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 fact that SAT proof logging is performed in conjunctive normal form (CNF) clausal format 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 an unverified translation of 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, certifying algorithms, certified translation, CNF encoding, cutting planes

Digital Object Identifier 10.4230/LIPIcs.SAT.2022.16

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) solving has witnessed striking improvements over the last couple of decades, starting with the introduction of *conflict-driven clause learning (CDCL)* [44, 48], and this has led to a wide range of applications including large-scale problems in both academia and industry [9]. The conflict-driven paradigm has also been successfully exported

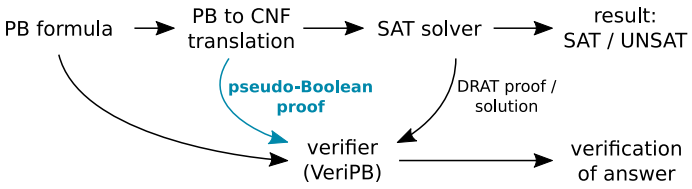


Figure 1 Proof logging workflow for pseudo-Boolean solving (our contribution in blue boldface).

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 heuristics and optimizations 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, 17, 29]. For SAT solvers, however, analogous problems [11] have been successfully addressed by the introduction of *proof logging*, requiring that solvers should be *certifying* [46] 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 solving, including *RUP* [34, 58], *TraceCheck* [8], *DRAT* [35, 36, 62], *GRIT* [19], and *LRAT* [18]. Since 2013 the SAT competitions [56] 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 limitation to a clausal format makes it hard to capture more advanced forms of reasoning in a succinct way. A more fundamental concern is that it is not clear how these proof logging methods should deal with input that is not presented 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 more expressive formalisms such as binary decision diagrams [4], algebraic reasoning [39, 40, 41, 53], pseudo-Boolean reasoning [26, 31, 32], and integer linear programming [15, 24].

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+ [23]. The two solvers NaPS [49, 55] and Open-WBO [50, 45] using this approach were among the top performers in the latest pseudo-Boolean evaluation [52]. While *DRAT* proof logging can certify unsatisfiability of the translated formula, it cannot prove 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 [13]), but also, and more fundamentally, because the input is not in CNF.

We demonstrate how to instead use the *cutting planes* proof method [16], enhanced with a rule for introducing extension variables [33], to show that the CNF formula resulting from the translation can be derived from the original pseudo-Boolean constraints. Since this method is a strict extension of *DRAT*, we can combine the proof for the translation with the SAT solver *DRAT* proof log (with appropriate syntactic modifications). In this way we achieve end-to-end verification of the pseudo-Boolean solving process using the proof checker VeriPB [60], as illustrated 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., [51]), 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 formula enforcing the computation of this circuit. An important part of our contribution is that we develop a general proof logging method for a wide class of such circuits. The pseudo-Boolean format used for proof logging makes it easy to derive 0-1 linear inequalities describing the circuit computations, and once this has been done the clauses in the CNF translation can simply be obtained by so-called *reverse unit propagation (RUP)* [34, 58], obviating the need for complicated syntactic proofs. We have applied this method to the *sequential counter* [57], *totalizer* [3], *generalized totalizer* [38] and *binary adder network* [23, 61] encodings, and report results from an empirical evaluation.

We note that a stronger result would be to certify *equivalence* of the original pseudo-Boolean formula F and the translated CNF formula F', in the sense that (a) any satisfying assignment α to F could be extended to an assignment α' also to the new variables introduced during translation that would satisfy F', and that (b) any satisfying assignment α' to F' also satisfies F. The tools we develop can reach this more ambitious goal in principle, but since some additional technical problems arise along the way we have to leave this as future work.

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 discuss how to apply it to adder networks in Section 5 (but due to space constraints, most of the details, and all discussion of the totalizer and generalized totalizer encodings, are deferred to Appendices A and B.) We report data from our experimental evaluation in Section 6 and conclude with a discussion of some 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., [14, 33]. A *literal* ℓ over a Boolean variable x is x itself or its negation $\overline{x}$, where variables can be assigned values 0 (false) or 1 (true), so that $\overline{x} = 1 - x$. For notational convenience, we define $\overline{\overline{x}} \doteq x$ (where we use $\doteq$ to denote syntactic equality). We write $[n] = \{1, 2, \dots, n\}$ to denote the n first positive integers, and sometimes write $\vec{x} = \{x_i \mid i \in [n]\}$ to denote a set of variables, where the size n of the set understood from context (or is not important). A *pseudo-Boolean (PB) constraint* is a 0-1 linear inequality

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

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 the constraint

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

(encoding that the sum of the coefficients of falsified literals in C is so large that coefficients of satisfied literals can contribute at most $A - 1$). We use equality constraints $C \doteq \sum_i a_i \ell_i = A$ as syntactic sugar for the pair of inequalities $C^{\text{geq}} \doteq \sum_i a_i \ell_i \geq A$ and $C^{\text{leq}} \doteq \sum_i -a_i \ell_i \geq -A$ (with the latter converted to normalized form). 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 $F = \bigwedge_j C_j$ of PB constraints. A *cardinality constraint* is a PB constraint with

all coefficients equal to 1. If the degree is also 1, then $\ell_1 + \cdots + \ell_k \geq 1$ is equivalent to the *(disjunctive) clause* $\ell_1 \vee \cdots \vee \ell_k$, and so CNF formulas are just special cases of PB formulas.

A *(partial) assignment* ρ is a (partial) function from variables to $\{0, 1\}$. Applying ρ to a constraint C as in (1) yields the constraint $C\restriction_\rho$ 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 $F\restriction_\rho = \bigwedge_j C_j\restriction_\rho$. The constraint C is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq A$ (or, equivalently, if the restricted constraint $C\restriction_\rho$ 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 [16] 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 derived. (By a linear combination of two equality constraints C and D, we mean the identical linear combinations of C^{geq} and D^{geq} and C^{leq} and D^{leq}, respectively.) We can also add *literal axioms* $\ell_i \geq 0$ to a previously derived constraint. For a constraint $\sum_i a_i \cdot \ell_i \geq A$ in normalized form, 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, and it is sometimes convenient to also include a *saturation* rule deriving $\sum_i \min\{a_i, A\} \cdot \ell_i \geq A$ from $\sum_i a_i \cdot \ell_i \geq A$. We remark that the soundness of the division and saturation rules as stated depends on the constraints being presented in normalized form.

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, and cutting planes is an *implicationally complete* method in the sense that any implied constraint can also be derived syntactically.

A constraint C is said to *unit propagate* the literal ℓ under ρ if $C\restriction_\rho$ cannot be satisfied unless ℓ is set to true. 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 $F \wedge \neg C$ 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, but the opposite direction is not necessarily true.

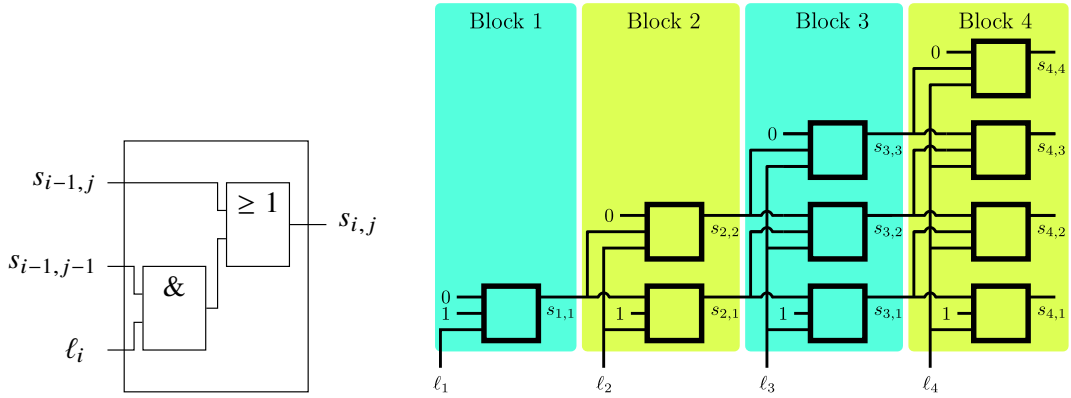
In addition to deriving constraints C that are implied by F, we will also need a way of adding so-called *redundant* constraints D having the property that F and $F \wedge D$ are equisatisfiable. For this purpose we will use the *reification* rules—special cases of the redundance rule in [33]—saying that we can introduce the *reified constraints*

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

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

provided that z is a *fresh variable* that is not in the formula and has not appeared previously in the derivation.

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 the constraints (3a) and (3b).



(a) Logic circuit of single component. **(b)** Circuit for 4 input literals counting up to 4.

Figure 2 Circuit representation of the sequential counter encoding.

Adding such reification constraints preserves equisatisfiability, since any satisfying assignment to F can be extended by setting the fresh variable z as required to satisfy the implications.

3 Certified CNF Translation Using the Sequential Counter Encoding

To give a concrete illustration of our approach for proving the correctness of translations of pseudo-Boolean constraints, in this section we consider how to convert cardinality constraints $\sum_{i=1}^{n} \ell_i \bowtie k$ to CNF using the *sequential counter* encoding [57]. This encoding is based on a circuit summing up the input bits one by one, with intermediate variables $s_{i,j}$ for $i \in [n]$ and $j \in [i]$ evaluating to true if and only if $\sum_{t=1}^{i} \ell_t \geq j$. The variables $s_{i,j}$ can be computed inductively as in Figure 2a by the formula

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

saying that $s_{i,j}$ is true either if the first $i-1$ literals add up to $j-1$ and the ith literal is true, or if already the first $i-1$ literals add up to j. The circuit constructed in this way, shown in Figure 2b, can be partitioned into n blocks, where the ith block computes the variables $s_{i,j}$ for $j \in [i]$ from the ith input bit ℓ_i and the variables $s_{i-1,j}$ in the previous block. Identifying such blocks in the circuit is a key component in our method for proving that the CNF translation is correct.

For the sequential counter circuit, we obtain the CNF encoding of the constraint $\sum_{i=1}^{n} \ell_i \bowtie k$ by translating each component in Figure 2a (as described by Equation (4)) to the clausal constraints

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

for $i \in [n]$ and $j \in [i]$. For all i we set $s_{i,0} = 1$ and simplify, so that constraint (5a) turns into $\bar{\ell}_i + s_{i,1} \geq 1$ and constraint (5d) is satisfied and disappears. We also set $s_{i-1,i} = 0$, so that (5c) becomes $\ell_i + \bar{s}_{i,i} \geq 1$ and (5b) is satisfied and disappears.

Once clauses (5a)–(5d) have been generated for all circuit components, we obtain a greater-than-or-equal-to-k constraint by adding the unit clause $s_{n,k} \geq 1$. Analogously,

a less-than-or-equal-to-k constraint is enforced using the clause $\bar{s}_{n,k+1} \geq 1$. A common optimization, known as *k-simplification*, is to omit clauses corresponding to the computation of variables $s_{i,j}$ for $j > k+1$, as such variables are not relevant for deciding whether the cardinality constraint is true or not.

As a preparation for our proof logging discussions, let us study the variables $s_{i,j}$ in more detail, ignoring k-simplification for now. Since $s_{i,j}$ is true if and only if $\sum_{t=1}^{i} \ell_t \geq j$ holds, for all $i \in [n]$ we should be able to deduce

$$\textstyle\sum_{j=1}^{i} \ell_j = \sum_{j=1}^{i} s_{i,j} \,. \tag{6}$$

However, the sequential counter circuit computes the variables $s_{i,j}$ in the ith block using only the variables $s_{i-1,j}$ from the previous block and the literal ℓ_i, and so if we only reason locally about the ith block what we can derive is the equality

$$\textstyle\ell_i + \sum_{j=1}^{i-1} s_{i-1,j} = \sum_{j=1}^{i} s_{i,j} \,. \tag{7}$$

If we look at the variables on wires entering and exiting the ith block of the circuit, we see that Equation (7) specifies that the sum of the inputs is equal to the sum of the outputs. If we represent the circuit in Figure 2b as a graph with every block contracted into a single node and the literals ℓ_i in the cardinality constraint collected into another separate node, then every ith block node has an incoming edge from the literals node and (for $i > 1$) another edge from the $(i-1)$st block node, and an outgoing edge to the $(i+1)$st block node (or, for $i = n$, to a special sink node that we can also introduce). If we label the incoming edges by ℓ_i and $\sum_{j=1}^{i-1} s_{i-1,j}$ and the outgoing edge by $\sum_{j=1}^{i} s_{i,j}$, as shown in Figure 3a, then we can view (7) as saying that for all vertices in the graph the sum of the labels of input edges should be equal to the sum of the output edge. We will refer to this as a *preservation equality*. What is not at all obvious from this particular example, but what we will show in later sections, is that many CNF translations of pseudo-Boolean constraints can be represented as graphs with preservation equalities in a similar way, though sometimes with larger coefficients in the linear combinations of the literals. And, jumping ahead a bit, our main contribution in this paper is a generic proof logging method that will certify correctness for any CNF encoding that can be represented in this graph framework with preservation equalities.

To see how such a graph representation might be useful, note that by traversing the graph and doing a telescoping sum of the preservation equalities for all nodes we can easily derive (6). From this, in turn, it is clear that a constraint on the input variables $\sum_{j=1}^{n} \ell_i \bowtie k$ implies the same constraint on the output variables, and formally this can be obtained by one final telescoping sum step combining $\sum_{j=1}^{n} \ell_i \bowtie k$ and $\sum_{j=1}^{n} \ell_i = \sum_{j=1}^{n} s_{n,j}$ to get

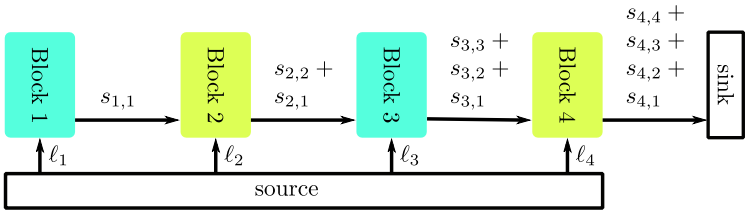
$$\textstyle\sum_{j=1}^{n} s_{n,j} \bowtie k \,. \tag{8}$$

Another important property of the variables $s_{i,j}$ is that they do not just take any values satisfying (7), but are ordered—since $s_{i,j}$ encodes $\sum_{t=1}^{i} \ell_t \geq j$, it follows that $s_{i,j}$ cannot be true unless also $s_{i,j'}$ is true for all $j' < j$. This can be expressed by *ordering constraints*

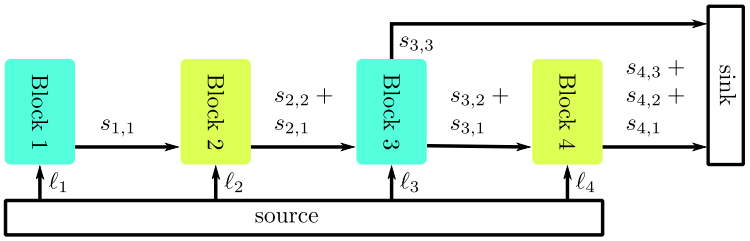
$$s_{i,j} \geq s_{i,j+1} \qquad\qquad i \in [n] \,, j \in [i-1] \,, \tag{9}$$

which are semantically implied by the circuit encoding.

Taking this view of the circuit encoding, the task of certifying the correctness of the CNF translation becomes surprisingly simple. If we can derive the pseudo-Boolean constraints (7)–(9), then it can be verified that the clauses of the sequential counter encoding (i.e., (5a)–(5d) plus $\bar{s}_{n,k+1} \geq 1$ and/or $s_{n,k} \geq 1$) all follow by reverse unit propagation. This



(a) Graph without k-simplification.



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

Figure 3 Graph representation of the sequential counter encoding.

is so since when asserting the clauses to false, the ordering constraints (9) will propagate enough variables $s_{i,j}$ for (7) to be falsified.

To see how to obtain the constraints (7)–(9), note that we already discussed above how to derive (8) by a telescoping sum over constraints (7), which is straightforward to do with standard cutting planes rules. To get constraints on the form (7), we can use reification to define the meaning of the variables $s_{i,j}$ by constraints

$$s_{i,j} \Leftrightarrow \ell_i + \sum_{j=1}^{i-1} s_{i-1,j} \geq j \tag{10}$$

(with notation as introduced in (3a)–(3b) in Section 2). If we do this in increasing order for i and j, then $s_{i,j}$ is fresh in (10) and so these are valid derivation steps. From the constraints (10) we can then derive (7) and (9) as illustrated in the next example.

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

$$\ell_3 + s_{2,1} + s_{2,2} = s_{3,1} + s_{3,2} + s_{3,3} \tag{11}$$

for block 3 in Figure 3a. We want $s_{3,j}$ to be true precisely when $\ell_3 + s_{2,1} + s_{2,2} \geq j$ for $j = 1, 2, 3$, and we can enforce this by reification steps as in (10), which results in constraints

$$\overline{s}_{3,1} + \ell_3 + s_{2,1} + s_{2,2} \geq 1 \tag{12a}$$
$$2\overline{s}_{3,2} + \ell_3 + s_{2,1} + s_{2,2} \geq 2 \tag{12b}$$
$$3\overline{s}_{3,3} + \ell_3 + s_{2,1} + s_{2,2} \geq 3 \tag{12c}$$
$$3s_{3,1} + \overline{\ell}_3 + \overline{s}_{2,1} + \overline{s}_{2,2} \geq 3 \tag{12d}$$
$$2s_{3,2} + \overline{\ell}_3 + \overline{s}_{2,1} + \overline{s}_{2,2} \geq 2 \tag{12e}$$
$$s_{3,3} + \overline{\ell}_3 + \overline{s}_{2,1} + \overline{s}_{2,2} \geq 1 \tag{12f}$$

when written out explicitly in pseudo-Boolean form.

By design, the constraints (12a)–(12f) implies (11). By the implicational completeness of cutting planes there is a derivation of this fact, i.e., of the two pseudo-Boolean inequalities

$$\overline{s}_{3,1} + \overline{s}_{3,2} + \overline{s}_{3,3} + \ell_3 + s_{2,1} + s_{2,2} \geq 3 \tag{13a}$$
$$s_{3,1} + s_{3,2} + s_{3,3} + \overline{\ell}_3 + \overline{s}_{2,1} + \overline{s}_{2,2} \geq 3 \tag{13b}$$

encoding the equality (11). To construct such a derivation, we first derive (13a) by processing the constraints (12a)–(12c) in order while maintaining the invariant

$$\sum_{j=1}^{i} \bar{s}_{3,j} + \ell_3 + s_{2,1} + s_{2,2} \geq i\,, \tag{14}$$

where i is the number of processed constraints. We start with (12a), for which the invariant holds, and add (12b) and divide by 2 to obtain $\bar{s}_{3,1} + \bar{s}_{3,2} + \ell_3 + s_{2,1} + s_{2,2} \geq 2$. Multiplying the latter constraint by 2, adding it to (12c), and finally dividing by 3 results in the constraint (13a) as desired. The derivation of (13b) is completely analogous, except we process (12d)–(12f) in reverse order. That is, we first add (12f) and (12e) and divide by 2 to get $s_{3,1} + s_{3,2} + \bar{\ell}_3 + \bar{s}_{2,1} + \bar{s}_{2,2} \geq 2$, and then multiply this constraint by 2, add to (12d), and divide by 3 to obtain our target constraint (13b).

For the ordering constraints (9), which in normalized form are written as

$$s_{3,j} + \bar{s}_{3,j+1} \geq 1\,, \tag{15}$$

we first add (12d) and (12b) to get $3s_{3,1} + 2\bar{s}_{3,2} \geq 2$, which when divided by 3 becomes $s_{3,1} + \bar{s}_{3,2} \geq 1$. In the same way, adding (12e) and (12c) yields $2s_{3,2} + 3\bar{s}_{3,3} \geq 2$, which we divide by 3 to obtain $s_{3,2} + \bar{s}_{3,3} \geq 1$. This concludes our example.

To obtain the encoding with k-simplification, the most naive approach would be to simply omit the clauses enforcing correct values for the variables $s_{i,j}$ that are not used. However, this could incur a significant overhead in the proof logging when k is small, as we would always introduce $\mathrm{O}(n^2)$ intermediate variables instead of the $\mathrm{O}(kn)$ variables actually used in the final encoding. To avoid this overhead, we can introduce "overflow variables" $s_{i,k+2}$ that do not encode that the first i bits sum to $k+2$ but instead ensure that the equality

$$\ell_i + \sum_{j=1}^{k+1} s_{i-1,j} = \sum_{j=1}^{k+2} s_{i,j} \tag{16}$$

holds. To maintain the equality of sums over incoming and outcoming edges in our graph representation, we label the edge to the next block by $\sum_{j=1}^{k+1} s_{i,j}$ instead of $\sum_{j=1}^{i} s_{i,j}$, and introduce an additional edge going directly to the sink with the label $s_{i,k+2}$ (see Figure 3b). 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 $k+1$ variables on the right-hand side.

▶ **Example 2.** To apply k-simplification for $k = 1$ to Figure 3a, the output from block 3 to block 4 should only contain the sum of the two variables $s_{3,1} + s_{3,2}$. To preserve equality of the sums of inputs and outputs, we add an edge from block 3 to the sink labelled $s_{3,3}$ as in Figure 3b.

When using k-simplification, we can derive an analogue of (6) by a telescoping sum of all preservation equalities (16) yielding $\sum_{i=1}^{n} \left(\ell_i + \sum_{j=1}^{k+1} s_{i-1,j}\right) = \sum_{i=1}^{n} \left(\sum_{j=1}^{k+2} s_{i,j}\right)$, which simplifies to $\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} s_{i,k+2} + \sum_{j=1}^{k+1} s_{n,j}$.

4 A General Framework for Certifying CNF Translations

As discussed in the introduction, there is a rich selection of encodings of pseudo-Boolean constraints in CNF. In this section, we develop a unified framework to provide proof logging for a wide range of different translations. Our approach is to represent encodings as directed graphs with preservation equalities between the incoming and outgoing edges of each node, as in our example in Figure 3, so that all clauses in the encoding can be obtained by reverse

unit propagation from (telescoping sums over) these equalities. In this way, the whole proof logging task is reduced to considering a few generic ways of deriving preservation equalities. Let us start with a formal definition of the graph representation.

▶ **Definition 3** (Arithmetic Graph). *Let a_i, c_i be integers, ℓ_i Boolean literals, and o_i Boolean variables. An* arithmetic graph *with input $\sum_i a_i\ell_i$ and output $\sum_i c_i o_i$ is a directed multi-graph $G = (V, E)$ that satisfies the following conditions:*

1. *Every edge $e \in E$ has a label of the form $\sum_i b_i^e y_i^e$ for each edge $e \in E$, where b_i^e are integers and y_i^e Boolean variables.*
2. *There is a unique source node s that has only outgoing edges, and these edges are labelled by input literals ℓ_i in such a way that $\sum_i a_i\ell_i = \sum_{(s,v)=e\in E} \sum_i b_i^e y_i^e$.*
3. *There is a unique sink t that has only incoming edges, and these edges are labelled by output variables o_i in such a way that $\sum_i c_i o_i = \sum_{(v,t)=e\in E} \sum_i b_i^e y_i^e$.*
4. *For all other nodes, which we refer to as the* inner nodes*, the preservation equality*

$$\sum_{(u,v)=e\in E} \sum_i b_i^e y_i^e = \sum_{(v,w)=e\in E} \sum_i b_i^e y_i^e \,, \tag{17}$$

saying that the sum of the incoming edges equals the sum of the outgoing edges, can be derived using cutting planes with reification over variables on outgoing edges.

The rest of this section will be devoted to discussing how preservation equalities (17) can be derived for different types of pseudo-Boolean expressions. Before doing so, let us just note for the record that if we have an arithmetic graph for an encoding of a pseudo-Boolean constraint, then by a telescoping argument as in Section 3 we can derive that the same constraint applies to the output of the graph.

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

Proof. By item 4 in Definition 3, we can derive preservation equalities (17) for all inner nodes in the graph. By making a graph traversal and adding all these equalities together (i.e., adding separately all greater-than-or-equal constraints and all less-than-or-equal constraints, as explained in Section 2), we obtain $\sum_i a_i\ell_i = \sum_i c_i o_i$, and combining this with $\sum_i a_i\ell_i \bowtie k$ yields $\sum_i c_i o_i \bowtie k$ as desired. ◀

Once the bound on the input literals 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 proof logging method shown in Algorithm 1. Note that the nodes of the graph should be traversed in topological order when deriving the preservation equalities—this is so that the variables used in the reification steps are all fresh.

Let us now discuss three different ways of representing values of natural numbers that are used in preservation equality for inner nodes. Perhaps the most straightforward way to encode a number j with domain $A = \{0, 1, \ldots, m\} \subseteq \mathbb{N}_0$ with Boolean variables is to write j in unary with variables z_i so that $j = \sum_{i\in[m]} z_i$. In such an encoding we can also require, using constraints $z_i \geq z_{i+1}$, that the variables z_i are ordered so that z_i is true if and only if $j \geq i$. This means that listing the variables in reverse order $z_m, z_{m-1}, \ldots, z_1$ yields the number j written in unary (after a prefix of zeros). This is known as the *order encoding*, and this type of representation is used in the sequential counter [57] and totalizer [3] encodings. We can certify the correctness of this encoding as stated in the next proposition (where we defer all proofs in what follows to Appendix A due to space constraints).

Algorithm 1 General algorithm for translating PB constraints to CNF with proof logging.

procedure translate_and_certify(C, f, G, F)
  ▷ input: pseudo-Boolean constraint C of the form $\sum_{i=1}^{n} a_i\ell_i \bowtie k$, with $\bowtie \in \{\geq, \leq, =\}$
  ▷ input: arithmetic graph $G = (V, E)$ with input $\sum_i a_i\ell_i$ and output $\sum_i c_i o_i$
  ▷ input: function f that takes a node and derives its preservation equality
  ▷ input: set of clauses F with CNF encoding to be derived
  sum constraints $f(v)$ for $v \in V$ in topological order to obtain $\sum_i a_i\ell_i = \sum_i c_i o_i$
  combine $\sum_i a_i\ell_i = \sum_i c_i o_i$ and C to obtain $\sum_i c_i o_i \bowtie k$
  derive each clause in the CNF encoding F with reverse unit propagation (RUP)

▶ **Proposition 5** (Unary Sum). *For literals ℓ_i and fresh variables z_i, $i \in [n]$, the constraints*

$$\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} z_i \tag{18a}$$

$$z_i \geq z_{i+1} \qquad i \in [n-1] \tag{18b}$$

can be derived in $\mathrm{O}(n)$ steps in cutting planes with reification.

A concrete illustration of how these derivations can be done was given in Example 1 (with ℓ_3, $s_{2,1}$, and $s_{2,2}$ playing the roles of the literals ℓ_i and $s_{3,j}$, $j \in [3]$, being the fresh variables).

When encoding the value of a number j that can only take a small number of values in a large range, it is wasteful to introduce variables for all values in the range. 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. In such cases we can instead use what we will refer to as a *sparse unary encoding*, where in our example $j \in \{0, 50, 75\}$ would be represented as $50 \cdot z_{50} + 25 \cdot z_{75}$, where we enforce $z_{50} \geq z_{75}$. More formally, for a (finite) domain $A \subseteq \mathbb{N}_0$ and variables $\vec{z} = \{z_i \mid i \in A \cup \{\infty\}\}$ we define

$$\mathsf{sparse}(\vec{z}, A) \doteq \sum_{i \in A \setminus \{0\}} (i - \mathsf{pred}(i, A)) \cdot z_i \,, \tag{19a}$$

where $\mathsf{pred}(i, A) = \max\{j \in A \cup \{0\} \mid j < i\}$, and we also use constraints

$$z_i \geq z_{\mathsf{succ}(i,A)} \qquad i \in A \setminus \{\max(A)\} \tag{19b}$$

to enforce that the variables z_i are ordered, where $\mathsf{succ}(i, A) = \min\{j \in A \mid j > i\}$. This representation is used in the sequential weight counter [37] and generalized totalizer [38] encodings, and we can certify correctness for it as stated next.

▶ **Proposition 6** (Sparse Unary Sum). *Let $A, B \subseteq \mathbb{N}_0$ be given with sparse encodings $\mathsf{sparse}(\vec{x}, A)$ and $\mathsf{sparse}(\vec{y}, B)$ as in (19a)–(19b). Then for $E = \{i + j \mid i \in A, j \in B\}$ and fresh variables $\vec{z}$ we can derive*

$$\mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) = \mathsf{sparse}(\vec{z}, E) \tag{20a}$$

$$z_i \geq z_{\mathsf{succ}(i,E)} \qquad i \in E \setminus \{\max(E)\} \tag{20b}$$

in cutting planes with reification using $\mathrm{O}(|A| \cdot |B|)$ steps.

As in the case of the unary sum in Proposition 5, adding the constraints (20a)–(20b) maintains equisatisfiability, because the fresh variables $\vec{z}$ are free to take values so that the constraints are satisfied. The general idea is again to introduce $\vec{z}$ via reification, but the rest of the proof of Proposition 6 gets a bit more complicated—we have to perform a brute-force

search on the possible combinations of values for A and B, showing that the equality holds in all cases, and provide a proof log for the correctness of this backtracking search.

If we perform sums repeatedly as in Proposition 6, then the size of the domain can double in every step in the worst case, leading to an exponential explosion (this happens, for instance, if all values in the domains are distinct powers of 2). The third encoding we consider addresses this worst-case scenario by using a *binary encoding* $j = \sum_{i=0}^{\lfloor \log_2(m) \rfloor} 2^i \cdot z_i$. To compute the binary representation, it is sufficient—as we will discuss next in Section 5—to compose multiple full adders, which compute the sum of up to three input bits, using a binary adder circuit as described in [23].

▶ **Proposition 7.** *For literals ℓ_1, ℓ_2, ℓ_3 and fresh variables c, s, we can derive the equality*

$$\ell_1 + \ell_2 + \ell_3 = 2c + s \tag{21}$$

in cutting planes with reification using O(1) *steps.*

Again, it should be clear that this maintains equisatisfiability, since the carry-out bit c and sum bit s can be set appropriately. To derive (21) we first reify

$$c \Leftrightarrow \ell_1 + \ell_2 + \ell_3 \geq 2 \tag{22a}$$
$$s \Leftrightarrow \ell_1 + \ell_2 + \ell_3 + 2\overline{c} \geq 3 \tag{22b}$$

and then multiply (22a) by 2, add (22b), and divide the result by 3. To show how this works for the $\Rightarrow$-direction of the reification, 2 times (22a) is $4\overline{c} + 2\ell_1 + 2\ell_2 + 2\ell_3 \geq 4$, adding $3\overline{s} + \ell_1 + \ell_2 + \ell_3 + 2\overline{c} \geq 3$ as in (22b) yields $6\overline{c} + 3\overline{s} + 3\ell_1 + 3\ell_2 + 3\ell_3 \geq 7$, and dividing by 3 gives us $2\overline{c} + \overline{s} + \ell_1 + \ell_2 + \ell_3 \geq 3$ as desired. We refer the reader to [33] for more details.

5 Certifying the Binary Adder Network Encoding

The idea behind the *binary adder encoding* [23] is to use an adder network to compute the value of $\sum_i a_i \ell_i$ as a binary number $\sum_{i=0}^{bits} 2^i o_i$, where $bits = \lfloor \log_2(\sum_i a_i) \rfloor$ is the required *bit width*, and then compare this to the right-hand side constant in the constraint $\sum_i a_i \ell_i \bowtie k$.

To recapitulate the algorithm for adder network construction in [23], let us say that a 2^m*-bit* is a literal representing the numerical value 2^m and that a 2^m*-bucket* is a queue of 2^m-bits. We use $[m]_2$ to denote the binary representation of a natural number m. The algorithm starts by initializing each 2^m-bucket with all literals ℓ_i in $\sum_i a_i \ell_i \bowtie k$ such that the 2^m-bit of $[a_i]_2$ is 1. Then for m in increasing order we repeat the following procedure: While there are at least 2 elements in the 2^m-bucket, dequeue three bits x, y, z, or set $z = 0$ if there are exactly 2 bits left. Use x, y, and z as input for a new full adder with fresh variables c and s as output (these are just placeholder names), and insert s in the 2^m-bucket and c in the 2^{m+1}-bucket (possibly creating a new bucket). See Algorithm 2 for the pseudocode.

The arithmetic graph is obtained from the adder network by representing each full adder by a node. Each inner node constructed from a 2^m-bucket has 3 input edges with labels $2^m \cdot x$, $2^m \cdot y$, and $2^m \cdot z$ and 2 output edges with labels $2^m \cdot s$ and $2^{m+1} \cdot c$. An example for the PB expression $5x_1 + 4x_2 + x_3 + x_4 + x_5$ is shown in Figure 4. The preservation equality can be derived using Proposition 7 and multiplying the resulting equality $x + y + z = 2c + s$ by 2^m to obtain $2^m \cdot x + 2^m \cdot y + 2^m \cdot z = 2^{m+1} \cdot c + 2^m \cdot s$. When the construction algorithm ends, 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 $\sum_{i=0}^{bits} 2^i \cdot o_i$. If the 2^i-bucket is empty, o_i is fixed to 0.

Algorithm 2 Construction of adder network [23]. Procedure full_adder adds full adder to network.

```
1:  procedure adder_network(b)
2:      ▷ input: vector of buckets b
3:      for i from 0 to b.size() do
4:          while b_i.size() ≥ 2 do
5:              if b_i.size() = 2 then
6:                  (x, y) ← b_i.dequeue()
7:                  (c, s) ← full_adder(x, y, 0)
8:              else
9:                  (x, y, z) ← b_i.dequeue()
10:                 (c, s) ← full_adder(x, y, z)
11:             b_i.enqueue(s)
12:             b_{i+1}.enqueue(c)
```

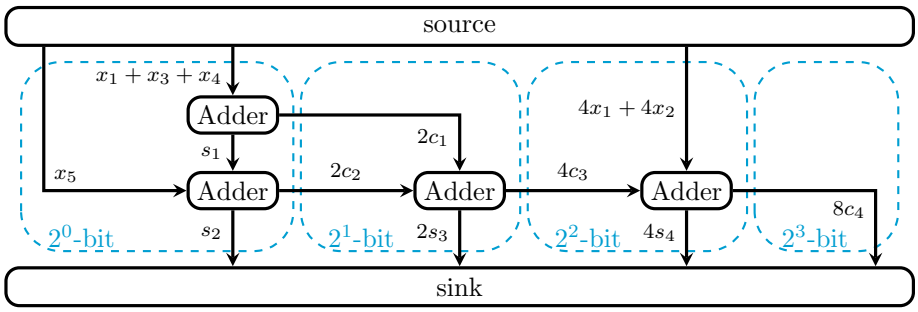


Figure 4 Layout of arithmetic graph for adder network encoding of $5x_1 + 4x_2 + x_3 + x_4 + x_5$.

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

$$\begin{array}{cccc} & \overline{x}+\overline{y}+\overline{z}+s\geq 1 & & x+y+z+\overline{s}\geq 1 \\ \overline{y}+\overline{z}+c\geq 1 & \overline{x}+y+z+s\geq 1 & y+z+\overline{c}\geq 1 & x+\overline{y}+\overline{z}+\overline{s}\geq 1 \\ \overline{x}+\overline{z}+c\geq 1 & x+\overline{y}+z+s\geq 1 & x+z+\overline{c}\geq 1 & \overline{x}+y+\overline{z}+\overline{s}\geq 1 \\ \overline{x}+\overline{y}+c\geq 1 & x+y+\overline{z}+s\geq 1 & x+y+\overline{c}\geq 1 & \overline{x}+\overline{y}+z+\overline{s}\geq 1 \end{array} \tag{23}$$

which are all RUP with respect to the preservation equality $x + y + z = 2c + s$.

To compare the constant k in the PB constraint with the output of the circuit, we encode a bitwise comparison $\vec{x} \geq \vec{y}$ for bit vectors $\vec{x}$ and $\vec{y}$, where $\vec{x} = o_{bits} \cdots o_1 o_0$ 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. For $\sum_{i=1}^{n} a_i \ell_i = k$, comparisons for both directions are performed. If the sizes of the two vectors are different, the shorter vector is padded with 0, after which the constraints

$$x_i + \overline{y}_i + \textstyle\sum_{j=i}^{bits} x_j \overline{y}_j + \overline{x}_j y_j \geq 1 \qquad\qquad i = 0, 1, \ldots, bits \tag{24}$$

are added to the CNF encoding. Since either $\vec{x}$ or $\vec{y}$ is a vector of constant bits, the constraints (24) are indeed clauses. If for a fixed index i we have that $\vec{x}$ and $\vec{y}$ differ in some coordinate $j > i$, then $\sum_{j=i+1}^{bits} x_j \overline{y}_j + \overline{x}_j y_j \geq 1$ holds, and the ith clause is satisfied. Otherwise, if $x_i = 1$ or $y_i = 0$ we have $x_i + \overline{y}_i \geq 1$ and the clause is again satisfied, which is in order since the comparison $\vec{x} \geq \vec{y}$ cannot fail for the 2^i-bit if all more significant bits are equal. If none of these cases apply, then the clause reduces to $x_i \overline{y}_i + \overline{x}_i y_i \geq 1$. This enforces that it must not be the case that all more significant bits are equal but that $x_i = 0$ and

$y_i = 1$, because if so we have $\vec{x} < \vec{y}$. The clauses (24) are RUP with respect to the constraint $\sum_{i=0}^{bits} 2^i \cdot o_i \bowtie k$, which we obtain from the arithmetic graph using Proposition 4. To see this, note that asserting (24) to false will set all 2^j-bits for $j > i$ equal but the 2^i-bits to opposite values, which immediately falsifies $\sum_{i=0}^{bits} 2^i \cdot o_i \bowtie k$.

6 Experimental Evaluation

To evaluate the proof logging methods developed in this paper, we have implemented certified translations to CNF for the sequential counter [57], adder network [23], totalizer [3], and generalized totalizer [38] encodings in a tool VeritasPBLib. This tool takes a pseudo-Boolean formula in OPB format [54] and returns a CNF translation with a proof logging certificate. We have employed the verifier VeriPB [60] to check the certificate returned by VeritasPBLib, and have used the solver Kissat [42], in a lightly modified version outputting *DRAT* proofs in pseudo-Boolean format,[1] to solve the CNF formula. Finally, we have conjoined the certificates from the CNF translation and the SAT solving and verified the end-to-end pipeline with VeriPB. See [30] for source code and experimental data.

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, which tends to be slower than solving the problem.

To evaluate VeritasPBLib, we collected 1,803 pseudo-Boolean formulas from the PB 2016 Evaluation.[2] These instances can be partitioned into formulas with (1) only clauses (279 instances), (2) clauses and cardinality constraints (772 instances), (3) clauses and general PB constraints (444 instances), and (4) clauses, cardinality and general PB constraints (308 instances). Since this work targets the verification of formulas with non-clausal constraints, we excluded the 279 pure CNF formula instances, as those can already be certified with existing techniques. Our evaluation aimed to answer the following questions:

1. Can we use our end-to-end framework to verify the results of CDCL-based pseudo-Boolean solving, and how efficient is the verification?
2. How long does verification of the proof logging take when compared to the translation of 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 encodings to translate those constraints to CNF. For instances with general PB constraints (*PB*), we translate such constraints using the adder network and generalized totalizer (*GTE*) encodings. Finally, for instances with both cardinality and general PB constraints (*Card+PB*), we use the sequential encoding for cardinality constraints and the adder network 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 for a variety of encodings.

[1] This modified version of Kissat with pseudo-Boolean proof logging is available at `https://gitlab.com/MIAOresearch/kissat_fork`.
[2] These pseudo-Boolean benchmarks are 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		Solving			
			#CNF	#Veri	#Solved		#Verified	
					SAT	UNSAT	SAT	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

The column *#CNF* shows for how many instances VeritasPBLib successfully generated the CNF translation, which is almost all. The exceptions are 19 instances using *GTE* 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.

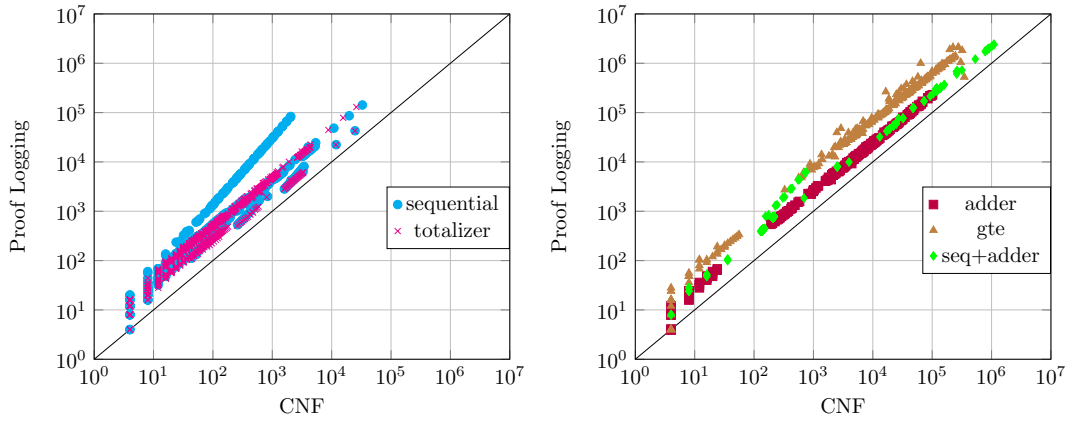
The column *#Veri* under *Translation* shows results for VeriPB verification of the translation certificate from VeritasPBLib. Except for a few instances for *GTE* and *Seq+Adder* yielding large proofs, VeriPB is successful. Note that if the translation check passes, then this guarantees that the CNF 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 Kissat, and from those how many can be verified by VeriPB. If a satisfiable formula is verified, this means that all clauses learned by Kissat are also valid for the original pseudo-Boolean formula, as is the satisfying assignment found. If an unsatisfiable formula is verified, then a correct proof of unsatisfiability for the PB formula has been produced.

We can verify 99% of the solved unsatisfiable instances, which shows that the current proof-of-concept approach is already practical in this setting. For satisfiable formulas we can verify 95% of the solved instances. However, even when VeriPB does not terminate within the timeout limit, we can still certify that the satisfying assignment found by the SAT solver is valid for the original PB formula. We note that there is still ample room for performance improvements in VeriPB proof checking—in particular, when it comes to verifying the *DRAT* proofs produced by the SAT solver, which do not even use pseudo-Boolean reasoning, but are simply clausal proofs syntactically rewritten in pseudo-Boolean format. Making VeriPB faster at validating such proofs would further increase the number of instances that could be verified, but work in that direction is orthogonal to the contributions of this paper.

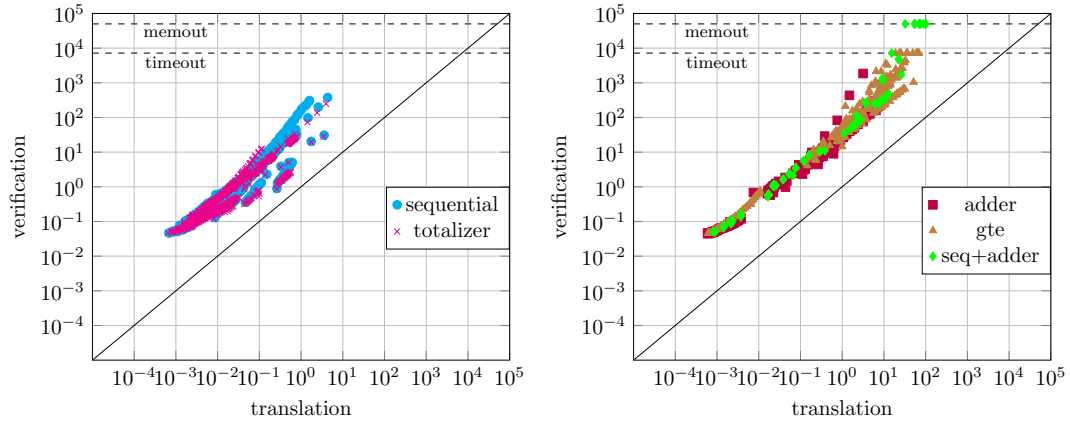
Translation and Verification Turning our focus to the certified translation only, our experiments show that the average overhead in running time for proof logging is a factor of 2–3 for all encodings except *GTE*, which incurs around a factor 5 in overhead. However, since translation is fast for the majority of instances (see Figure 6), the additional overhead of proof logging is not an issue when translating the pseudo-Boolean formulas to CNF.

This overhead can be explained by the proofs being larger than the generated CNF formulas, as shown in Figure 5. For most instances the proof size seems to be within a constant factor of the CNF formula size, but for a collection of crafted vertex cover problems [25] the sequential counter encoding turns out to require proofs of super-linear size. These instances contain a constraint enforcing a constant fraction of the literals in the formula to be false, which is a worst-case scenario for the sequential counter encoding. While the number of clauses in the CNF translation is still linearly related to the number of steps



(a) Cardinality formulas. **(b)** General pseudo-Boolean formulas.

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



(a) Cardinality formulas. **(b)** General pseudo-Boolean formulas.

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

in the proof, each reification step in the unary sum derivation in Proposition 5 introduces a constraint of linear size, making the total proof size quadratic. It would be desirable to find a more efficient derivation that only requires linear proof size.

Figure 6 compares the time for VeritasPBLib to generate the CNF translation and VeriPB to verify it. The verification overhead is far from negligible, but not unreasonable. Over all encodings, for 75% of benchmarks verification takes at most 49 times longer than translation, and for 98% of benchmarks at most 100 times longer. While some overhead is natural, since the translation algorithm can just output a claimed proof while the verifier needs to perform the calculations to actually check it, our experiments indicate the need to improve the efficiency both of the verifier and of the proof logging methods used.

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 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, the experimental evaluation shows that our approach is feasible in practice. We believe that the generality of our method, which expresses the proof logging steps in terms of simple operations on a graph representation of the PB-to-CNF translation, is an important aspect of our work. However, it should be noted that for some *sorting network* encodings found to be particularly efficient in [23], such as the *odd-even merge sorters* [6] used in MiniSat+, we do not yet know of a nice way of capturing them in the framework developed in this paper. The same problem applies to encodings based on *binary decision diagrams BDDs* [12], and we leave this as future work.

As discussed already in the introduction, our paper does not quite reach the goal of certifying *equivalence* of the original pseudo-Boolean formula F and the CNF translation F'. In one direction, it is clear that as long as F' is derived from F using cutting planes with reification, any satisfying assignment α to F yields a unique extended assignment $\alpha' \supseteq \alpha$ satisfying F' by giving all newly introduced variables the values determined by the reification rules (3a)–(3b). In the other direction, however, we do not formally establish that the CNF translation F' is as strong as the original pseudo-Boolean formula F in the sense that any satisfying assignment α' for F' is guaranteed to also satisfy F. As a quick technical detour, one way of achieving such guarantees would be, after having derived all clauses in F', to erase all constraints in F using the "checked deletion" rule in [10], and to only allow standard, implicational, cutting planes rules in the proof that the deleted constraint can be rederived from what is left in the constraint database. This is certainly doable in principle, but we currently know of no clean and simple way to formalize this in our graph-based translation framework. This is therefore another problem that we have to leave as future research.

Concluding this section, we wish to emphasize that we view pseudo-Boolean decision problems as only a first step, and believe that the techniques in this paper should also be sufficient to support proof logging for MaxSAT solvers, including derivation of clauses added during core extraction and objective function reformulation in *core-guided solving* [28, 47]. While designing efficient proof logging for MaxSAT approaches such as *implicit hitting sets (IHS)* [20] and *abstract cores* [7] seems more challenging, we are still hopeful that our work could lead to a unified proof logging method not only for MaxSAT solving but also for pseudo-Boolean optimization using cutting-planes-based reasoning as in [21, 22, 27, 43].

Acknowledgements

The authors wish to acknowledge helpful and stimulating discussions with Bart Bogaerts and Ciaran McCreesh. We are particularly grateful to Bart for sharing the manuscript [59] using a very elegant reification technique that we wish we would have thought of, and we believe it would be worth exploring whether similar ideas could be used in our framework to improve the efficiency of verification. We are also thankful for discussions and feedback at the workshops *Pragmatics of SAT* and *Proof eXchange for Theorem Proving* in 2021, where a preliminary version of this work was presented, and we would like to extend a special thanks to the *SAT '22* anonymous reviewers for a wealth of comments and questions that helped to improve this manuscript considerably.

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, March-April 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)*, volume 12699 of *Lecture Notes in Computer Science*, pages 252–272. Springer, July 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 Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS '68)*, volume 32, pages 307–314, April 1968.

7 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.

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

9 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.

10 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, February 2022. To appear.

11 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.

12 Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

13 Randal E. Bryant, Armin Biere, and Marijn J. H. Heule. Clausal proofs for pseudo-Boolean reasoning. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '22)*, volume 13243 of *Lecture Notes in Computer Science*, pages 443–461. Springer, April 2022.

14 Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [9], chapter 7, pages 233–350.

15 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.

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

17 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.

18 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.

19 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.

20 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.

21 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*.

22 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.

23 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.

24 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.

25 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.

26 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.

27 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.

28 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.

29 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.

30 Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Experimental repository for "Certified CNF translations for pseudo-Boolean solving". Available at `https://doi.org/10.5281/zenodo.6610581`, June 2022.

31 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.

32 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.

33 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.

34 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.

35 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.

36 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.

37 Steffen Hölldobler, Norbert Manthey, and Peter Steinke. A compact encoding of pseudo-Boolean constraints into SAT. In *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.

38 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.

39 Daniela Kaufmann, Paul Beame, Armin Biere, and Jakob Nordström. Adding dual variables to algebraic reasoning for circuit verification. In *Proceedings of the 25th Design, Automation and Test in Europe Conference (DATE '22)*, pages 1435–1440, March 2022.

40 Daniela Kaufmann and Armin Biere. AMulet 2.0 for verifying multiplier circuits. 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 357–364. Springer, March-April 2021.

41 Daniela Kaufmann, Mathias Fleury, and Armin Biere. The proof checkers Pacheck and Pastèque for the practical algebraic calculus. In *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design (FMCAD '20)*, pages 264–269, September 2020.

42 Kissat SAT solver. `http://fmv.jku.at/kissat/`.

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

44 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*.

45 Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. 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 438–445. Springer, July 2014.

46 Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

47 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.

48 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.

49 NaPS (Nagoya pseudo-Boolean solver). https://www.trs.cm.is.nagoya-u.ac.jp/projects/NaPS/.

50 Open-WBO: An open source version of the MaxSAT solver WBO. http://sat.inesc-id.pt/open-wbo/.

51 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.

52 Pseudo-Boolean competition 2016. http://www.cril.univ-artois.fr/PB16/, July 2016.

53 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.

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

55 Masahiko Sakai and Hidetomo Nabeshima. Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. *IEICE Transactions on Information and Systems*, 98-D(6):1121–1127, June 2015.

56 The international SAT Competitions web page. http://www.satcompetition.org.

57 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.

58 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.

59 Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. Manuscript, 2022.

60 VeriPB: Verifier for pseudo-Boolean proofs. https://gitlab.com/MIAOresearch/VeriPB.

61 Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.

62 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 3 Deriving a unary sum over fresh variables z_i.

procedure derive_unary_sum(C')
  ▷ input: C' of the form $\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} z_i$ and describing 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**
    $D_j^{\text{geq}}, D_j^{\text{leq}} \leftarrow$ reify($z_j \Leftrightarrow \sum_{i=1}^{n} 1 \cdot \ell_i \geq j$) ▷ **Step 3.1**: introduce variables
  $C^{\text{geq}} \leftarrow$ derive_sum($D_1^{\text{geq}}, D_2^{\text{geq}}, \ldots, D_n^{\text{geq}}$) ▷ **Step 3.2**: derive $\sum_{i=1}^{n} \ell_i \geq \sum_{i=1}^{n} z_i$
  $C^{\text{leq}} \leftarrow$ derive_sum($D_n^{\text{leq}}, D_{n-1}^{\text{leq}}, \ldots, D_1^{\text{leq}}$) ▷ **Step 3.3**: derive $\sum_{i=1}^{n} \ell_i \leq \sum_{i=1}^{n} z_i$
  for i from 1 to $k-1$ **do**
    derive_ordering(D_i^{leq}, D_{i+1}^{geq}) ▷ **Step 3.4**: derive $z_i \geq z_{i+1}$, $i \in [n-1]$
  return $C^{\text{geq}}, C^{\text{leq}}$

Algorithm 4 Reify $\sum_{i=1}^{n} a_i \ell_i \geq j$ using the fresh variable z_j.

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

A Derivations for Proof Logging Building Blocks

In this appendix we provide the missing technical details for Section 4. Let us start by explaining our proof logging notation, which is similar to what is used in VeriPB proof files.

The proof is constructed line by line using the proof_log(·) function, with each call to this function adding a new pseudo-Boolean constraint derived from previous lines in the proof. Every constraint in the proof gets a unique *identifier*, and we can write down cutting planes derivations of new constraints in reverse polish notation using these identifiers to refer to previous constraints. For example, given previously derived constraints with identifiers C and D, calling 'proof_log(`pol` C 2 `d` D 3 `* + s`)' divides C by 2 (and rounds up), multiplies D by 3, adds the two constraints obtained in this way, applies saturation, and returns the resulting constraint. A reverse unit propagation (RUP) constraint C can be added using 'proof_log(`rup` C)'. The syntax we use for deriving a constraint by reification is 'proof_log(`red` $z \Rightarrow C$ `;` z 0)' and 'proof_log(`red` $z \Leftarrow C$ `;` z 1)' (where this somewhat cryptic notation is due to that reification is a special case of the redundance rule in [33]). We use '▷' to denote comments in the pseudocode.

A.1 Deriving Unary Sum Constraints

Deriving the unary sum constraints

$$\sum_{i=1}^{n} \ell_i \geq \sum_{i=1}^{n} z_i \tag{25a}$$
$$\sum_{i=1}^{n} \ell_i \leq \sum_{i=1}^{n} z_i \tag{25b}$$
$$z_i \geq z_{i+1} \qquad i \in [n-1] \tag{25c}$$

in Proposition 5 for fresh variables z_j is described in Algorithm 3, which is split into four steps. Step 3.1 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 4 for the more general case of arbitrary positive coefficients.

Algorithm 5 Derive sum of reification variables.

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

Algorithm 6 Deriving an ordering constraint $z_A \geq z_B$ from the reification constraints.

procedure derive_ordering(C, D)
  ▷ input: C is of the form $z_A \Rightarrow \sum_{i=1}^{n} a_i\ell_i \geq A$
  ▷ input: D is of the 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`)

In Step 3.2 the lower bound (25a) is derived using Algorithm 5 maintaining the invariant $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \overline{z}_i \geq j$. For the base case $j = 1$, the invariant is equivalent to the reification constraint $z_1 \Rightarrow \sum_{i=1}^{n} \ell_i \geq 1$, which in normalized form is $\sum_{i=1}^{n} \ell_i + \overline{z}_1 \geq 1$ and hence this case is covered. For the inductive step, to go 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 $\sum_{i=1}^{n} \ell_i + (j+1)\overline{z}_{j+1} \geq j+1$ in normalized form, to get $(j+1)\sum_{i=1}^{n} \ell_i + j\sum_{i=1}^{j} \overline{z}_i + (j+1)\overline{z}_{j+1} \geq j^2 + j + 1$. Note that $j^2 + j + 1 = (j+1)^2 - j$ and hence division by $j+1$ and rounding up yields $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \overline{z}_i + \overline{z}_{j+1} \geq j+1$, i.e., the invariant for $j+1$. For $j = k+1$ the invariant is the normalized form of (25a).

In Step 3.3 the upper bound (25b) is again derived using Algorithm 5, except that the constraints are processed in reverse order (just as in Example 1 on page 7).

In Step 3.4 the ordering constraint is derived in Algorithm 6, 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 $(j+1)\overline{z}_{j+1} + \sum_{i=1}^{n} a_i\ell_i \geq j+1$ and $(m-j+1)z_j + \sum_{i=1}^{n} a_i\overline{\ell}_i \geq m-j+1$, where $m = \sum_{i=1}^{n} a_i$. Adding both constraints together yields $(m-j+1)z_j + (j+1)\overline{z}_{j+1} \geq 2$ and we get the desired ordering constraint after division by a large enough number, such as m.

A.2 Deriving Sparse Unary Sum Constraints

Let us now prove Proposition 6 by presenting and analyzing Algorithm 7, which given two numbers in sparse unary representation derives their sum. Just as for the unary sum, we start in Step 7.1 by introducing the required fresh variables via reification. However, we only need to introduce the variables 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 ordering constraints can be derived as before using Algorithm 6.

In Step 7.2 we introduce a variable z_{eq} which is true if and only if the equality to be derived is true. Since an equality is actually two inequalities, we need to introduce separate variables z_{geq}, z_{leq} for each inequality and then combine them into z_{eq}.

In Step 7.3 we derive $z_{eq} \geq 1$ by checking all combinations of values in A and B, which requires $O(|A| \cdot |B|)$ steps. Asymptotically, this is the same number of steps required to

Algorithm 7 Deriving a sparse unary sum over fresh variables $\vec{z}$.

procedure derive_sparse_unary_sum(C')
  ▷ input: C' of the form $\mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) = \mathsf{sparse}(\vec{z}, E)$ and describing the constraint to be derived such that $A, B \subseteq \mathbb{N}$, $E = \{i + j | i \in A, j \in B\}$
  ▷ **Step 7.1**: Introduce variables as reification and derive ordering.
  for $j \in E \setminus \{0\}$ **do**
    $D_j^{\text{geq}}, D_j^{\text{leq}} \leftarrow \text{reify}(z_j \Leftrightarrow \mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) \geq j)$
  for $i \in E \setminus \{0, \max(E)\}$ **do**
    derive_ordering(D_i^{leq}, $D_{\mathsf{succ}(i,E)}^{\text{geq}}$) ▷ derive $z_i \geq z_{\mathsf{succ}(i,E)}$
  ▷ **Step 7.2**: : reify constraint to be derived
  $C^{\text{geq}}, \_ \leftarrow \text{reify}(z_{geq} \Leftrightarrow \mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) \geq \mathsf{sparse}(\vec{z}, E))$
  $C^{\text{leq}}, \_ \leftarrow \text{reify}(z_{leq} \Leftrightarrow \mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) \leq \mathsf{sparse}(\vec{z}, E))$
  $\text{reify}(z_{eq} \Leftrightarrow z_{geq} + z_{leq} \geq 2)$
  ▷ **Step 7.3**: derive that $z_{eq} \geq 1$
  $\text{try\_all\_values}(\mathsf{sparse}(\vec{x}, A), \mathsf{sparse}(\vec{y}, B), z_{eq})$
  ▷ **Step 7.4**: 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 $C^{\text{geq}}, C^{\text{leq}}$

compute which elements are in E, so this is still linear in the time needed to construct the encoding.

In Step 7.4 we use that $z_{eq} \geq 1$ and hence $z_{geq} = z_{leq} = 1$, which allows us to derive $\mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) \geq \mathsf{sparse}(\vec{z}, E)$ and $\mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) \leq \mathsf{sparse}(\vec{z}, E)$ by removing z_{geq} and z_{leq} from the constraints introduced in Step 7.2.

Algorithm 8 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 in Algorithm 8 the clauses

$$x_2 + y_2 + \quad z_{eq} \geq 1 \tag{26a}$$
$$x_2 + \overline{y}_2 + y_4 + z_{eq} \geq 1 \tag{26b}$$
$$x_2 + \quad \overline{y}_4 + z_{eq} \geq 1 \tag{26c}$$

have been derived. Deriving (26a) by RUP sets $x_2 = y_2 = z_{eq} = 0$. This causes the ordering constraints to propagate all variables in $\vec{x}$ and $\vec{y}$. As all $\vec{x}$ and $\vec{y}$ variables are set, the reification constraints introduced in Step 7.1 will cause all $\vec{z}$ variables to propagate. As the constraints reified in Step 7.2 are satisfied, $z_{geq} = z_{leq} = 1$ is propagated and hence z_{eq} should be 1. However, we already set z_{eq} to 0, which is a contradiction showing that (26a) can be derived. Deriving the other clauses works analogously.

If we add all clauses in (26) together, we are left with $3x_2 + 3z_{eq} \geq 1$, which is saturated to obtain $x_2 + z_{eq} \geq 1$. Analogously, in the second iteration we derive $\overline{x}_2 + z_{eq} \geq 1$, which added to the result of the first iteration yields $2z_{eq} \geq 1$ and using saturation we get $z_{eq} \geq 1$.

Algorithm 8 Given a reified sparse unary sum, derive that the reification variable is true.

procedure fix($\mathsf{sparse}(\vec{x}, A), a$)
  return $\overline{x}_a + x_{\mathsf{succ}(a,A)}$ ▷ replace x_0 by 1 and x_∞ by 0
procedure try_all_values($\mathsf{sparse}(\vec{x}, A), \mathsf{sparse}(\vec{y}, B), z_{eq}$)
  $C_{outer} \leftarrow 0 \geq 0$
  for $i \in A$ **do**
    $C_{inner} \leftarrow 0 \geq 0$
    for $j \in B$ **do**
      ▷ a (respectively b) is the value encoded by $\mathsf{sparse}(\vec{x}, A)$ ($\mathsf{sparse}(\vec{y}, B)$)
      ▷ encode that $(a = i \wedge b = j) \Rightarrow z_{eq}$
      $D \leftarrow \mathrm{fix}(\mathsf{sparse}(\vec{x}, A), i) + \mathrm{fix}(\mathsf{sparse}(\vec{y}, B), j) + z_{eq} \geq 1$
      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 $z_{eq} \geq 1$

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

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

A.3 Deriving Binary Adder Network Constraints

Algorithm 9 provides the details for the derivation of (and proof logging for) the preservation equality (21) for a single binary full adder, and this establishes Proposition 7.

B Certifying the Totalizer and Generalized Totalizer Encodings

The totalizer and generalized totalizer encoding accumulate the input in the form of a balanced binary tree. The totalizer encodes cardinality constraints and uses the order encoding to represent values, while the generalized totalizer encodes 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. The nodes are combined in form of a binary tree, where we ensure that the value is preserved for each inner node. To perform k-simplification, the arithmetic graph has additional edges that go directly to the sink node. The formal definition of the arithmetic graph for the (generalized) totalizer encoding is as follows.

▶ **Definition 8** (Arithmetic graph for the generalized totalizer encoding)**.** *Given a linear sum $\sum_i a_i \ell_i$ over n variables, let G be a binary tree with edges directed towards the root r,*

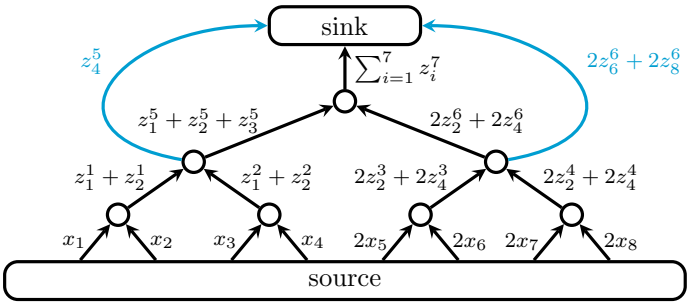


Figure 7 Layout of the arithmetic graph for the generalized totalizer encoding of $x_1 + x_2 + x_3 + x_4 + 2x_5 + 2x_6 + 2x_7 + 2x_8 \leq 2$. Edges introduced for k-simplification are colored cyan.

leaves s_i for $i \in [n]$ and an additional sink node t with an edge (r, t). The edge (s_i, v) is labelled with $a_i x_i$. For an inner node v with two incoming edges labelled $\mathsf{sparse}(\vec{x}, A)$ *and* $\mathsf{sparse}(\vec{y}, B)$*, the outgoing edge is labelled* $\mathsf{sparse}(\vec{z}, E)$*, where $\vec{z}$ are fresh variables and $E = \{i + j \mid i \in A, j \in B\}$. All s_i are combined into a single source node. For k-simplification we split* $\mathsf{sparse}(\vec{z}, E) = \sum_{i \in E} a_i z_i$ *into* $\sum_{i \leq \mathsf{succ}(k,E)} a_i z_i$ *and* $\sum_{i > \mathsf{succ}(k,E)} a_i c_i$.

To see that this graph is an arithmetic graph, we only need to check that we can derive the preservation equality for each inner node. We can use Proposition 6 to derive the required preservation equality. Proposition 6 also requires to have ordering constraints on the input literals. However, it is easy to see by an inductive argument that the ordering constraints on the literals can also be derived as we process the nodes in topological order. For the base case, edges from source nodes only contain a single literal, which is vacuously ordered. For inner nodes we get the ordering constraints by applying Proposition 6. If E contains all integers between 0 and $\max(E)$, we can use Proposition 5 to derive the preservation equality, which requires $O(|E|)$ steps instead of $O(|A| \cdot |B|)$ steps and hence reduces overhead.

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

$$\overline{x}_i + \overline{y}_j + z_{i+j} \geq 1 \qquad i \in A, j \in B \tag{27a}$$

$$x_{\mathsf{succ}(i,A)} + y_{\mathsf{succ}(j,B)} + \overline{z}_{\mathsf{succ}(i+j,E)} \geq 1 \qquad i \in A, j \in B \tag{27b}$$

for $\mathsf{succ}(i, A) = \min\{j \mid j \in A \cup \{\infty\}, j > i\}$ (where compared to Section 4 we have added an element ∞) and for x_0, y_0 replaced by 1 and $x_\infty, y_\infty, z_\infty$ by 0, with ensuing simplification.

For the proof logging of the CNF encoding we can simply add all clauses using reverse unit propagation. A RUP check of (27a) will assign $x_i = y_j = 1$ and $z_{i+j} = 0$. The ordering constraints on $\vec{x}, \vec{y}$ will propagate variables in $\vec{x}, \vec{y}$ to true so that $\mathsf{sparse}(x, A) + \mathsf{sparse}(y, B)$ has a value of at least $i + j$, while the ordering constraints on $\vec{z}$ will propagate variables in $\vec{z}$ to false so that $\mathsf{sparse}(z, E)$ can only take a value strictly less than $i + j$. This will violate the preservation equality $\mathsf{sparse}(z, E) = \mathsf{sparse}(x, A) + \mathsf{sparse}(y, B)$, showing that (27a) is indeed a RUP clause. Deriving the clause (27b) works analogously.

To enforce a pseudo-Boolean constraint $\sum_i a_i \ell_i \bowtie k$, 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 $\sum_i a_i \ell_i \geq k$ or $\sum_i a_i \ell_i \leq k$ the unit clause $z_{\mathsf{succ}(k-1,E)} \geq 1$ or $\overline{z}_{\mathsf{succ}(k,E)} \geq 1$ 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 $\sum_i a_i \ell_i = k$ both unit clauses are added.