyle\sum_{j=1}^{n} s_{n,j} \quad . \tag{7}$$

Another important observation is that the variables $s_{i,j}$ should not just take any value satisfying (6), but they should also be ordered, that is if $s_{i,j+1}$ is true, the sum should be at least $j+1$ and hence also at least j and $s_{i,j}$ should be true as well (and also $s_{i,j-1} = 1, s_{i,j-2} = 1$ etc.). This can be enforced with *ordering constraints*

$$s_{i,j} \geq s_{i,j+1} \quad i \in [n], j \in [i-1] \quad . \tag{8}$$

With this improved understanding of the encoding, we can now tackle the task of proof logging, which becomes surprisingly simple. The constraints (6), (7), (8) are all pseudo-Boolean constraints and if we are able to derive them, then the clauses of the sequential counter encoding ((5) and $\bar{s}_{n,k+1} \geq 1$ and/or $s_{n,k} \geq 1$) can all be derived via reverse unit propagation: The propagations due to (8) will cause enough variables to propagate, such that (6) is falsified. The derivation of (7) from (6) was already discussed when introducing (7), where we summed all constraints (6) and the constraint to be encoded. This summation can be expressed directly in cutting planes. For deriving the other constraints, remember that for proof logging we want to demonstrate that adding constraints does not change satisfiability. However, it is easy to see that the preserving equality (6) and ordering constraints (8) can always be satisfied by choosing a suitable value for the $s_{i,j}$ variables. If the constraints are added in ascending order of i, then the $s_{i,j}$ are fresh and can indeed be chosen freely. In the proof format this reasoning is expressed through reification as discussed in the next example and for the general case in Appendix A.1.

▶ **Example 1.** Let us consider how to derive the preserving equality

$$\ell_3 + s_{2,1} + s_{2,2} = s_{3,1} + s_{3,2} + s_{3,3} \tag{9}$$

for Block 3 in Figure 3a. To satisfy (9) we want that $s_{3,1}$ is true if $\ell_3 + s_{2,1} + s_{2,2}$ is greater equal 1, $s_{3,2}$ is true if it is greater equal 2 and $s_{3,3}$ is true if it is greater equal 3. We can enforce these conditions by introducing the fresh variables $s_{3,1}, s_{3,2}, s_{3,3}$ via reification, i.e., $s_{3,1} \Leftrightarrow \ell_3 + s_{2,1} + s_{2,2} \geq 1$, $s_{3,2} \Leftrightarrow \ell_3 + s_{2,1} + s_{2,2} \geq 2$ and $s_{3,3} \Leftrightarrow \ell_3 + s_{2,1} + s_{2,2} \geq 3$, which results in the pseudo-Boolean constraints

$$\bar{s}_{3,1} + \ell_3 + s_{2,1} + s_{2,2} \geq 1 \tag{10a}$$
$$2\bar{s}_{3,2} + \ell_3 + s_{2,1} + s_{2,2} \geq 2 \tag{10b}$$
$$3\bar{s}_{3,3} + \ell_3 + s_{2,1} + s_{2,2} \geq 3 \tag{10c}$$
$$3s_{3,1} + \bar{\ell}_3 + \bar{s}_{2,1} + \bar{s}_{2,2} \geq 3 \tag{10d}$$
$$2s_{3,2} + \bar{\ell}_3 + \bar{s}_{2,1} + \bar{s}_{2,2} \geq 2 \tag{10e}$$
$$s_{3,3} + \bar{\ell}_3 + \bar{s}_{2,1} + \bar{s}_{2,2} \geq 1 \ . \tag{10f}$$

By design, (10) implies (9) and hence (9) can be derived via cutting planes. To do so in practice, we accumulate the constraints (10a)-(10c) while maintaining the invariant $\sum_{j=1}^{i} \bar{s}_{3,j} + \ell_3 + s_{2,1} + s_{2,2} \geq i$, where $i = 1, 2, 3$ is the number of accumulated constraints. When starting with (10a) the invariant holds. Next we add (10b) and divide by 2 to obtain $\bar{s}_{3,1} + \bar{s}_{3,2} + \ell_3 + s_{2,1} + s_{2,2} \geq 2$ and continue by multiplying with 2, adding (10c) and dividing by 3, which results in $\bar{s}_{3,1} + \bar{s}_{3,2} + \bar{s}_{3,3} + \ell_3 + s_{2,1} + s_{2,2} \geq 3$, which is equivalent to $\ell_3 + s_{2,1} + s_{2,2} \geq \bar{s}_{3,1} + \bar{s}_{3,2} + \bar{s}_{3,3}$, as desired. Analogously, we can accumulate (10d)-(10f) in reverse order to obtain $\ell_3 + s_{2,1} + s_{2,2} \leq \bar{s}_{3,1} + \bar{s}_{3,2} + \bar{s}_{3,3}$. The ordering constraints $s_{3,1} \geq s_{3,2}$ can be obtained by adding (10d) and (10b), which yields $3s_{3,1} + 2\bar{s}_{3,2} \geq 1$ and can be divided by 3 to obtain $s_{3,1} + \bar{s}_{3,2} \geq 1$, which is equivalent to $s_{3,1} \geq s_{3,2}$, as desired. Analogously, we can obtain $s_{3,2} \geq s_{3,3}$ by using (10e) and (10c).

To perform k-simplification, we could simply omit deriving the unneeded clauses, however this potentially introduces a large overhead for proof logging if k is small, as we would always introduce $O(n^2)$ intermediate variables instead of the $O(kn)$ variables that are needed. To avoid this overhead, as demonstrated in Figure 3b, we want that the edge going to the next block is labelled with $\sum_{j=1}^{k+1} s_{i,j}$ instead of $\sum_{j=1}^{i} s_{i,j}$. However, this means we need to introduce an additional edge going directly to the sink with the label $s_{i,k+2}$ to preserve the equality of in- and output, i.e.,

$$\ell_i + \sum_{j=1}^{k+1} s_{i-1,j} = \sum_{j=1}^{k+2} s_{i,j} \quad i \in [n] \ . \tag{11}$$

Note that without the additional variable $s_{i,k+2}$ we could not guarantee equality, as we would have $k + 2$ literals on the left hand side and only $k + 1$ fresh variable on the right hand side.

▶ **Example 2.** To demonstrate k-simplification, consider Block 3 in Figure 3b, which has input edges with labels $s_{2,1} + s_{2,2}$ and ℓ_3 and let us perform 1-simplification. The output of Block 3 to Block 4 should only contain the 2 variables $s_{3,1} + s_{3,2}$. To preserve equality of in- and output, we add an edge from Block 3 to the sink labelled $s_{3,3}$.

As before, we can obtain the constraint that in- and output of the graph are equal by summing the preserving constraint (11) of each node, which yields $\sum_{i=1}^{n}\left(\ell_i + \sum_{j=1}^{k+1} s_{i-1,j}\right) = \sum_{i=1}^{n}\left(\sum_{j=1}^{k+2} s_{i,j}\right)$ and can be simplified to $\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} s_{i,k+2} + \sum_{j=1}^{k+1} s_{n,j}$.

4 General Framework for Certifying CNF Translations

A major challenge of providing proof logging for translations of pseudo-Boolean constraints to CNF is that there are so many different encodings of pseudo-Boolean constraints. To support a wide range of encodings, we can generalize the idea of the graph representation used in the previous section to obtain a general framework. The main ingredient of the framework is a graph representing the connection between the variables of the encoded constraint and auxiliary variables used in the encoding. This graph has the property that we can derive a preserving equality of in- and output for each node and that the CNF encoding follows from these equalities. To derive the preserving equality, we provide proof logging for general purpose operations for different ways to represent natural numbers. Let us start with a formal definition of the graph representation.

▶ **Definition 3** (Arithmetic Graph). *An* arithmetic graph *with input $\sum_i a_i x_i$ and output $\sum_i c_i o_i$ is a directed graph $G = (V, E)$ with a source node s, a sink node t, and edge labels of the form $\sum_i b_i^e y_i^e$ for each edge $e \in E$. For convenience, we allow to have multiple edges between two nodes. Additionally, we require that*

- *the source s has only outgoing edges and the input is split among edges of s, i.e., $\sum_i a_i x_i \equiv \sum_{(s,v)=e\in E} \sum_i b_i^e y_i^e$,*
- *the sink t has only incoming edges and the output is split among edges of t, i.e., $\sum_i c_i o_i \equiv \sum_{(v,t)=e\in E} \sum_i b_i^e y_i^e$, and*
- *for every inner node v the input is equal to the output, which can be derived via proof logging, i.e., we can derive the preserving equality*

$$\sum_{(u,v)=e\in E} \sum_i b_i^e y_i^e = \sum_{(v,u)=e\in E} \sum_i b_i^e y_i^e \quad . \tag{12}$$

The general strategy for providing proof logging will be to formulate the used encoding in terms of an arithmetic graph, where the preserving equality (12) will depend on the representation of natural numbers used in the encoding and will be derived using one of the operations described later in this section. For each encoding, we will make sure that the clauses in the encoding directly correspond to a node in the graph and will follow by reverse unit propagation from the preserving equality (12). However, each encoding has also clauses to restrict the output variables o_i, which can only be derived after translating the bound known on the input variables to a bound on the output variables.

▶ **Proposition 4.** *Given an arithmetic graph with input $\sum_i a_i x_i$ and output $\sum_i c_i o_i$ and a pseudo-Boolean constraint $\sum_i a_i x_i \bowtie k$, where $\bowtie \in \{\geq, \leq, =\}$, we can derive $\sum_i c_i o_i \bowtie k$ using cutting planes.*

Proof. As we have an arithmetic graph, we know that we can derive (12) for every inner node in the graph. By adding all these constraints together, we obtain the constraint $\sum_i a_i x_i = \sum_i c_i o_i$, which can be combined with $\sum_i a_i x_i \bowtie k$ to obtain $\sum_i c_i o_i \bowtie k$. ◀

Algorithm 1 General algorithm for proof logging arithmetic encodings.

procedure proof_log_encoding(C, f, G, F)
    ▷ input: C is of the form $\sum_{i=1}^{n} a_i\ell_i \bowtie k$, with $k, n \in \mathbb{N}$ and $\bowtie \in \{\geq, \leq, =\}$.
    ▷ input: an arithmetic graph $G = (V, E)$ with input $\sum_i a_i x_i$ and output $\sum_i c_i o_i$
    ▷ input: a function f that takes a node and derives its preserving equality
    ▷ input: the CNF encoding F to be derived
    sum the constraints $f(v)$ for $v \in V$ in topological order to obtain $\sum_i a_i x_i = \sum_i c_i o_i$
    combine $\sum_i a_i x_i = \sum_i c_i o_i$ and C to obtain $\sum_i c_i o_i \bowtie k$
    derive each clause in the CNF encoding F via RUP

Once the bound on the input variables is translated to a bound on the output variables, all clauses of the CNF encoding will follow by reverse unit propagation. This results in the general algorithm for proof logging encodings shown in Algorithm 1. Note that the nodes of the graph need to be traversed in a topological order when deriving the preserving equality. Otherwise we can not use that the output variables of a node are fresh, which will be crucial for the presented derivations.

Let us now discuss three common ways to represent natural numbers, as well as some general purpose operations on these representations that are used to derive the preserving equality for inner nodes. The easiest way to encode a natural number j with domain $A = \{0, 1, \ldots, m\}$ using Boolean variables is to use a unary number, where the number of variables z_i set to true is equal to j, i.e., $j = \sum_{i\in[m]} z_i$. For better propagation behaviour, it is usually required that the z_i variables are ordered via constraints $z_i \geq z_{i+1}$, which enforces that z_i is true if and only if $j \geq i$. This representation is used in the sequential counter [46] and totalizer encoding [3] and is known as *order encoding*.

▶ **Proposition 5** (Unary Sum). *For any literals $\ell_1, \ldots, \ell_n$ we can derive the constraints*

$$\textstyle\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} z_i \tag{13}$$

$$z_i \geq z_{i+1} \quad i \in [n-1] \ . \tag{14}$$

using $O(n)$ steps, where $z_1, \ldots, z_n$ are fresh variables.

Conceptually, adding these constraints does not change satisfiability, because they can always be satisfied using the fresh variables. We already discussed deriving these constraints in the context of the sequential counter encoding. The general idea is to introduce the fresh variables via reification $z_i \Leftrightarrow \sum_{i=1}^{n} \ell_i \geq i$, after which we can obtain the greater-than part of the equality by maintaining the invariant $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \bar{z}_i \geq j$ and analogously for the less-than part. A detailed description of the algorithm for deriving a unary sum is provided in Appendix A.1.

If we want to encode a natural number j, for which we know that it can only take values in a small domain A, then introducing variables for all values in the range introduces a lot of redundant variables. For example if $j \in \{0, 50, 75\}$, then the first 50 variables in a full unary representation are either all true or all false, but will never take different values. For a more concise encoding we can use a sparse representation, i.e., we represent $j \in \{0, 50, 75\}$ as $50 \cdot z_{50} + 25 \cdot z_{75}$ and enforce that $z_{50} \geq z_{75}$. In general, we use

$$\mathit{sparse}(\vec{z}, A) = \textstyle\sum_{i \in A\setminus\{0\}} (i - \mathit{pred}(i, A)) z_i \ , \tag{15}$$

where $\mathit{pred}(i, A) = \max(\{j \in A \mid j < i\})$. Additionally, we enforce that the z_i variables are ordered, i.e., $z_i \geq z_{\mathit{succ}(i,A)}$, where $\mathit{succ}(i, A) = \min(\{j \in A \cup \{\infty\} \mid j > i\})$. This representation is used in the sequential weight counter [31] and generalized totalizer encoding [32].

▶ **Proposition 6** (Sparse Unary Sum). *Given $A, B \subseteq \mathbb{N}$, $E = \{ i + j \mid i \in A, j \in B \}$, ordering constraints on variables $\vec{y}$ and $\vec{y}\,'$, as well as fresh variables $\vec{z}$, we can derive*

$$sparse(\vec{y}, A) + sparse(\vec{y}\,', B) = sparse(\vec{z}, E) \ , \ and \tag{16a}$$

$$z_i \geq z_{succ(i,E)} \quad i \in E \setminus \{ \max(E) \} \ , \tag{16b}$$

using $O(|A| \cdot |B|)$ steps.

As in the case of the unary sum, these constraints can be added without changing satisfiability, because we can always set the fresh z_i variables such that the constraints are satisfied. The general idea is to introduce the fresh variables via reification $z_i \Leftrightarrow \sum_{i=1}^{n} \ell_i \geq i$. Then we simulate a brute-force search on the possible combinations of values for A and B, showing that the equality holds in all cases. A detailed description can be found in Appendix A.2.

Finally, if we want to represent a natural number that is large and has a large domain with maximal value m, then we can encode it using a binary representation, i.e., $j = \sum_{i=0}^{\lfloor \log_2(m) \rfloor} 2^i z_i$. To build a binary number (as is discussed in Section 5) it sufficient to compose multiple full adders, which compute the sum of up to three input bits, using a binary adder circuit [18].

▶ **Proposition 7.** *For literals ℓ_1, ℓ_2, ℓ_3 and fresh variables z_1, z_0 we can derive the constraints*

$$\ell_1 + \ell_2 + \ell_3 = 2z_1 + z_0 \tag{17}$$

using $O(1)$ steps.

Again, it should be clear that this equality can be added without changing satisfiability because it can be satisfied using the fresh variables. To derive it, we reify

$$c \Leftrightarrow x + y + z \geq 2 \tag{18a}$$

$$s \Leftrightarrow x + y + z + 2\bar{c} \geq 3 \ . \tag{18b}$$

The equality can be derived by multiplying (18a) by 2, adding (18b) and dividing the result by 3 as discussed in detail in [27].

In Section 5 and Appendix B, it is demonstrated how to apply this framework for the binary adder and the (generalized) totalizer encoding, respectively.

5 Binary Adder Encoding

The *binary adder encoding* [18] is used to encode general pseudo-Boolean constraints of the form $\sum_i a_i \ell_i \bowtie k$. The idea is to use an adder network to obtain the value of $\sum_i a_i \ell_i$ as a binary number $\sum_{i=0}^{bits} 2^i o_i$, where o_i are the output literals and $bits = \lfloor \log_2(\sum_i a_i) \rfloor$ is the required bit width. To enforce the constraint, the output bits o_i are constrained by clauses that perform a bitwise comparison with k in binary representation.

To recapitulate the algorithm for the construction of the adder network in [18], we need some more notation. A 2^m*-bit* is a literal that represents the numerical value 2^m. A 2^m*-bucket* is a queue of bits where each bit has the value 2^m and that supports

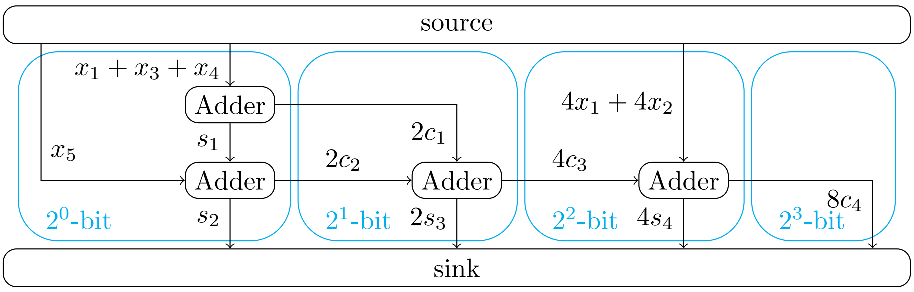


Figure 4 Layout of the arithmetic graph for adder network encoding of $5x_1+4x_2+x_3+x_4+x_5 \geq 5$.

operations to insert and extract bits. We use $[m]_2$ to denote the binary representation of a natural number m.

The construction of the network starts by initializing each 2^m-bucket with all literals ℓ_i such that the 2^m-bit of $[a_i]_2$ is 1. Then we repeat the following steps until there is at most one element left in each bucket. Consider the 2^m-bucket with the smallest value that has at least 2 elements in it. If there are only 2 elements in the 2^m-bucket, take x and y from the bucket and set $z = 0$. Otherwise, let x, y and z be 3 elements from the 2^m-bucket and remove them from the 2^m-bucket. The bits x, y and z are used as input for a new full adder with fresh variables c and s as output, where c is a 2^{m+1}-bit and s is a 2^m-bit. The bits c and s are then inserted in their respective buckets, possibly creating a new bucket. An algorithm for constructing the network is given in Appendix A.3.

The arithmetic graph is constructed directly from the adder network such that each full adder is represented by a node. Each inner node constructed from the 2^m-bucket, i.e., which has 2^m-bits as input, has input edges with labels $2^m x$, $2^m y$ and $2^m z$ and output edges with labels $2^m s$ and $2^{m+1} c$. An example of the resulting graph is shown in Figure 4. The preserving equality can be derived using Proposition 7 and multiplying the resulting equality $x + y + z = 2c + s$ by 2^m to obtain $2^m x + 2^m y + 2^m z = 2^{m+1} c + 2^m s$. After construction of the adder network, each 2^m-bucket has at most one 2^m-bit left and we connect the corresponding edges to the sink, resulting in an output of the form $\sum_{i=0}^{bits} 2^i o_i$. If the 2^m-bucket is empty, o_m is set to 0.

Each full adder of the network is encoded to CNF via the clauses

$$\begin{array}{llll}
 & \bar{x} + \bar{y} + \bar{z} + s \geq 1 & & x + y + z + \bar{s} \geq 1 \\
\bar{y} + \bar{z} + c \geq 1 & \bar{x} + y + z + s \geq 1 & y + z + \bar{c} \geq 1 & x + \bar{y} + \bar{z} + \bar{s} \geq 1 \\
\bar{x} + \bar{z} + c \geq 1 & x + \bar{y} + z + s \geq 1 & x + z + \bar{c} \geq 1 & \bar{x} + y + \bar{z} + \bar{s} \geq 1 \\
\bar{x} + \bar{y} + c \geq 1 & x + y + \bar{z} + s \geq 1 & x + y + \bar{c} \geq 1 & \bar{x} + \bar{y} + z + \bar{s} \geq 1 \ .
\end{array} \tag{19}$$

Note that all the clauses in (19) are RUP with respect to the preserving equality $x + y + z = 2c + s$.

To compare k with the output of the circuit, the encoding performs the comparison $\vec{x} \geq \vec{y}$ for bit vectors $\vec{x}$ and $\vec{y}$, where either $\vec{x} = o_{bits} \ldots o_1 o_0$ and $\vec{y} = [k]_2$ or vice versa, depending on whether we want to encode $\sum_{i=1}^{n} a_i \ell_i \geq k$ or $\sum_{i=1}^{n} a_i \ell_i \leq k$, respectively. If we want to encode $\sum_{i=1}^{n} a_i \ell_i = k$, then the comparison for both directions is performed. If the size of these vectors is different, the shorter vector is padded with 0. Then, for $i = 0, \ldots, bits$, the constraint

$$\bar{x}_i + y_i + \sum_{j=i}^{bits} x_j \bar{y}_j + \bar{x}_j y_j \geq 1 \tag{20}$$

is added to the CNF encoding. Note that either $\vec{x}$ or $\vec{y}$ is constant and hence the constraint is always a clause. This clause guarantees that the 2^i-bit on the variable side is equal to the 2^i-bit in $[k]_2$ or there was already a 2^j-bit for $j > i$ that is different to the 2^j-bit in $[k]_2$.

The clauses (20) are RUP with respect to $\sum_{i=0}^{bits} 2^i o_i \bowtie k$, which we obtain from the arithmetic graph using Proposition 4. The clauses are RUP because the RUP step will set all 2^j-bits, where $j > i$, to the same value as in $[k]_2$ and the 2^i-bit to the opposite value of the 2^i-bit in $[m]_2$, which falsifies $\sum_{i=0}^{bits} 2^i o_i \bowtie k$.

6 Experimental Results

To show the generality of our approach for proof logging arithmetic encodings, we implemented the sequential counter encoding [46], binary adder encoding [18], totalizer [3] and generalized totalizer encodings [32], in a certified encoding framework called VeritasPBLib. This framework inputs a pseudo-Boolean formula in OPB format and returns a CNF translation with the corresponding proof logging certificate. We used the verifier *VeriPB* [48] to verify the proof logging certificate returned by VeritasPBLib. The CNF formula is then solved by a modified version of the SAT solver *kissat* [34] [1] that generates proof logging compatible with the *VeriPB* verifier. Finally, we conjoin the proof logging from the CNF translation with the proof logging from SAT solving and verify the end-to-end pipeline with *VeriPB*.

The experiments were conducted on Amazon EC2 r5.large instances (2 vCPU) with Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz CPUs, 16 GB of memory, and gp2 volumes. We ran one process on each instance with a memory limit of 15 GB and a time limit of 7,200 seconds for verifying the proof with *VeriPB*, and a time limit of 1,800 seconds for CNF translation with VeritasPBLib and SAT solving with *kissat*. We gave additional time for verification, since verification is slower than solving the problem.

To evaluate VeritasPBLib, we collected 1,803 pseudo-Boolean formulas from the PB 2016 Evaluation.[2] We can split these instances into four categories: (1) formulas with only clauses (279 instances), (2) formulas with clauses and cardinality constraints (772 instances), (3) formulas with clauses and general PB constraints (444 instances), and (4) formulas with clauses, cardinality and general PB constraints (308 instances). Since this work targets the verification of formulas with cardinality or general PB constraints, we excluded the 279 pure CNF formula instances, as those can already be certified with existing techniques. More details about the instances can be found in Appendix C.1.

The goal of our evaluation is to answer the following questions:

1. Can we use the end-to-end framework to verify the results of SAT-based approaches to solve pseudo-Boolean formulas and how efficient is verification?
2. How long does verification of the proof logging take when compared to translating the pseudo-Boolean formula to CNF?

End-to-End Solving and Verification Table 1 shows how VeritasPBLib can be used to generate a CNF formula that can be solved by *kissat* and verified by *VeriPB*. For instances with cardinality constraints (*Card*), we use the sequential and totalizer encoding to translate those constraints to CNF. For instances with general PB constraints

---

[1] Available at `https://gitlab.com/MIAOresearch/kissat_fork`

[2] Available at `http://www.cril.univ-artois.fr/PB16/`

Table 1 Number of translated, solved and verified instances for each encoding

Category	#Inst	Encoding	Translation #CNF	Translation #Veri	Solving #Solved SAT	Solving #Solved UNSAT	Solving #Verified SAT	Solving #Verified UNSAT
Card	772	Sequential	772	772	139	480	133	479
		Totalizer	772	772	139	475	130	474
PB	444	Adder	444	444	179	167	178	165
		GTE	425	414	164	162	150	151
Card+PB	308	Seq+Adder	306	296	134	152	128	151

(*PB*), we use the adder and generalized totalizer encoding (GTE) to translate general PB constraints to CNF. Finally, for instances with both cardinality and general PB constraints (*Card+PB*), we use the sequential encoding for cardinality constraints and the adder encoding for PB constraints, henceforth denoted by *Seq+Adder*. Even though other combinations of cardinality and PB encodings could be explored, the goal of this work is not to find the best performing encodings but to show that we can verify the final result with a variety of encodings.

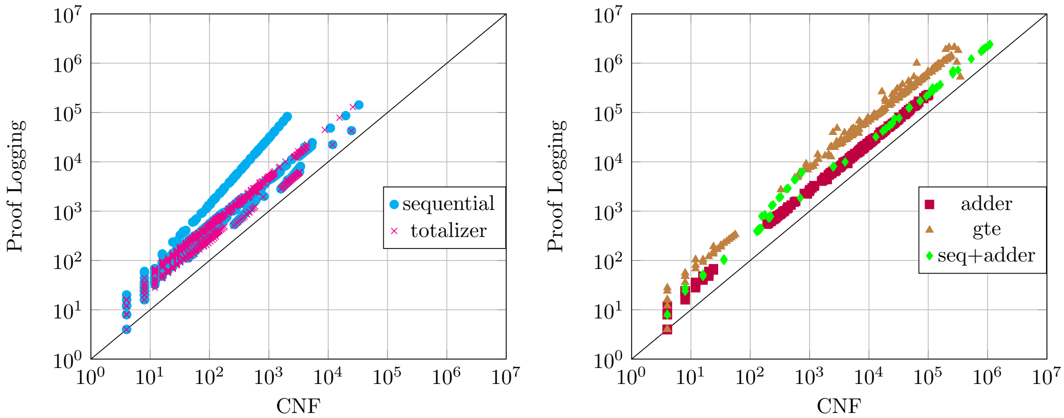
The column *#CNF* shows for how many instances VeritasPBLib successfully generated the CNF translation. For most of the formulas, we can translate the PB formula to CNF. The exceptions are 19 instances using the generalized totalizer (*GTE*) encoding and 2 instances using the *Seq+Adder* encoding. In those cases, the number of clauses generated is too large and exceeds the resource limits used in our evaluation.

The column *#Veri* under translation shows how many instances *VeriPB* can verify the proof logging certificate generated by VeritasPBLib. Except for a few instances for the *GTE* and *Seq+Adder* where the proof is large, *VeriPB* can verify the CNF translation. Note that if verification of the translation is successful, then this guarantees that the CNF encoding does not remove any solutions of the PB formula.

The columns *#Solved* and *#Verified* under solving show how many instances can be solved by the SAT solver *kissat* and from those how many can be verified by *VeriPB*. If a satisfiable formula is verified, then it means that all clauses derived by *kissat* are due to correct derivations and the satisfying assignment returned by the SAT solver is a satisfying assignment of the original PB formula. If an unsatisfiable formula is verified, then it means that the reason of unsatisfiability is due to correct derivations.

We can verify 99% of the solved instances for unsatisfiable instances, which shows that the current approach can be used in practice to verify unsatisfiable results of SAT solvers when solving PB formulas. For satisfiable instances, we can verify 95% of the solved instances. However, for instances that *VeriPB* does not verify the result within the time limit, we can still certify that the satisfying assignment of the SAT solver satisfies the original PB formula. Even though *VeriPB* is already able to verify the majority of the proof logging, improvements to the verifier are orthogonal to our approach and can further increase the number of verified instances.

Translation and Verification Let us now focus on the CNF translation without solving. Our experiments show that the average overhead for proof logging ranges from 2× to 3× slower for all encodings with the exception of GTE which is around 5× slower. However, since translation is fast for the majority of instances (see Figure 6), the additional overhead of proof logging is not an issue when translating the PB formulas to CNF. A more detailed comparison of running times between CNF translation with and without



(a) Cardinality formulas

(b) General pseudo-Boolean formulas

Figure 5 Comparison between CNF file size and proof logging file size in KiB

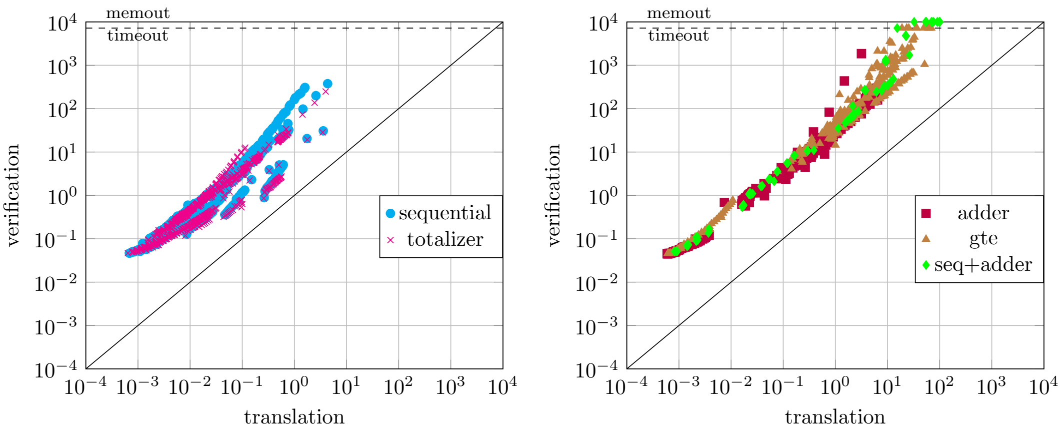
proof logging can be found in Appendix C.2.

The overhead for translation can be explained with the increased proof size compared to the size of the CNF encoding as shown in Figure 5. For most instances the proof size seems to be within a constant factor of the CNF file size. However, there is a series of benchmarks for which the sequential counter encoding requires super linear (but still polynomial) proofs. It turns out that these instances are all crafted instances encoding a vertex cover [20]. These instances contain a constraint enforcing a constant fraction of the literals in the formula to be true, which is the worst case scenario for the sequential counter. At first glance, this super linear relationship seems to contradict the expected linear relationship between the number of clauses in the CNF and the number of steps in the proof. However, this can be explained as each reification step for deriving the unary sum introduces a constraint of linear size, so even though the number of steps for deriving a unary sum is linear, the proof size will be quadratic. It would be desirable to find a derivation of the unary sum that only requires linear proof size.

Figure 6 shows the relationship between the time to generate the CNF translation using VeritasPBLib and the time to verify the translation using *VeriPB*. The time to verify the translation compared to the translation itself is not negligible. Over all encodings, for 75% of benchmarks verification takes at-most 49 times longer than translation and for 98% of benchmarks take at-most 100 times longer. To some degree, such an overhead in verification time of the translation is expected, as the translation does not need to reason about its steps and the verification needs to perform some reasoning to justify the correctness of the proof steps. However, this also indicates that there is still room for improvement, both in terms of improving the performance of the verifier but potentially also by finding easier to verify derivation steps.

7 Concluding Remarks

In this work, we develop a general framework for certified translations of pseudo-Boolean constraints into CNF using cutting-planes-based proof logging. Since our method is a strict extension of *DRAT*, the proof for the translation can be combined with a SAT solver *DRAT* proof log to provide, for the first time, end-to-end verification for CDCL-based pseudo-Boolean solvers. Our use of the cutting planes method is not only



(a) Cardinality formulas

(b) General pseudo-Boolean formulas

Figure 6 Comparison between CNF translation and verification of the corresponding proof logging

crucial to deal with the pseudo-Boolean format of the input, but the expressivity of the 0-1 linear constraints also allows us to certify the correctness of the translation to CNF in a concise and elegant way. While there is still room for performance improvements in proof logging and verification, our experimental evaluation shows that this approach is feasible in practice.

We want to point out that the tools we develop can also be used for the more general task of proving equivalence of reformulated problems. For the decision problem for a PB formula F, we only need to show that the CNF translation $Tr(F)$ can be derived from F, since a proof of unsatisfiability of $Tr(F)$ then shows that F is also unsatisfiable. However, our method can be adapted to show that if the PB formula F over variables X is translated to a CNF formula $Tr(F)$ over variables $X \cup Y$, then the two formulas are equivalent in the sense that (i) any satisfying assignment α to F propagates an assignment β to Y such that $\alpha \cup \beta$ satisfies $Tr(F)$, and (ii) for any satisfying assignment $\alpha \cup \beta$ to $Tr(F)$ it holds that α satisfies F. We believe that such certified problem reformulation should be useful also in, e.g., constraint programming.

In our view, proof logging for pseudo-Boolean decision problems is only a first step. We believe that our method should also be sufficient to support proof logging for MaxSAT solvers. As a concrete example, using the techniques developed in this paper it should be possible to certify the clauses added during core extraction and objective function reformulation in *core-guided MaxSAT solving* [23, 38]. While supporting MaxSAT solvers using approaches such as *implicit hitting set (IHS)* [17] and *abstract cores* [6] seems a bit more challenging, we are still hopeful that our work could lead to a unified proof logging method for both MaxSAT solving and pseudo-Boolean optimization using cutting-planes-based reasoning as in [22, 35].

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 Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12651 of *Lecture Notes in Computer Science*, pages 59–75. Springer, MarchApril 2021.

3 Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP '03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, September 2003.

4 Lee A. Barnett and Armin Biere. Non-clausal redundancy properties. In *Proceedings of the 28th International Conference on Automated Deduction (CADE-28)*, 2021.

5 Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, January 1995.

6 Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set MaxSat solving. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, July 2020.

7 Armin Biere. Tracecheck. `http://fmv.jku.at/tracecheck/`, 2006. Accessed on 2021–03–19.

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

9 Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, July 2010.

10 Randal E. Bryant, Armin Biere, and Marijn J. H. Heule. Clausal proofs for pseudo-boolean reasoning. In *Tools and Algorithms for the Construction and Analysis of Systems - TACAS 2022*, page To appear., 2022.

11 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.

12 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.

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

14 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.

15 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, 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 *LNCS*, pages 220–236. Springer, 2017.

16 Luís Cruz-Filipe, Joao 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 *LNCS*, pages 118–135. Springer, 2017.

17 Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP '11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, September 2011.

18 Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.

19 Leon Eifler and Ambros Gleixner. A computational status update for exact rational mixed integer programming. In *Proceedings of the 22nd International Conference on Integer Programming and Combinatorial Optimization (IPCO '21)*, volume 12707 of *Lecture Notes in Computer Science*, pages 163–177. Springer, May 2021.

20 Jan Elffers, Jesús Giráldez-Cru, Jakob Nordström, and Marc Vinyals. Using combinatorial benchmarks to probe the reasoning power of pseudo-Boolean solvers. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, pages 75–93. Springer, July 2018.

21 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.

22 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.

23 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, August 2006.

24 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.

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

26 Stephan Gocht, Ciaran McCreesh, 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.

27 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.

28 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.

29 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.

30 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.

31 Steffen Hölldobler, Norbert Manthey, and Peter Steinke. A compact encoding of pseudo-boolean constraints into SAT. In Birte Glimm and Antonio Krüger, editors, *Proceedings of KI 2012: Advances in Artificial Intelligence, the 35th Annual German Conference on AI*, volume 7526 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2012.

32 Saurabh Joshi, Ruben Martins, and Vasco M. Manquinho. Generalized totalizer encoding for pseudo-Boolean constraints. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP '15)*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, August-September 2015.

33 Daniela Kaufmann, Mathias Fleury, and Armin Biere. The proof checkers pacheck and pastèque for the practical algebraic calculus. In *Proceedings of the 20th International Conference on Formal Methods in Computer-Aided Design (FMCAD '20)*, pages 264–269. IEEE, 2020.
34 Kissat SAT solver. `http://fmv.jku.at/kissat/`.
35 Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, July 2010.
36 João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. Preliminary version in *ICCAD '96*.
37 Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.
38 António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João P. Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, October 2013.
39 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 530–535, June 2001.
40 NaPS (Nagoya pseudo-Boolean solver). `https://www.trs.cm.is.nagoya-u.ac.jp/projects/NaPS/`.
41 Open-WBO: An open source version of the MaxSAT solver WBO. `http://sat.inesc-id.pt/open-wbo/`.
42 Tobias Philipp and Peter Steinke. PBLib – a library for encoding pseudo-Boolean constraints into cnf. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT '15)*, volume 9340 of *Lecture Notes in Computer Science*, pages 9–16. Springer, September 2015.
43 Pseudo-Boolean competition 2016. `http://www.cril.univ-artois.fr/PB16/`, July 2016.
44 Daniela Ritirc, Armin Biere, Manuel Kauers, A Bigatti, and M Brain. A practical polynomial calculus for arithmetic circuit verification. In *3rd International Workshop on Satisfiability Checking and Symbolic Computation (SC2'18)*, pages 61–76, 2018.
45 The international SAT Competitions web page. `http://www.satcompetition.org`.
46 Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP '05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, October 2005.
47 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`.
48 VeriPB: Verifier for pseudo-Boolean proofs. `https://gitlab.com/MIAOresearch/VeriPB`.
49 Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.
50 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 Internatjuional Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

A Derivations for Building Blocks

Before going into detail on the derivations and presenting their respective algorithms, the notation for the proof logging is described. This is similar to the notation of the proof file used by *VeriPB*.

Lines are added to the proof file using the proof_log($\cdot$) command. In this format, every constraint in the proof gets a unique *identifier* (or just *id* for brevity). We can

Algorithm 2 Deriving a unary sum over fresh variables z_i.

procedure derive_unary_sum(C')
  ▷ input: C' is of the form $\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} z_i$ and describes the constraint to be derived
  ▷ the z_i variables need to be fresh, the left hand side is the sum to be encoded
  for j from 1 to k **do**
    $D_j^{\text{geq}}, D_j^{\text{leq}} \leftarrow \text{Reify}(z_j \Leftrightarrow \sum_{i=1}^{n} 1 \cdot \ell_i \geq j)$ ▷ Step 1: introduce variables as reification
  $C^{\text{geq}} \leftarrow \text{deriveSum}(D_1^{\text{geq}}, D_2^{\text{geq}}, \ldots, D_n^{\text{geq}})$ ▷ Step 2: derive $\sum_{i=1}^{n} \ell_i \geq \sum_{i=1}^{n} z_i$
  $C^{\text{leq}} \leftarrow \text{deriveSum}(D_n^{\text{leq}}, D_{n-1}^{\text{leq}}, \ldots, D_1^{\text{leq}})$ ▷ Step 3: derive $\sum_{i=1}^{n} \ell_i \leq \sum_{i=1}^{n} z_i$
  for i from 1 to $k-1$ **do**
    DeriveOrdering(D_i^{leq}, D_{i+1}^{geq}) ▷ Step 4: derive $z_i \geq z_{i+1}$, $i \in [n-1]$
  return $C^{\text{geq}}, C^{\text{leq}}$

Algorithm 3 Reify $\sum_{i=1}^{n} a_i \ell_i \geq j$ using the fresh variable z_j.

procedure reify($z_j \Leftrightarrow \sum_{i=1}^{n} a_i \ell_i \geq j$)
  $C^{\text{geq}} \leftarrow \sum_{i=1}^{n} a_i \ell_i + j \bar{z}_j \geq j$ ▷ $z_j \Rightarrow \sum_{i=1}^{n} a_i \ell_i \geq j$ in normalized form
  proof_log(`red` C^{geq} `;` z_j `-> 0`)
  $C^{\text{leq}} \leftarrow \sum_{i=1}^{n} a_i \bar{\ell}_i + (\sum_{i=1}^{n} a_i - j + 1) z_j \geq \sum_{i=1}^{n} a_i - j + 1$ ▷ $z_j \Leftarrow \sum_{i=1}^{n} a_i \ell_i \geq j$ in normalized form
  proof_log(`red` C^{leq} `;` z_j `-> 1`)
  return $C^{\text{geq}}, C^{\text{leq}}$

express cutting planes derivations in reverse polish notation where constraints are referred to by their ids. For example, given previously derived constraints C and D, the line ‘proof_log(`pol` C D `+ 3 * 4 d`)’ adds C and D, multiplies the result by 3, and finally divides by 4 (rounding up). In the concrete format constraints in reverse polish notation are represented by an identifier, but we omit this detail for simplicity and operate on the constraints directly. The proof format also supports the *saturation* rule, which, given a normalized constraint $\sum_i a_i \ell_i \geq A$, allows to derive $\sum_i \min(a_i, A) \ell_i \geq A$. We use ‘proof_log(`pol` C `s`)’ to denote saturation in the proof format.

A RUP constraint C can be added using ‘proof_log(`rup` C)’. The syntax for adding a constraint as reification is ‘red $z \Rightarrow C$; z 1’ and ‘red $z \Leftarrow C$; z 0’, respectively (for more details please refer to [27]).

A.1 Deriving the Unary Sum

Deriving the constraints of a unary sum over fresh variables z_j, i.e.,

$$\sum_{i=1}^{n} \ell_i \geq \sum_{i=1}^{n} z_i \quad , \tag{21a}$$
$$\sum_{i=1}^{n} \ell_i \leq \sum_{i=1}^{n} z_i \quad , \text{ and} \tag{21b}$$
$$z_i \geq z_{i+1} \quad i \in [n-1] \quad , \tag{21c}$$

is described in Algorithm 2, which is split into four steps. **Step 1** is to introduce the fresh variables z_j as reifications of the constraints $\sum_{i=1}^{n} \ell_i \geq j$, which is shown in Algorithm 3 for the more general case using arbitrary positive coefficients.

Step 2: Deriving the Lower Bound. To derive (21a) in Algorithm 4 we maintain the invariant $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \bar{z}_i \geq j$, which holds by induction. For $j = 1$ the invariant

Algorithm 4 Derive sum of reification variables.

procedure deriveSum($D_1, \ldots, D_n$)
  ▷ input: D_j is of the form $\sum_{i=1}^{n} \ell_i + j\bar{z}_j \geq j$
  $C \leftarrow D_1$
  for j from 2 to n **do** ▷ Invariant: $C : \sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \bar{z}_i \geq j$
    proof_log(`pol` C $j-1$ `*` D_j `+` j `d`)
    $C \leftarrow ((j-1) \cdot C + D_j)/j$
  return C

Algorithm 5 Deriving an ordering constraint $z_A \geq z_B$ from the reification constraints.

procedure DeriveOrdering(C, D)
  ▷ input: C is of form $z_A \Rightarrow \sum_{i=1}^{n} a_i\ell_i \geq A$
  ▷ input: D is of form $z_B \Leftarrow \sum_{i=1}^{n} a_i\ell_i \geq B$
  $divisor \leftarrow \sum_{i=1}^{n} a_i$
  ▷ derive $z_A \geq z_B$ if $A < B$
  proof_log(`pol` C D `+` $divisor$ `d`)

is equivalent to the reification constraint $z_1 \Rightarrow \sum_{i=1}^{n} \ell_i \geq 1$, which in normalized form is $\sum_{i=1}^{n} \ell_i + \bar{z}_1 \geq 1$ and hence the base case is covered. For the inductive step going from j to $j+1$, we multiply the invariant by j and add the reification constraint $z_{j+1} \Rightarrow \sum_{i=1}^{n} \ell_i \geq j+1$, which is $\sum_{i=1}^{n} \ell_i + (j+1)\bar{z}_{j+1} \geq j+1$ in normalized form, to get $(j+1)\sum_{i=1}^{n} \ell_i + j\sum_{i=1}^{j} \bar{z}_i + (j+1)\bar{z}_{j+1} \geq j^2 + j + 1$. Note that $j^2 + j + 1 = (j+1)^2 - j$ and hence division by $j+1$ and rounding up yields $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \bar{z}_i + \bar{z}_{j+1} \geq j+1$, i.e., the invariant for $j+1$. For $j = k+1$ the invariant is the normalized form of (21a).

Step 3: Deriving the Upper Bound. To derive (21b) we can use Algorithm 4 again but need to provide the constraints in reverse order to fit the required input format.

Step 4: Deriving the Ordering Constraints. The ordering constraint is derived in Algorithm 5, using the reification constraints: We add the constraints used for reification, that is $z_{j+1} \Rightarrow \sum_{i=1}^{n} a_i\ell_i \geq j+1$ and $z_j \Leftarrow \sum_{i=1}^{n} a_i\ell_i \geq j$. In normalized form these two constraints are $(j+1)\bar{z}_{j+1} + \sum_{i=1}^{n} a_i\ell_i \geq j+1$ and $(m-j+1)z_j + \sum_{i=1}^{n} a_i\bar{\ell}_i \geq m-j+1$, where $m = \sum_{i=1}^{n} a_i$. Adding both constraints together yields $(m-j+1)z_j + (j+1)\bar{z}_{j+1} \geq 2$ and we get the desired ordering constraint after division by a large enough number, e.g., m.

A.2 Deriving the Sparse Unary Sum

In this section we prove Proposition 6 by providing Algorithm 6, which derives the sparse unary sum of two numbers in sparse unary representation. As for the unary sum, we start in Step 6.1 by introducing the required fresh variables via reification. However, we only need to introduce the variables that will be used, i.e., those with index in E. If k-simplification is used, then also variables with index bigger than k need to be introduced, as without them equality cannot be derived. (The introduction of variables with index bigger than k can be avoided by having a arithmetic graph each for the upper and lower bound and relaxing the preserving equality to inequalities.) After introducing the variables we can derive the ordering constraints as before.

In Step 6.2 we introduce a variable z_{eq} which is true if and only if the equality to be derived is true. Note that we need to represent an equality as two inequalities and hence need to introduce separate variables z_{geq}, z_{leq} for each inequality and then combine them into z_{eq}.

In Step 6.3 we derive $z_{eq} \geq 1$ by checking all combinations of values in A and B, which requires $O(|A| \cdot |B|)$ steps. Note that asymptotically this is the same number of steps as is required to compute which elements are in E so this step is still linear in the time needed to construct the encoding.

In Step 6.4 we use that $z_{eq} \geq 1$ and hence $z_{geq} = z_{leq} = 1$, which allows us to derive $\mathit{sparse}(\vec{y}, A) + \mathit{sparse}(\vec{y}\,', B) \geq \mathit{sparse}(\vec{z}, E)$ and $\mathit{sparse}(\vec{y}, A) + \mathit{sparse}(\vec{y}\,', B) \leq \mathit{sparse}(\vec{z}, E)$ respectively by removing z_{geq}, z_{leq} from the constraints introduced in Step 6.2.

Algorithm 7 describes in detail how to derive $z_{eq} \geq 1$ by checking all combinations of values in A and B. Let us illustrate how the algorithm works with an example. Let $A = \{0, 2\}$ and $B = \{0, 2, 4\}$. After the first iteration of the outer loop the algorithm derives the clauses

$$y_2 + y'_2 + z_{eq} \geq 1 \ , \tag{22a}$$

$$y_2 + \overline{y}'_2 + y'_4 + z_{eq} \geq 1 \ , \text{ and} \tag{22b}$$

$$y_2 + \overline{y}'_4 + z_{eq} \geq 1 \ . \tag{22c}$$

Note that deriving (22a) by reverse unit propagation sets $y_2 = y'_2 = z_{eq} = 0$. This causes the ordering constraints to propagate all variables in $\vec{y}$ and $\vec{y}\,'$. As all $\vec{y}$ and $\vec{y}\,'$ variables are set, the reification constraints introduced in Step 6.1 will cause all $\vec{z}$ variables to propagate. As the constraints reified in Step 6.2 are now satisfied we also get the propagation $z_{geq} = z_{leq} = 1$ and hence z_{eq} should be set to 1 as well. However, we already set z_{eq} to 0 and hence have a contradiction showing that (22a) can be derived. Deriving the other clauses works analogously.

If we add all clauses in (22) together, then y'_2 and y'_4 get canceled out and we are left with $3y_2 + 3z_{eq} \geq 1$ which is saturated to obtain $y_2 + z_{eq} \geq 1$. Analogously, the second iteration of the outer loop derives $\overline{y}_2 + z_{eq} \geq 1$, which added to the result of the first iteration yields $2z_{eq} \geq 1$ and using saturation we obtain $z_{eq} \geq 1$ as desired.

A.3 Derivation for binary adder encoding

This section provides the algorithm for constructing the adder network in Algorithm 8 and the proof logging and derivation of the preserving equality (17) from Proposition 7 for a single binary full adder in Algorithm 9.

B Totalizer and Generalized Totalizer Encoding

The totalizer and generalized totalizer encoding accumulate the input in form of a balanced binary tree. The totalizer encoding is designed for encoding cardinality constraints and uses the order encoding to represent values, while the generalized totalizer is designed for general pseudo-Boolean constraints and uses a sparse representation. An example of an arithmetic graph for the generalized totalizer encoding is shown in Figure 7. This graph contains a leaf node for each of the variables in the encoded constraint (to obtain a unique source we simply combine all leaf nodes into one node). The leaf nodes are combined in form of a binary tree, where we ensure that the value is preserved for each inner node, i.e., each possible value of incoming edges is representable as value of the outgoing edges. To perform k-simplification the arithmetic graph has

Algorithm 6 Deriving a sparse unary sum over fresh variables $\vec{z}$.

procedure derive_sparse_unary_sum(C')
  ▷ input: C' is of the form $sparse(\vec{y}, A) + sparse(\vec{y}\,', B) = sparse(\vec{z}, E)$ and describes the constraint to be derived such that $A, B \subseteq \mathbb{N}$, $E = \{\, i + j \mid i \in A, j \in B \,\}$ and $\vec{z}$ variables are fresh
  ▷ **Step 6.1**: introduce variables as reification and derive ordering
  for $j \in E \setminus \{\, 0 \,\}$ **do**
    $D_j^{\text{geq}}, D_j^{\text{leq}} \leftarrow \text{reify}(z_j \Leftrightarrow sparse(\vec{y}, A) + sparse(\vec{y}\,', B) \geq j)$
  for $i \in E \setminus \{\, 0, \max(E) \,\}$ **do**
    DeriveOrdering(D_i^{leq}, $D_{succ(i,E)}^{\text{geq}}$) ▷ derive $z_i \geq z_{succ(i,E)}$
  ▷ **Step 6.2**: : reify constraint to be derived
  $C^{\text{geq}}, \_ \leftarrow \text{reify}(z_{geq} \Leftrightarrow sparse(\vec{y}, A) + sparse(\vec{y}\,', B) \geq sparse(\vec{z}, E))$
  $C^{\text{leq}}, \_ \leftarrow \text{reify}(z_{leq} \Leftrightarrow sparse(\vec{y}, A) + sparse(\vec{y}\,', B) \leq sparse(\vec{z}, E))$
  $\text{reify}(z_{eq} \Leftrightarrow z_{geq} + z_{leq} \geq 2)$
  ▷ **Step 6.3**: derive that $z_{eq} \geq 1$
  try_all_values($sparse(\vec{y}, A), sparse(\vec{y}\,', B), z_{eq}$)
  ▷ **Step 6.4**: derive constraint to be derived from its reification
  $M \leftarrow \max(A) + \max(B)$ ▷ Coefficient so that reification variables get eliminated.
  $D \leftarrow z_{geq} \geq 1$
  proof_log(`rup` D)
  proof_log(`pol` C^{geq} D M `* +`)
  $C^{\text{geq}} \leftarrow C^{\text{geq}} + M \cdot D$
  $D \leftarrow z_{leq} \geq 1$
  proof_log(`rup` D)
  proof_log(`pol` C^{leq} D M `* +`)
  $C^{\text{leq}} \leftarrow C^{\text{leq}} + M \cdot D$
  return $C^{\text{leq}}, C^{\text{leq}}$

additional edges that go directly into the sink node. The formal definition of arithmetic graph for the (generalized) totalizer encoding is as follows.

▶ **Definition 8** (Arithmetic graph for the generalized totalizer encoding). *Given a linear sum $\sum_i a_i x_i$ over n variables, let G be a binary tree with edges directed towards the root r, leaves s_i for $i \in [n]$ and an additional sink node t with an edge (r, t). In what follows we will consider r as an inner node. The edge (s_i, v) from the leave s_i is labeled with $a_i x_i$, which can be viewed as a sparse representation for values $\{\, 0, a_i \,\}$. For an inner node v with two incoming edges with labels $sparse(\vec{y}, A)$ and $sparse(\vec{y}\,', B)$, the outgoing edge e is labeled $sparse(\vec{z}, E)$, where $\vec{z}$ are fresh variables and $E = \{\, i + j \mid i \in A, j \in B \,\}$. To obtain a graph with a single source we combine all s_i into a single node s. To perform k-simplification we split $sparse(\vec{z}, E) = \sum_{i \in E} a_i z_i$ into $\sum_{i \leq succ(k,E)} a_i z_i$, which is the label of the outgoing edge e, and $\sum_{i > succ(k,E)} a_i c_i$, which is the label for an addition outgoing edge $e' = (v, t)$.*

To see that the defined graph is an arithmetic graph, we only need to check that we can derive the preserving equality for each inner node. Each inner node has two incoming edges that are labeled with a sparse unary representation and all outgoing edges together form a sparse unary representation as well, so that we can use Proposition 6 to derive the required preserving equality. Note that Proposition 6 also requires to have ordering

Algorithm 7 Given a reified sparse unary sum, derive that the reification variable is true.

procedure fix($sparse(\vec{y}, A), a$)
  return $\overline{y}_a + y_{succ(a,A)}$ ▷ replace y_0 by 1 and y_∞ by 0
procedure try_all_values($sparse(\vec{y}, A), sparse(\vec{y}\,', B), z_{eq}$)
  $C_{outer} \leftarrow 0 \geq 0$
  for $i \in A$ **do**
    $C_{inner} \leftarrow 0 \geq 0$
    for $j \in B$ **do**
      ▷ assuming that a (respectively b) is the value encoded by $sparse(\vec{y}, A)$ ($sparse(\vec{y}\,', B)$)
      ▷ encode that $(a = i \wedge b = j) \Rightarrow z_{eq}$
      $D \leftarrow fix(sparse(\vec{y}, A), i) + fix(sparse(\vec{y}\,', B), j) + z_{eq} \geq 1$
      proof_log(`rup` D)
      proof_log(`pol` C_{inner} D `+`)
      $C_{inner} \leftarrow C_{inner} + D$
    proof_log(`pol` C_{outer} C_{inner} `s +`)
    $C_{outer} \leftarrow C_{outer} + \mathsf{saturate}(C_{inner})$
  proof_log(`pol` C_{outer} `s`)
  $C_{outer} \leftarrow \mathsf{saturate}(C_{outer})$
  return C_{outer} ▷ C_{outer} is now $z_{eq} \geq 1$

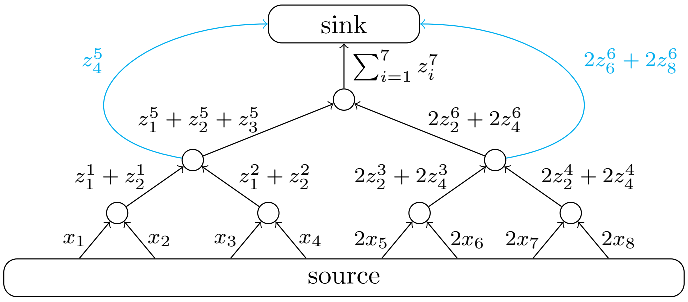


Figure 7 Layout of the arithmetic graph for the generalized totalizer encoding of $x_1 + x_2 + x_3 + x_4 + 2x_5 + 2x_6 + 2x_7 + 2x_8 \leq 2$. Edges introduced for k-simplification are colored cyan.

constraints on the input variables, however, it is easy to see by an inductive argument that the ordering constraints on the variables will be present, when processing the graph in topological order: Edges from the source only contain a single variable and hence the ordering constraints exist trivially. For inner nodes we get the ordering constraints by applying Proposition 6.

If the set of achievable values E is dense for some node, i.e., E contains all values from 0 to $\max(E)$, then we can also use Proposition 5 to derive the required preserving equality, which only requires $O(|E|)$ instead of $O(|A| \cdot |B|)$ steps and hence can reduce the proof logging overhead.

For each inner node in the graph with incoming edge labels $sparse(\vec{y}, A)$ and $sparse(\vec{y}\,', B)$, the (generalized) totalizer encoding contains the clauses

$$\overline{y}_i + \overline{y}'_j + z_{i+j} \geq 1 \quad \text{for } i \in A, j \in B \tag{23a}$$

$$y_{succ(i,A)} + y'_{succ(j,B)} + \overline{z}_{succ(i+j,E)} \geq 1 \quad \text{for } i \in A, j \in B \text{ s.t. } i + j \tag{23b}$$

Algorithm 8 Construction of the binary adder network [18].

procedure adder_network(b)
  ▷ input: vector of buckets b
  for i from 0 to b.size() **do**
    while b_i.size()≥ 2 **do**
      if b_i.size()$= 2$ **then**
        $x, y \leftarrow b_i$.dequeue()
        $c, s \leftarrow$ full_adder$(x, y, 0)$
      else
        $x, y, z \leftarrow b_i$.dequeue()
        $c, s \leftarrow$ full_adder(x, y, z)
      b_i.enqueue(s)
      b_{i+1}.enqueue(c)

Algorithm 9 Proof logging the encoding of a single full adder.

procedure full_adder(x, y, z)
  $D^{\text{geq}}_{carry}, D^{\text{leq}}_{carry} \leftarrow$ reify$(c \Leftrightarrow x + y + z \geq 2)$
  $D^{\text{geq}}_{sum}, D^{\text{leq}}_{sum} \leftarrow$ reify$(s \Leftrightarrow x + y + z + 2\bar{c} \geq 3)$
  $D^{\text{geq}} \leftarrow (2 \cdot D^{\text{geq}}_{carry} + D^{\text{geq}}_{sum})/3$
  proof_log(`pol` D^{geq}_{carry} `2 *` D^{geq}_{sum} `+ 3 d`)
  $D^{\text{leq}} \leftarrow (2 \cdot D^{\text{leq}}_{carry} + D^{\text{leq}}_{sum})/3$
  proof_log(`pol` D^{leq}_{carry} `2 *` D^{leq}_{sum} `+ 3 d`)
  return D, c, s ▷ D is the preserving equality of the full adder

where $succ(i, A) = \min(\{ j \mid j \in A \cup \{ \infty \}, j > i \})$ and we replace y_0, y'_0 with 1, and $y_\infty, y'_\infty, z_\infty$ with 0 and simplify accordingly. Note that, (23) encodes that $a + b = c$ (where a and b are the incoming values and c is the output value), because (23a) encodes that if $a \geq i$ (expressed by assigning y_i to 1) and $b \geq j$ then $c \geq i+j$ while (23a) encodes that if $a \leq i$ (which is the same as saying that $a < succ(i, A)$, expressed by assigning $y_{succ(i,A)}$ to 0) and $b \leq j$ then $c \leq i + j$.

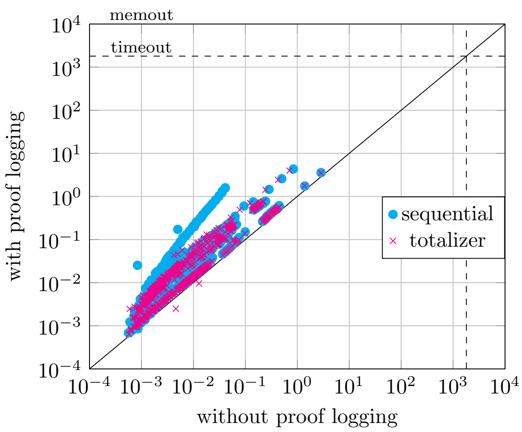
For proof logging the CNF encoding we can simply add all clauses using RUP: A RUP check of (23a) will assign $y_i = y'_j = 1$ and $z_{i+j} = 0$. The ordering constraints on $\vec{y}, \vec{y}\,'$ will cause a propagation setting multiple $\vec{y}, \vec{y}\,'$ variables to true such that $sparse(y, A) + sparse(y', B)$ has a value of at least $i+j$, while the ordering constraints on $\vec{z}$ will propagate multiple $\vec{z}$ to false such that $sparse(z, E)$ can only take a value that is strictly less than $i+j$ and hence causes a conflict with the preserving equality $sparse(z, E) = sparse(y, A) + sparse(y', B)$. Similarly, a RUP check of (23b) will assign $y_{succ(i,A)} = y'_{succ(j,B)} = 0$ and $z_{succ(i+j,E)} = 1$ causing propagations such that $sparse(y, A) + sparse(y', B)$ takes a value less than or equal to $i + j$ and $sparse(z, E)$ takes a value strictly greater than $i + j$ causing again a conflict with the preserving equality.

To enforce a pseudo-Boolean constraint $\sum_i a_i x_i \bowtie k$, we first derive a bound on the output of the arithmetic graph $\sum_i c_i o_i \bowtie k$, using Proposition 4. Then we can derive unit clauses on the output via reverse unit propagation.

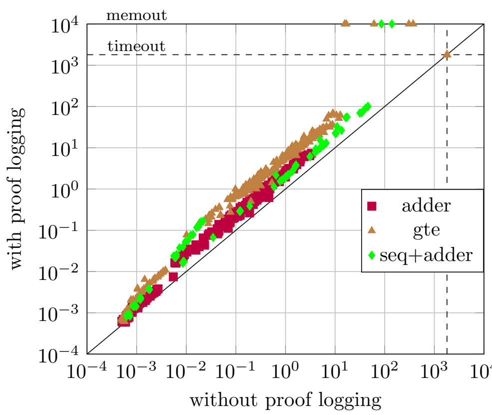
To encode $\sum_i a_i x_i \geq k$ or $\sum_i a_i x_i \leq k$ the clause $z_{succ(k-1,E)} \geq 1$ or $\bar{z}_{succ(k,E)} \geq 1$ is added, respectively. This clause is RUP, as the derived sum $\sum_i c_i o_i$ has a value of at most $k - 1$ or at least $k + 1$ and thus the constraint $\sum_i c_i o_i \geq k$ or $\sum_i c_i o_i \leq k$ is falsified, respectively. To encode $\sum_i a_i x_i = k$ both clauses are added.

Table 2 Properties of pseudo-Boolean formulas used in the experimental results.

		Card	PB	Card+PB
	#Inst.	772	442	308
Card	Avg. #	107.01±252.57	0.00	1,154.43±5,881.78
	Avg. #Lits	36.45±47.43	0.00	16.96±26.57
	Avg. Coeff. Size	1.00±0.00	0.00	1.00±0.00
PB	Avg. #	0.00	1,020.73±2,294.43	33,379.31±18,3229.66
	Avg. #Lits	0.00	24.95±27.60	105.21±109.99
	Avg. Coeff. Size	0.00	204.93±1,118.74	10.79±50.42



(a) Cardinality formulas



(b) General pseudo-Boolean formulas

Figure 8 Comparison of runtimes between CNF translation with and without proof logging.

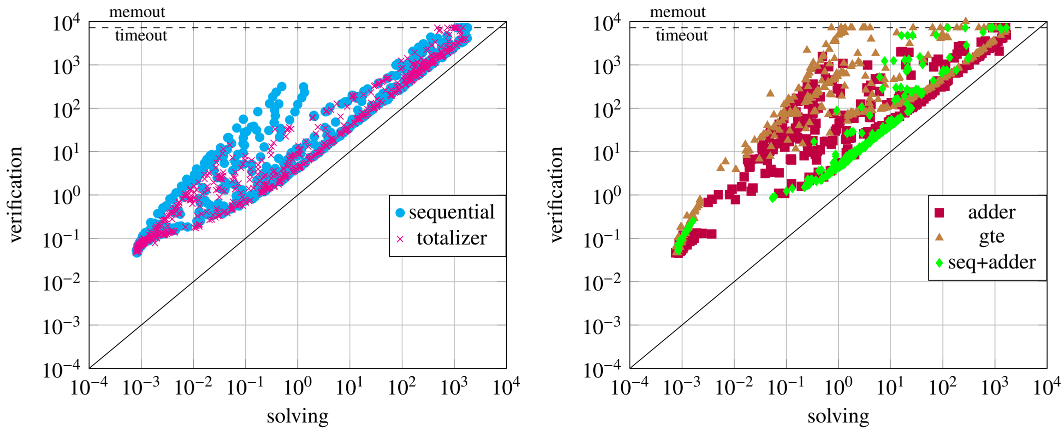
C Additional Evaluation Data

C.1 Benchmarks

Table 2 shows some properties of the benchmarks used in the experimental results, namely, the average number of cardinality constraints (Card), the average number of literals in each constraint, and the average size of coefficients associated with each literal. (The same is shown for PB constraints.) Since the benchmark set is composed of instances from multiple domains, there is a large dispersion of values between instances. For example, the number of cardinality constraints for instances in the *Card* benchmark set ranges from 1 to 2,720. Whereas the number of PB constraints for instances in the *PB* benchmark set ranges from 1 to 18,798. In the *Card+PB* benchmark set, we have an even larger dispersion with instances that have from 1 to 2,378,901 PB constraints and from 1 to 75,582 cardinality constraints.

C.2 Overhead of Proof Logging

Figure 8 shows the overhead of proof logging when translating the pseudo-Boolean formulas to CNF. For the majority of the instances, the overhead is not too significant, and formulas with just cardinality constraints can still be translated under 10 seconds, while formulas with PB constraints can be translated under 100 seconds. The exception



(a) Cardinality formulas

(b) General pseudo-Boolean formulas

Figure 9 Comparison between end-to-end solving and verification time

are the cardinality formulas from vertex cover that require super linear proofs, which lead to a higher overhead when storing the proof. Additionally, there were 6 instances that had memory outs when storing the proof in memory, which could be improved in the future by a more compact representation of the proof logging in VeritasPBLib.

C.3 Solving and Verification

Figure 9 shows the relationship between the time to generate the CNF translation and solve it using *kissat* and the time to verify the translation and solution using *VeriPB*. It can be seen that even though we can verify most instances, verification is often considerably slower than solving.

A lot of instances are spread in a wide range of different overheads. This wide range only comes from verifying the solution, which is out of the scope of this work. However, it motivates potential improvements to *VeriPB* which are complementary to the work proposed in this paper and can further increase the number of verified instances.