Journal of Artificial Intelligence Research X (2022) Y-Z Submitted 9/22; published VV/WW

Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs

Stephan Gocht STEPHAN.GOCHT@CS.LTH.SE
Lund University, Lund, Sweden
University of Copenhagen, Copenhagen, Denmark

Jakob Nordström JN@DI.KU.DK
University of Copenhagen, Copenhagen, Denmark
Lund University, Lund, Sweden

Abstract

The dramatic improvements in combinatorial optimization algorithms over the last decades have had a major impact in artificial intelligence, operations research, and beyond, but the output of current state-of-the-art solvers is often hard to verify and is sometimes wrong. For Boolean satisfiability (SAT) solvers proof logging has been introduced as a way to certify correctness, but the methods used seem hard to generalize to stronger paradigms. What is more, even for enhanced SAT techniques such as parity (XOR) reasoning, cardinality detection, and symmetry handling, it has remained beyond reach to design practically efficient proofs in the standard *DRAT* format. In this work, we show how to instead use pseudo-Boolean inequalities with extension variables to concisely justify XOR reasoning. Our experimental evaluation of a SAT solver integration shows a dramatic decrease in proof logging and verification time compared to existing *DRAT* methods. Since our method is a strict generalization of *DRAT*, and readily lends itself to expressing also 0-1 programming and even constraint programming problems, we hope this work points the way towards a unified approach for efficient machine-verifiable proofs for a rich class of combinatorial optimization paradigms.

1. Introduction

Since around the turn of the millennium, combinatorial optimization has been successfully applied to solve an ever increasing range of problems in e.g., resource allocation, scheduling, logistics, and disaster management (Pardalos, Du, & Graham, 2013), and more recent applications in biology, chemistry, and medicine include, e.g., protein analysis and design (Allouche et al., 2014; Mann et al., 2008) and planning for kidney transplants (Manlove & O'Malley, 2012; Biró et al., 2021). Yet other examples are government auctions generating billions of dollars in revenue (Leyton-Brown, Milgrom, & Segal, 2017), as well as allocation of education and work opportunities (Manlove, 2016; Manlove, McBride, & Trimble, 2017) and matching of adoptive families with children (Delorme et al., 2019).

As more and more such problems are dealt with using combinatorial optimization solvers, an urgent question is whether we can trust that the solutions computed by such algorithms are correct and complete. The answer, unfortunately, is currently a clear "no": State-of-the-art solvers sometimes return "solutions" that do not satisfy the constraints or erroneously claim optimality of solutions (Cook et al., 2013; Akgün et al., 2018; Gillard et al., 2019). This can be fatal for applications such as, e.g., chip design, compiler optimization, and

combinatorial auctions, where correctness is absolutely crucial, not to speak about when human lives depend on finding the best solutions.

Conventional software testing has made little progress in addressing this problem, and formal verification techniques cannot handle the level of complexity of modern solvers. Instead, the most successful approach to date has been that of *proof logging* in the Boolean satisfiability (SAT) community, where solvers are required to be *certifying* (McConnell, Mehlhorn, Näher, & Schweitzer, 2011) in the sense that they output not only a result but also a simple, machine-verifiable proof that this result is correct.

This does not certify the correctness of the solver itself, but it does mean that if it ever produces an incorrect answer (even if due to hardware errors), then this can be detected. Furthermore, such proofs can in principle be stored and audited later by a third party using independently developed software. A number of different proof logging formats such as *RUP* (Goldberg & Novikov, 2003), *TraceCheck* (Biere, 2006), *DRAT* (Heule, Hunt Jr., & Wetzler, 2013a, 2013b; Wetzler, Heule, & Hunt Jr., 2014), *GRIT* (Cruz-Filipe, Marques-Silva, & Schneider-Kamp, 2017b), and *LRAT* (Cruz-Filipe et al., 2017a) have been developed, with *DRAT* now established as the standard in the SAT competitions.[1]

A quite natural, and highly desirable, goal would be to extend these proof logging techniques to stronger combinatorial optimization paradigms such as pseudo-Boolean (PB) optimization, MaxSAT solving, mixed integer linear programming (MIP), and constraint programming (CP), but such attempts have had limited success. Either the proofs require trusting in powerful and complicated rules (as in, e.g., (Veksler & Strichman, 2010)), defeating simplicity and verifiability, or they have to justify such rules by long explanations, leading to an exponential slow-down (see (Gange & Stuckey, 2019)). In fact, even for SAT solvers a long-standing problem is that more advanced techniques for detecting and reasoning with parity constraints (a.k.a. exclusive or, or XOR, constraints), cardinality constraints, and symmetries have remained out of reach for efficient proof logging. Although in theory it might seem like there should be no problems—the *DRAT* proof system is extremely powerful, and can in principle justify such reasoning and much more with at most a polynomial amount of work (Sinz & Biere, 2006; Heule, Hunt Jr., & Wetzler, 2015; Philipp & Rebola-Pardo, 2016)—in practice the overhead seems completely prohibitive. Thus, a key challenge on the road to efficient proof logging for more general combinatorial optimization solvers would seem to be to design a method that can capture the full range of techniques used in modern SAT solvers.

1.1 Our Contribution

In this work, we present a new, efficient proof logging method for parity reasoning that is—perhaps somewhat surprisingly—based on pseudo-Boolean reasoning with 0-1 integer linear inequalities. Though such inequalities might seem ill-suited to representing XOR constraints, this can be done elegantly by introducing auxiliary so-called *extension variables* (Dixon, Ginsberg, & Parkes, 2004). Using this observation, we strengthen the *VeriPB* tool[2] recently introduced by Elffers, Gocht, McCreesh, and Nordström (2020), which can be viewed as a generalization to pseudo-Boolean proofs of *RUP* (Goldberg & Novikov, 2003).

1. `http://www.satcompetition.org`.
2. `https://gitlab.com/MIAOresearch/VeriPB`.

Borrowing inspiration from Heule, Kiesl, and Biere (2017), Buss and Thapen (2019), we develop stronger, but still efficient, rules that can handle also extension variables, making *VeriPB*, in effect, into a strict generalization of *DRAT*.

We have implemented our method for representing XOR constraints and performing Gaussian elimination on such constraints in a library with a simple, clean interface for SAT solvers. As a proof of concept, we have also integrated it in *MiniSat* (Eén & Sörensson, 2004), which still serves as the foundation of many state-of-the-art SAT solvers. Our library also provides *DRAT* proof logging for XORs as described by Philipp and Rebola-Pardo (2016), but with some optimizations, to allow for a comparative evaluation. Our experiments show that the overhead for proof logging, the size of the produced proofs, and the time for verification all go down by orders of magnitude for our pseudo-Boolean method compared to *DRAT*. Furthermore, the fact that PB reasoning forms the basis for solvers like *Sat4j* (Le Berre & Parrain, 2010) and *RoundingSat* (Elffers & Nordström, 2018) means that our library can also empower such pseudo-Boolean solvers to reason with parities.

Since cardinality constraints are just a special case of PB constraints, it is clear that our method should suffice to justify the cardinality reasoning used in SAT solvers. The method presented in this paper is not sufficient for efficient proof logging of general symmetry breaking,but at least we can perform as efficiently for symmetry breaking as any approach using DRAT, since our proof system subsumes DRAT. More excitingly, the original *VeriPB* tool has already been shown to be capable of efficiently justifying a number of constraint programming techniques (Elffers et al., 2020; Gocht et al., 2020b, 2020a). Our optimistic interpretation is that pseudo-Boolean reasoning with extension variables shows great potential as a unified method of proof logging for SAT solving, pseudo-Boolean optimization, MaxSAT solving, constraint programming, and maybe even mixed integer programming.

1.2 Subsequent Developments

The last couple of years have witnessed quite significant developments in proof logging. Since the conference version of this paper appeared, our pseudo-Boolean proof logging method has been extended further to deal with fully general symmetry breaking in SAT solving (Bogaerts et al., 2022), and also to support pseudo-Boolean solving using SAT solvers (Gocht et al., 2022). Furthermore, there have been promising preliminary results on providing proof logging for MaxSAT solvers (Vandesande, Wulf, & Bogaerts, 2022) and constraint programming solvers (Gocht, McCreesh, & Nordström, 2022).

The *DRAT* proof logging method has recently been extended to *FRAT* (Baek, Carneiro, & Heule, 2021), which allows to integrate different forms of reasoning. Proof logging using binary decision diagrams (BDDs) (Bryant, 2022), generating proofs in all of the *DRAT*, *LRAT*, and *FRAT* formats, has also been developed for pseudo-Boolean reasoning (Bryant, Biere, & Heule, 2022) and parity reasoning (Soos & Bryant, 2022). Further evaluation will be needed to decide whether such clausal proof logging methods can be truly competitive with pseudo-Boolean proof logging.

1.3 Organization of This Paper

After some brief background in Section 2, we introduce the key technical notions needed for our new proof logging rules in Section 3 and show how they can be used to justify parity

reasoning in Section 4 with a worked out example in Section 5. We present an experimental evaluation in Section 6 and provide some concluding remarks in Section 7.

2. Preliminaries

Let us start by quickly reviewing the required material on pseudo-Boolean reasoning, referring the reader to, e.g., Buss and Nordström (2021) for more context. A few pieces of standard notation are that we write $\mathbb{N} = \{0, 1, 2, \ldots\}$ to denote non-negative integers and $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$ to denote positive integers. For $n \in \mathbb{N}^+$, we write $[n] = \{1, 2, \ldots, n\}$ to denote the set consisting of the first n positive integers.

A *literal* ℓ over a Boolean variable x is x itself or its negation $\overline{x} = 1 - x$, where variables take values 0 (false) or 1 (true). For notational convenience, we define $\overline{\overline{x}} = x$. A *pseudo-Boolean (PB) constraint* C over literals $\ell_1, \ldots, \ell_n$ is a 0-1 linear inequality

$$\sum_{i=1}^{n} a_i \ell_i \geq A \ , \tag{2.1}$$

which without loss of generality we always assume to be in *normalized form*; 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. Conversion to normalized form can be performed efficiently by using equalities $\overline{x} = 1 - x$ to rewrite the left-hand side of any inequality as a positive linear combination of literals, and so in what follows we will consider any pseudo-Boolean constraint and its normalized form to be one and the same constraint. We will use equality

$$\sum_{i=1}^{n} a_i \ell_i = A \tag{2.2a}$$

as syntactic sugar for the pair of inequalities

$$\sum_{i=1}^{n} a_i \ell_i \geq A \tag{2.2b}$$

$$\sum_{i=1}^{n} -a_i \ell_i \geq -A \tag{2.2c}$$

(but rewritten in normalized form) and the *negation* $\neg C$ of (2.1) is (the normalized form of)

$$\sum_{i=1}^{n} -a_i \ell_i \geq -A + 1 \ . \tag{2.3}$$

A *pseudo-Boolean formula* is a conjunction $F = \bigwedge_{j=1}^{m} C_j$ of pseudo-Boolean 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 formulas in *conjunctive normal form (CNF)* are special cases of pseudo-Boolean formulas.

A *(partial) assignment* is a (partial) function from variables to $\{0, 1\}$ and a substitution is a (partial) function from variables to literals or $\{0, 1\}$. For an assignment or substitution ρ we will use the convention $\rho(x) = x$ for x not in the domain of ρ, denoted $x \notin \mathit{dom}(\rho)$, and

define $\rho(\overline{x}) = 1 - \rho(x)$. We also write $x \mapsto b$ instead of $\rho(x) = b$, where b denotes 0, 1, or a literal, when ρ is clear from context or is immaterial. Applying ρ to a pseudo-Boolean constraint C as in (2.1), denoted $C{\restriction_\rho}$, yields the constraint obtained by substituting values for all assigned variables, shifting constants to the right-hand side, and adjusting the degree appropriately, i.e.,

$$C{\restriction_\rho} \;=\; \textstyle\sum_i a_i \rho(\ell_i) \geq A \tag{2.4}$$

with appropriate normalization, and for a formula F we define $F{\restriction_\rho} = \bigwedge_j C_j{\restriction_\rho}$. The normalized constraint C is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq A$ (or, equivalently, if the restricted constraint (2.4) has a non-positive degree and is thus trivial). A PB formula is satisfied by ρ if all constraints in it are, in which case it is *satisfiable*. If there is no satisfying assignment, the formula is *unsatisfiable*. Two formulas are *equisatisfiable* if they are both satisfiable or both unsatisfiable.

The *cutting planes* proof system as defined by Cook, Coullard, and Turán (1987) is a method for iteratively deriving new constraints C implied by a pseudo-Boolean formula F. Cutting planes contains rules for *literal axioms*

$$\frac{}{\ell_i \geq 0}\ , \tag{2.5}$$

and *linear combinations*

$$\frac{\sum_i a_i \ell_i \geq A \qquad \sum_i b_i \ell_i \geq B}{\sum_i (c_A a_i + c_B b_i)\ell_i \geq c_A A + c_B B} \quad [\, c_A, c_B \in \mathbb{N}\,]\ . \tag{2.6}$$

For notational convenience, in this paper we will sometimes use linear combinations of equalities as in (2.2a), which is just a shorthand for taking pairwise linear combinations of inequalities of the form (2.2b) and (2.2c), respectively. There is also a rule for *division*

$$\frac{\sum_i a_i \ell_i \geq A}{\sum_i \lceil a_i/c \rceil \ell_i \geq \lceil A/c \rceil} \quad [\, c \in \mathbb{N}^+\,] \tag{2.7}$$

(where we note that the soundness of this rule depends on that the pseudo-Boolean constraint is written in normalized form). As a toy example, the derivation

$$\cfrac{\cfrac{6x + 2y + 3z \geq 5 \qquad x + 2y + w \geq 1}{8x + 6y + 3z + 2w \geq 7}\ \text{Linear combination } (c_A = 1,\ c_B = 2)}{3x + 2y + z + w \geq 3}\ \text{Division } (c = 3) \tag{2.8}$$

illustrates how these rules can be combined to obtain new constraints.

The proof system that we use for the proof logging in *VeriPB* also supports additional rules such as the *saturation rule*

$$\frac{\sum_i a_i \ell_i \geq A}{\sum_i \min(a_i, A) \cdot \ell_i \geq A}\ , \tag{2.9}$$

which is not part of the cutting planes proof system defined by Cook et al. (1987) but was introduced in the context of pseudo-Boolean solving by Chai and Kuehlmann (2005). For example, from the constraint $8x + 6y + 3z + 2w \geq 7$ in the example above it is possible to derive $7x + 6y + 3z + 2w \geq 7$ via saturation. While this might not be clear from a small

example like this, the division and saturation rules are incomparable in strength (Gocht, Nordström, & Yehudayoff, 2019).

For pseudo-Boolean 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 not hard to see that any collection of constraints F' derived (iteratively) from F by cutting planes are implied in this sense, and so it holds that F and $F \wedge F'$ are equisatisfiable. A particularly simple type of implication is when a constraint C' can be derived from some other constraint C using only addition of literal axioms as in (2.5). When this is is the case, we will say that C' is *implied syntactically* by C.

A constraint C is said to *unit propagate* the literal ℓ under ρ if $C|_\rho$ cannot be satisfied unless $\ell \mapsto 1$. During *unit propagation* on F under ρ, we extend ρ iteratively by any propagated literals $\ell \mapsto 1$ 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 referred to as a *conflict*, since ρ' *violates* the constraint C in this case, and ρ' is called a *conflicting* assignment.

Using the generalization of (Goldberg & Novikov, 2003) by Elffers et al. (2020), we say that F implies C by *reverse unit propagation (RUP)*, and write $RUP(F, C)$, if $F \wedge \neg C$ unit propagates to conflict under the empty assignment. It is not hard to see that $RUP(F, C)$ implies $F \models C$, but the opposite direction is not necessarily true. Cutting planes as defined above is not only *sound* in the sense that it can only derived implied constraints, but it is also *implicationally complete*, which means that if a pseudo-Boolean formula F implies a constraint C, then there is also a cutting planes derivation of C from F. This holds, in particular, if C is RUP with respect to F. However, it might not always be obvious how to construct such a derivation, and therefore we can add a derivation rule for adding RUP constraints as a convenient shorthand. An important special case of completeness is that if a set of pseudo-Boolean constraints F is unsatisfiable, then there exists a cutting planes derivation of the contradiction $0 \geq 1$ from F, which we refer to as a *proof of unsatisfiability*, or *refutation*, of F.

3. Redundance-Based Strengthening

In order to provide proof logging for parity reasoning, we need the ability not only to perform cutting planes reasoning, but also to introduce *fresh variables* not occurring in the formula F under consideration. In particular, we want to be able to use a fresh variable y to encode the *reification* of a constraint $\sum_i a_i \ell_i \geq A$, i.e., that y is true if and only if the constraint is satisfied. We will use the shorthand

$$y \Leftrightarrow \textstyle\sum_i a_i \ell_i \geq A \tag{3.1}$$

to denote the two constraints

$$A\overline{y} + \textstyle\sum_i a_i \ell_i \geq A \tag{3.2a}$$

$$\left(-A+1+\textstyle\sum_i a_i\right) \cdot y + \textstyle\sum_i a_i \overline{\ell}_i \geq -A + 1 + \textstyle\sum_i a_i \tag{3.2b}$$

enforcing this condition (which is the case under the the assumption that the constraint $\sum_i a_i \ell_i \geq A$ is written in normalized form). By way of a concrete example, the reification

of the constraint

$$x_1 + x_2 + x_3 \geq 2 \tag{3.3}$$

using y is encoded as

$$2\overline{y} + x_1 + x_2 + x_3 \geq 2 \tag{3.4a}$$
$$2y + \overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 2 \tag{3.4b}$$

in pseudo-Boolean form. Note that introducing such constraints maintains equisatisfiability provided that the *reification variable* y does not appear in any other constraint, since any assignment to the literals ℓ_i will satisfy either (3.2a) or (3.2b), which allows us to assign y so that the other constraint is also satisfied.

More generally, it would be convenient to allow the "derivation" of any constraint C from F such that F and $F \wedge C$ are equisatisfiable—in which case we say that C is *redundant with respect to* F—regardless of whether $F \models C$ holds or not. A moment of thought reveals that such a completely generic rule would be too good to be true—for any unsatisfiable formula F we would then be able to "derive" contradiction (say, $0 \geq 1$) in just one step, and such derivations would be hard to check for correctness. What we need, therefore, is a sufficient criterion for redundancy of pseudo-Boolean constraints that is simple to verify. To this end, we generalize the characterization of redundancy by Heule et al. (2017), Buss and Thapen (2019) from CNF formulas to pseudo-Boolean formulas as follows.

Proposition 3.1 (Substitution redundancy)**.** *A pseudo-Boolean constraint C is redundant with respect to the formula F if and only if there is a substitution ω, called a* witness*, for which it holds that*

$$F \wedge \neg C \models (F \wedge C)\restriction_\omega \ .$$

Proof. ($\Rightarrow$) Suppose C is redundant. If F is unsatisfiable, then for any constraint C' it vacuously holds that $F \models C'$. Hence, any substitution ω fulfils the condition. If F is satisfiable, then $F \wedge C$ must also be satisfiable as C is redundant by assumption. If we choose ω to be a satisfying assignment for $F \wedge C$, the implication in the proposition again vacuously holds since $(F \wedge C)\restriction_\omega$ is fixed to true.

($\Leftarrow$) Suppose now that ω is such that $F \wedge \neg C \models (F \wedge C)\restriction_\omega$. If F is unsatisfiable, then every constraint is redundant and there is nothing to check. Otherwise, let α be a (total) satisfying assignment for F. If α also satisfies C, then clearly the constraint is redundant. Now consider the case that α does not satisfy C. If so, α must satisfy $\neg C$ and hence, by the assumed implication, also $(F \wedge C)\restriction_\omega$. But then the assignment β defined by

$$\beta(x) = \begin{cases} \alpha(x) & \text{if } x \notin \mathit{dom}(\omega), \\ \alpha(\omega(x)) & \text{otherwise,} \end{cases} \tag{3.5}$$

satisfies both C and F (since $(F \wedge C)\restriction_\beta = ((F \wedge C)\restriction_\omega)\restriction_\alpha$ by construction), so $F \wedge C$ is satisfiable. □

We remark that this proof does not make use of that we are operating with a pseudo-Boolean constraint C—we only need that the negation $\neg C$ is easy to represent in the same formalism. Thus, the argument generalizes to other types of constraints with this property

(such as, for instance, polynomial equations over finite fields when evaluated on Boolean inputs $\{0,1\}^n$, as in the *polynomial calculus* proof system (Clegg, Edmonds, & Impagliazzo, 1996; Alekhnovich, Ben-Sasson, Razborov, & Wigderson, 2002) formalizing Gröbner basis computations).

Let us return to our example reification of the constraint in (3.3) and show how this can be derived using substitution redundancy. Let us write $C_{3.4a}$ for the constraint in (3.4a) and $C_{3.4b}$ for (3.4b), where y is fresh with respect to the current formula F. To show that $C_{3.4b}$ is substitution redundant with respect to F we choose the witness $\omega = \{y \mapsto 1\}$, which clearly satisfies $C_{3.4b}$. Since y does not appear in F we have $F\restriction_\omega = F$, and so the implication $F \wedge \neg C_{3.4b} \models (F \wedge C_{3.4b})\restriction_\omega$ vacuously holds. Showing that $C_{3.4a}$ is substitution redundant with respect to $F \wedge C_{3.4b}$ is a bit more interesting. For this we choose $\omega = \{y \mapsto 0\}$, which satisfies $C_{3.4a}$ and again leaves F unchanged. Thus, the only implication for which we need to do some work is $F \wedge C_{3.4b} \wedge \neg C_{3.4a} \models C_{3.4b}\restriction_\omega$. The negation of $C_{3.4a}$ is

$$-2\overline{y} - x_1 - x_2 - x_3 \geq -1 \ , \tag{3.6a}$$

or, converted to normalized form,

$$2y + \overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 4 \tag{3.6b}$$

using the rewriting rule $\ell = 1 - \overline{\ell}$. Adding the literal axiom $\overline{y} \geq 0$ twice to $\neg C_{3.4a}$, and using rewriting again to cancel $y + \overline{y} = 1$, we obtain

$$\overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 2 \ , \tag{3.7}$$

which is $C_{3.4b}\restriction_\omega$. Hence, $C_{3.4b}\restriction_\omega$ can be derived from $\neg C_{3.4a}$ by just adding literal axioms—or, in the terminology introduced in the preliminaries, $\neg C_{3.4a}$ syntactically implies $C_{3.4b}\restriction_\omega$—and so it certainly holds that $F \wedge C_{3.4b} \wedge \neg C_{3.4a} \models C_{3.4b}\restriction_\omega$. This completes the proof that $C_{3.4a}$ is redundant with respect to $F \wedge C_{3.4b}$.

In our proof system for pseudo-Boolean proof logging, we will include a *redundance-based strengthening*[3] rule that allows to derive constraints that satisfy the condition in Proposition 3.1. In order to do so, we need to discuss how the implication in this substitution redundancy condition is to be verified. Whenever this rule is used, the user needs to explicitly specify a witness ω, but this is not enough. Arbitrary implication checks are as hard to verify as determining satisfiability of a formula, and hence some kind of efficiently verifiable certificate that the implication indeed holds is necessary to be able to validate the proof. One way of providing such a certificate is to exhibit a cutting planes derivation establishing the validity of the implication, as in the example just presented. A more convenient alternative from a proof logging point of view is to follow the lead of *DRAT* and allow adding constraints without proof if the implication can be verified automatically, e.g., using reverse unit propagation. We describe our pseudo-Boolean version of this automatic verification method in Algorithm 1. It is easy to see that the condition in Proposition 3.1 is satisfied if Algorithm 1 issues a positive verdict: If the algorithm accepts because of

3. In the conference version (Gocht & Nordström, 2021) of this paper, this rule was called *substitution redundancy*. However, since then an additional rule using witness substitutions has been introduced by Bogaerts et al. (2022), and we follow the terminology in this later paper to adhere to a consistent naming scheme.

Algorithm 1 Automatically checking substitution redundancy of C with respect to F

```
1: procedure RedundancyCheck(F, C, ω)          ▷ C, ω are given in the proof log
2:     if RUP(F, C) then return ACCEPT
3:     for D ∈ (F ∧ C)↾ω do
4:         if not (D ∈ F or ¬C implies D syntactically or RUP(F ∧ ¬C, D)) then
5:             return REJECT
6:     return ACCEPT
```

$RUP(F, C)$, then $F \models C$ and it is in order to add the constraint. Otherwise, the algorithm will reject unless for all constraints D in $(F \wedge C)\restriction_\omega$, i.e., all constraints on the right hand side of the implication in Proposition 3.1, it holds that (a) $D \in F$, (b) $\neg C$ implies D syntactically, or (c) $RUP(F \wedge \neg C, D)$ evaluates to true. In all three cases it follows that $F \wedge C \models D$, as desired.

We remark that this algorithm is very similar to what is used for checking RAT clauses in $DRAT$ proof verification, except that our unit propagation is on PB constraints rather than clauses and that we need the additional syntactic check on line 4 in Algorithm 1. To see why this extra step is necessary, note that if we used only unit propagation, then we would fail to certify the correctness of our example above. Assuming for simplicity that $F = \emptyset$, if we try to verify $C_{3.4b} \wedge \neg C_{3.4a} \models C_{3.4b}\restriction_\omega$ by reverse unit propagation we get the constraints

$$2y + \overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 2 \qquad [\, C_{3.4b} \text{ in (3.4b)} \,] \tag{3.8a}$$

$$2y + \overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 4 \qquad [\, \neg C_{3.4a} \text{ in (3.6b)} \,] \tag{3.8b}$$

$$x_1 + x_2 + x_3 \geq 2 \qquad [\, \text{negation of desired RUP constraint } \neg(C_{3.4b}\restriction_\omega) \,] \tag{3.8c}$$

and although visual inspection shows that this collection of constraints is inconsistent, since it requires a majority of the variables $\{x_1, x_2, x_3\}$ to be true and false at the same time, unit propagation is too myopic to see this contradiction and only yields $y \mapsto 1$. Thanks to the fact that we instead use the stronger checks in Algorithm 1, we can automatically detect that the implication $C_{3.4b} \wedge \neg C_{3.4a} \models C_{3.4b}\restriction_\omega$ holds. This means that to introduce extension variables encoding reifications $y \Leftrightarrow C$, we do not need to do anything more than just specifying witness assignments to the new variable as in the example above for constraints (3.4a) and (3.4b). For completeness, we write out the details in the general case for the constraints (3.2a) and (3.2b) in the next proposition.

Proposition 3.2. *Let F be a pseudo-Boolean formula and C be a pseudo-Boolean constraint, and suppose y is a fresh variable that does not appear in F or C. Then the constraints* (3.2a) *and* (3.2b) *encoding $y \Leftrightarrow C$ can both be derived and added to F by the redundance-based strengthening rule using Algorithm 1 to verify the substitution redundancy conditions.*

Proof. Let us write $C_{3.2a}$ for the constraint in (3.2a) and $C_{3.2b}$ for (3.2b). To show that $C_{3.2b}$ is substitution redundant with respect to F we choose the witness $\omega = \{y \mapsto 1\}$, which clearly satisfies $C_{3.2b}$. Since the negation of a satisfied constraint is contradiction, this means, technically speaking, that $C_{3.2b}\restriction_\omega$ is RUP with respect to F. Since y does not

appear in F we have $D\restriction_\omega = D$ for all $D \in F$, which means that all constraints pass the check on Line 4 in Algorithm 1.

As in our example above, showing that $C_{3.2a}$ is substitution redundant with respect to $F \wedge C_{3.2b}$ requires slightly more work. Here we choose the witness $\omega = \{y \mapsto 0\}$, which satisfies $C_{3.2a}$ and again leaves F unchanged, which means that implication checks for constraints in F are again vacuous. The only constraint left to check is $C_{3.2b}\restriction_\omega$, which is implied syntactically by $\neg C_{3.2a}$, as we will see next. The negation of $C_{3.2a}$ is

$$A\overline{y} + \textstyle\sum_i a_i\ell_i \leq A - 1 \tag{3.9a}$$

or

$$-A\overline{y} + \textstyle\sum_i - a_i\ell_i \geq -A + 1 \ , \tag{3.9b}$$

which in normalized form becomes

$$Ay + \textstyle\sum_i a_i\overline{\ell}_i \geq 1 + \sum_i a_i \tag{3.9c}$$

(rewriting using the equality $\ell = 1 - \overline{\ell}$ to obtain a positive linear combination of literals on the left-hand side of the inequality). Adding A times the literal axiom $\overline{y} \geq 0$ to $\neg C_{3.2a}$ and applying cancellation $y + \overline{y} = 1$, we obtain

$$\textstyle\sum_i a_i\overline{\ell}_i \geq -A + 1 + \sum_i a_i \ , \tag{3.10}$$

which is $C_{3.2b}\restriction_\omega$. Hence, $\neg C_{3.2a}$ syntactically implies $C_{3.2b}\restriction_\omega$, and so the condition on Line 4 is satisfied.

This concludes the proof that the conditions required to derive the constraints $C_{3.2b}$ and $C_{3.2a}$ by redundance-based strengthening can be checked efficiently by Algorithm 1 regardless of what the pseudo-Boolean formula F is. □

4. Proof Logging for XOR Constraints

We now proceed to explain how the cutting planes proof system in Section 2 extended with the redundance-based strengthening rule in Section 3 can be used to certify the correctness of parity reasoning.

An *XOR* or *parity constraint*, i.e., an equality modulo 2, over k Boolean variables is written as

$$x_1 \oplus x_2 \oplus \cdots \oplus x_k = b \tag{4.1}$$

for $b \in \{0, 1\}$. Note that we can assume that there is no parity constraint with a negated variable $\overline{x}$, because we can always substitute $\overline{x} = x \oplus 1$. Systems of XOR constraints can be handled in a solver through Gaussian elimination (Soos et al., 2009; Han & Jiang, 2012; Laitinen et al., 2012b) or conflict analysis (Laitinen et al., 2012a). In this paper we will focus on the integration of Gaussian elimination into conflict-driven clause learning (CDCL) (Bayardo Jr. & Schrag, 1997; Marques-Silva & Sakallah, 1999; Moskewicz et al., 2001), and so we start by a quick review of how the CDCL main loop works and how parity reasoning is included. The reader can consult the pseudocode in Algorithm 2 to complement the description below.

Algorithm 2 Conflict-driven clause learning with parity reasoning

```
1: procedure SOLVE(F)
2:     trail ← ∅
3:     G ← detectParities(F)
4:     D ← F
5:     while True do
6:         (ℓ, C_reason) ← propagate(D, trail)
7:         if ℓ = NULL then (ℓ, C_reason) ← propagateXOR(G, trail)
8:         if ℓ = NULL then (ℓ, C_reason) ← (nextDecision(), NULL)
9:         if ℓ = NULL then return SATISFIABLE
10:        trail.push(ℓ, C_reason)
11:        if hasConflict(trail) then
12:            C_learned ← analyse(trail)
13:            if C_learned = ⊥ then
14:                return UNSATISFIABLE
15:            else
16:                D ← D ∪ {C_learned}
17:                trail ← backjump(trail)
```

Let us first describe CDCL without parity reasoning, i.e., without the boldface italicized code on lines 3 and 7. When run on a formula F, the CDCL solver has a *database* $\mathcal{D}$ of clauses, which is initialized to the clauses in F. The solver also maintains a *trail* consisting of an ordered list of literals assigned to true together with reasons for these assignments. In what follows, it will be convenient to identify the trail with the assignment setting the literals on the trail to true. The trail is initialized to be empty.

The solver adds assigned literals to the trail, one by one, according to the following procedure. If some clause $C_{\text{reason}} \in \mathcal{D}$ unit propagates an unassigned literal ℓ in the sense explained in Section 2 (which for a clause C_{reason} means that all literals in the clause except ℓ are falsified by the current trail), then the trail is extended by adding ℓ with C_{reason} as the *reason clause* explaining the propagation. (If several clauses propagate at the same time, then ties will be split in a somewhat arbitrary fashion depending on low-level details in the algorithm implementation.) Otherwise, the solver uses a decision heuristic to pick some literal to assign to true. Such a decision literal has no reason clause. If there is no literal left to assign, then this means that all variables have been assigned without violating any clause in F. In other words, a satisfying assignment has been found, and so the solver returns that the formula is satisfiable. Assuming instead that some literal has been added to the trail, this literal can lead to some clause $D \in \mathcal{D}$ being falsified by the trail. This is referred to as a *conflict* with D as the *conflict clause*. When a conflict arises, a *conflict analysis* algorithm is called to derive a new clause C_{learned} from the conflict clause and the reason clauses currently on the trail. If the result of this conflict analysis is the empty clause $\bot$ without any literals, then contradiction has been derived and the solver returns that the formula is unsatisfiable. Otherwise, the clause C_{learned} is *learned*, i.e., added to the clause database, after which the solver *backjumps* by removing some literals from the

trail until C_{learned} is no longer falsified. The details of exactly how backjumps are done are not relevant for our proof logging discussion, and there are also other details of CDCL that we are ignoring in this description, such as that the solver sometimes does *restarts* (which means resetting the trail to be empty) and sometimes performs *database reduction* (removing learned clauses from $\mathcal{D}$).

To add parity reasoning to CDCL, the solver is modified by first detecting implicit parity constraints in the CNF formula on line 3 in Algorithm 2. This can be done by checking syntactically if all clauses in the canonical clausal encoding of a parity constraint are present. For instance, the clauses

$$\{x_1 \vee x_2 \vee x_3,\ \overline{x}_1 \vee \overline{x}_2 \vee x_3,\ \overline{x}_1 \vee x_2 \vee \overline{x}_3,\ x_1 \vee \overline{x}_2 \vee \overline{x}_3\} \tag{4.2a}$$

encode the parity constraint

$$x_1 \oplus x_2 \oplus x_3 = 1\ . \tag{4.2b}$$

Parity constraints detected in this way can then be used for Gaussian elimination, which generates new parity constraints. If all variables in a parity constraint except one is assigned by the trail, then the final variable is propagated to a value on line 7. It can also happen that a parity constraint is violated by the current trail, and detection of this condition is included on line 11. In both of these cases, the solver will need a reason or conflict clause, respectively, to justify the steps taken. Such a clause can be computed from the parity constraint in a straightforward way. Suppose, for example that from parity constraints $x_1 \oplus x_2 \oplus x_3 = 1$ and $x_2 \oplus x_3 \oplus x_4 = 1$ Gaussian elimination has derived $x_1 \oplus x_4 = 0$, and suppose also that x_1 is assigned to true on the trail. Then x_4 will also propagate to true, and the reason clause provided for this will be $\overline{x}_1 \vee x_2$.

There are many variations on how this general idea can be implemented. For instance, parity detection can also be run later during the search over the clause database $\mathcal{D}$, as done in *CryptoMiniSat*. Another interesting question studied by Yang and Meel (2021) is whether it is better to propagate all clauses first (as in our pseudocode here) or all parity constraints first, or if the propagation on different types of constraints should be interleaved. However, such aspects are not relevant to how proof logging for parity reasoning should be designed, and our description in Algorithm 2 has been chosen mainly to make the exposition simple.

To provide proof logging for CDCL solvers with Gaussian elimination, we will need the four ingredients listed below:

1. **XOR encoding:** An efficient encoding of parity constraints as linear pseudo-Boolean constraints.

2. **XOR reasoning:** A method of deriving (the pseudo-Boolean encoding of) a new parity constraint from existing parity constraints.

3. **Reason and conflict clause generation:** The ability to prove the validity of reason and conflict clauses from the pseudo-Boolean encoding of parity constraints when such parity constraints give rise to propagations or conflicts, respectively.

4. **Translation from CNF:** A way of translating clausal encodings of parity constraints to pseudo-Boolean form (which is where we will need to go beyond cutting planes by using extension variables and redundance-based strengthening).

We will describe these components in detail in the rest of this section. In Section 5, we will then provide a worked-out example to illustrate how everything comes together to yield a method for CDCL solving with parity constraints.

4.1 Linear Pseudo-Boolean Encoding of Parity Constraints

Our encoding of parity constraints in linear pseudo-Boolean form is based on the observation by Dixon et al. (2004) that for any partial assignment to the variables $x_1, \ldots, x_k$, the parity constraint $x_1 \oplus x_2 \oplus \cdots \oplus x_k = b$ as in (4.1) is satisfiable if and only if the $0-1$ integer linear equality

$$\sum_{i \in [k]} x_i = b + \sum_{i \in [\lfloor k/2 \rfloor]} 2y_i \tag{4.3}$$

is satisfiable, where $y_1, \ldots, y_{\lfloor k/2 \rfloor}$ are fresh variables not appearing in other constraints. Since the variables y_i are otherwise unconstrained, the right-hand side can take any even (odd) value for $b = 0$ ($b = 1$) in the range from 0 to k, and these are exactly the values that we want to allow for $\sum_{i \in [k]} x_i$. Recalling that any equality on the form (2.2a) can be represented with the two inequalities (2.2b) and (2.2c), we have obtained a representation of parity constraints as linear pseudo-Boolean inequalities.

In fact, we can generalize this by observing that if we let $\mathcal{B}$ denote any integer linear combination of variables, possibly also with a constant term, then the two inequalities

$$\sum_{i \in [k]} x_i \geq b + 2\mathcal{B} \tag{4.4a}$$

$$\sum_{i \in [k]} -x_i \geq -b - 2\mathcal{B} \tag{4.4b}$$

forming the equality $\sum_{i \in [k]} x_i = b + 2\mathcal{B}$ imply the parity constraint

$$x_1 \oplus x_2 \oplus \cdots \oplus x_k = b \ . \tag{4.4c}$$

We will make repeated use of this observation below.

4.2 XOR Reasoning Using Pseudo-Boolean Constraints

Whenever we want to combine two XOR constraints to derive a new XOR constraint as is done during Gaussian elimination, we only need to add the pseudo-Boolean equalities corresponding to these two XOR constraints. Consider again our example derivation

$$\frac{x_1 \oplus x_2 \oplus x_3 = 1 \qquad x_2 \oplus x_3 \oplus x_4 = 1}{x_1 \oplus x_4 = 0} \tag{4.5}$$

from before, and assume that the two premises are represented in pseudo-Boolean form as

$$x_1 + x_2 + x_3 = 2y_1 + 1 \tag{4.6a}$$

and

$$x_2 + x_3 + x_4 = 2y_2 + 1 \tag{4.6b}$$

for fresh variables y_1 and y_2. Then adding both equalities together we obtain

$$x_1 + 2x_2 + 2x_3 + x_4 = 2y_1 + 2y_2 + 2 \tag{4.6c}$$

which implies the desired XOR constraint by the observation we just made regarding (4.4a)–(4.4c). (Recall that a linear combination of equalities as in (2.2a) is a notational shorthand for taking pairwise linear combinations of inequalities (2.2b) and (2.2c).)

4.3 Reason and Conflict Clause Generation from XOR Constraints

As explained above, CDCL solvers justify all propagation and conflict analysis steps using clauses. If we want to use XOR constraints to propagate forced variable assignments or derive contradiction, then we need to provide clauses that justify such derivation steps, together with proof logging steps explaining why these clauses are valid. We next show how to derive such clauses from pseudo-Boolean encodings of XOR constraints.

Suppose we have a parity constraint encoded by inequalities of the form (4.4a)–(4.4b), and let ρ be an assignment to the variables $x_1, \dots, x_k$ that is inconsistent with these inequalities because it falsifies the implied XOR constraint (4.4c). We want to derive from (4.4a)–(4.4b) a clause that is falsified under ρ. Let

$$\mathcal{F}(\rho) = \{i \in [k] \mid \rho(x_i) = 0\} \tag{4.7}$$

be the set of indices of variables assigned to false by ρ and

$$\mathcal{T}(\rho) = \{j \in [k] \mid \rho(x_j) = 1\} \tag{4.8}$$

the indices of variables assigned to true . Using the literal axiom rule (2.5) we can derive (the normalized form of) the trivially true constraint

$$\sum_{i \in \mathcal{F}(\rho)} x_i + \sum_{j \in \mathcal{T}(\rho)} -x_j \geq -|\mathcal{T}(\rho)| \ , \tag{4.9}$$

which when added to (4.4a) yields

$$\sum_{i \in \mathcal{F}(\rho)} 2x_i \geq b - |\mathcal{T}(\rho)| + 2\mathcal{B} \ . \tag{4.10}$$

By assumption, we have that $b - |\mathcal{T}(\rho)|$ is odd, since otherwise ρ would not falsify the XOR constraint implied by (4.4a)–(4.4b). All other terms in the inequality (4.10) are divisible by 2. Hence, even though (4.10) is not presented in normalized form, we can see that if we apply the division rule (2.7) with divisor 2, this will round up and increase the degree of falsity. This means that if we divide the constraint (4.10) and then multiply by 2 (which is just a special case of the linear combination rule (2.6)), we get

$$\sum_{i \in \mathcal{F}(\rho)} 2x_i \geq b - |\mathcal{T}(\rho)| + 1 + 2\mathcal{B} \ . \tag{4.11}$$

We continue by adding (4.4b) to get

$$\sum_{i \in \mathcal{F}(\rho)} x_i - \sum_{j \in \mathcal{T}(\rho)} x_j \geq 1 - |\mathcal{T}(\rho)| \ , \tag{4.12}$$

which is the same constraint as

$$\sum_{i \in \mathcal{F}(\rho)} x_i + \sum_{j \in \mathcal{T}(\rho)} \overline{x}_j \geq 1 \tag{4.13}$$

after normalization. This last constraint, which is a disjunctive clause, is falsified under ρ as desired, and so can serve as the conflict clause justifying why the assignment ρ is inconsistent. Derivations of reason clauses for propagation work in a similar way—essentially, we can pretend that the propagated variable is set to the wrong value and then perform the derivation above to obtain a clause (4.13) that propagates the variable to the right value instead. An example for deriving a reason clause can be found towards the end of Section 5.

4.4 Translating Parity Constraints from CNF to Pseudo-Boolean Form

An XOR constraint as in (4.1) can be encoded into CNF in a canonical way by including for each of the 2^{k-1} assignments falsifying the constraint the disjunctive clause ruling out that assignment. For example, for $k = 3$ and $b = 1$ the parity constrain (4.2b) can be encoded by the clauses in (4.2a), which are written as the 0-1 integer linear inequalities

$$x_1 + x_2 + x_3 \geq 1 \tag{4.14a}$$
$$\overline{x}_1 + \overline{x}_2 + x_3 \geq 1 \tag{4.14b}$$
$$\overline{x}_1 + x_2 + \overline{x}_3 \geq 1 \tag{4.14c}$$
$$x_1 + \overline{x}_2 + \overline{x}_3 \geq 1 \tag{4.14d}$$

in pseudo-Boolean form. Since the number of clauses in this canonical CNF encoding of an XOR constraint scales exponentially with the number of variables, it is only feasible to encode short XORs into CNF in this manner. However, it is possible to split up a long XOR constraint into multiple constant-size XORs using auxiliary variables z_i, which represent the partial parities up to and including x_i, i.e., $z_i = \sum_{j \in [i]} x_j \pmod 2$. In this way, a collection of parity constraints

$$x_1 \oplus x_2 \oplus z_2 = 0 \tag{4.15a}$$
$$z_2 \oplus x_3 \oplus z_3 = 0 \tag{4.15b}$$
$$\vdots$$
$$z_{k-2} \oplus x_{k-1} \oplus x_k = b \tag{4.15c}$$

can be used to represent the constraint (4.1). Assuming that we can split up parity constraints in this manner, we will only need to translate short parity constraints from CNF to pseudo-Boolean form. The original, long, parity constraints can then be recovered by XOR reasoning, just summing up the constraints (4.15a)–(4.15c), and proof logging for this derivation can be done as described in Section 4.2 above.

We perform the translation to the pseudo-Boolean XOR encoding from CNF in two steps, which we will describe in more detail after providing the general idea. The first step is to derive the constraint

$$\sum_{i \in [k]} x_i = \sum_{i \in [\lfloor k/2 \rfloor]} 2y_i + y' \ , \tag{4.16}$$

where y' and y_i, $i \in [\lfloor k/2 \rfloor]$, are all fresh variables. Note that adding the equality constraint (4.16) to any formula does not affect satisfiability, because we can always assign the fresh variables so that this additional constraint holds true. However, although the constraint (4.16) is redundant in the sense of Proposition 3.1, we cannot use the redundance-based strengthening rule to derive the constraint, because we do not have an efficient procedure for constructing a witness ω that is efficiently verifiable by Algorithm 1. Instead, we will introduce the auxiliary variables y' and y_i, $i \in [\lfloor k/2 \rfloor]$, one by one, in a similar fashion to what was done in Section 3. We remark that an alternative to (4.16) would be to encode the sum $2 \cdot \left\lfloor \frac{1}{2} \sum_{i \in [k]} x_i \right\rfloor$ of the x_i-variables rounded down to the nearest even integer as a sum of powers of 2, resulting in an equality constraint

$$\sum_{i \in [k]} x_i = \sum_{i \in [\lceil \log_2(k/2) \rceil]} 2^i y_i + y' \ . \tag{4.17}$$

For parity constraints over a large number of variables, this encoding has a substantially smaller number of auxiliary variables. However, since we are recovering parity constraints from CNF, we only expect to have parity constraints over few variables, as the number of clauses in the CNF encoding is exponential in the number of variables.

The second step, once we have derived the equality constraint (4.16), is to brute-force over all possible assignments to the x_i-variables to derive the equality

$$y' = b \ . \tag{4.18}$$

Summing the equalities (4.16) and (4.18), we obtain a constraint of the desired form (4.3). Note that since we are considering all possible assignments to the x_i-variables, this derivation will require an exponential number of derivation steps measured in the number of variables, but this is still polynomial measured in the number of clauses in the canonical CNF encoding of parity constraints. We now proceed to describe this process in detail.

Step 1a: To derive (4.16) we will construct a chain of 1-bit full adders, as illustrated in Figure 1b for an adder with output carry bit y and sum bit z. Let us start by showing how the encoding of a single adder can be derived. A 1-bit full adder (shown in Figure 1a) computes the sum of three variables x_1, x_2, x_3 and returns the result as a binary number. This can be encoded using the pseudo-Boolean equality

$$2y + z = x_1 + x_2 + x_3 \ . \tag{4.19}$$

Recalling the shorthand (3.1) for the two reification constraints (3.2a) and (3.2b), in order to obtain (4.19) we start by deriving

$$y \Leftrightarrow x_1 + x_2 + x_3 \geq 2 \tag{4.20a}$$

$$z \Leftrightarrow x_1 + x_2 + x_3 - 2y \geq 1 \tag{4.20b}$$

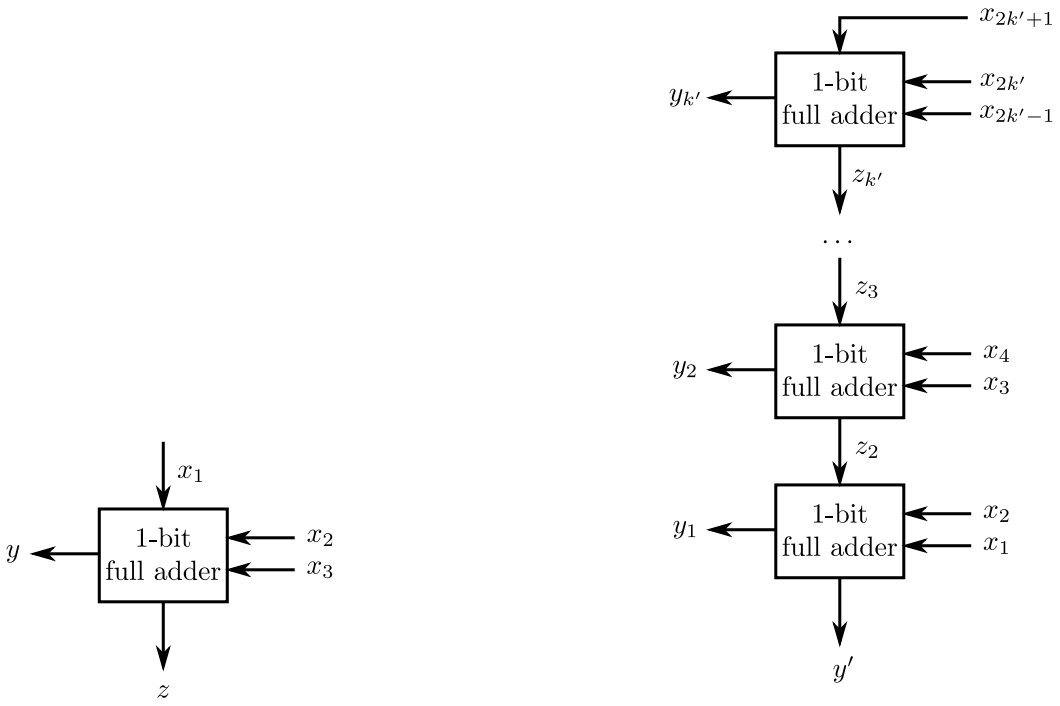
for fresh variables y and z using redundance-based strengthening as described in Proposition 3.2. This means that we have now derived the four constraints

$$2\overline{y} + x_1 + x_2 + x_3 \geq 2 \tag{4.21a}$$

$$2y + \overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 2 \tag{4.21b}$$

$$3\overline{z} + x_1 + x_2 + x_3 + 2\overline{y} \geq 3 \tag{4.21c}$$

$$3z + \overline{x}_1 + \overline{x}_2 + \overline{x}_3 + 2y \geq 3 \tag{4.21d}$$



(a) A 1-bit full adder.

(b) Chain of 1-bit full adders.

Figure 1: Using adders to encode parities of subsets of variables.

when written as pseudo-Boolean inequalities in normalized form. To derive the less-than-or-equal part $2y + z \leq x_1 + x_2 + x_3$ of (4.19), which in normalized form is

$$x_1 + x_2 + x_3 + 2\overline{y} + \overline{z} \geq 3 \ , \tag{4.22a}$$

we take a linear combination of (4.21c) and 2 times (4.21a), followed by division by 3. In a similar fashion, to derive the greater-than-or-equal part $2y + z \geq x_1 + x_2 + x_3$ of (4.19), or

$$\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + 2y + z \geq 3 \tag{4.22b}$$

in normalized form, we add together (4.21d) and 2 times (4.21b) followed by division by 3.

Step 1b: To derive the equality constraint (4.16), we use a chain of 1-bit full adders connected as in Figure 1b, where we set $k' = \lfloor k/2 \rfloor$. The x_i-variables are used as inputs to the adders, and the final variable $x_{2k'+1}$, which appears in the topmost adder, will only be there if the number of variables k is odd. Otherwise, we replace $x_{2k'+1}$ by 0, so that the topmost adder only has $x_{2k'}$ and $x_{2k'-1}$ as input. (Formally, if $x_{2k'+1}$ does not exist, then it is a fresh variable, and so we can derive the equality $x_{2k'+1} = 0$ by redundance-based strengthening before continuing as described below.) The output carry variables y_i, $i \in [k']$, and the final sum bit y' will be used to derive the equality (4.16), while the z_i-variables are intermediate parity bits. We apply the procedure

in Step 1a to all adders to derive PB constraints on the form (4.22a) and (4.22b). After this, for the topmost adder in Figure 1b we have obtained

$$2y_{k'} + z_{k'} = x_{2k'+1} + x_{2k'} + x_{2k'-1} \ , \tag{4.23a}$$

for the intermediate adders the equations

$$2y_i + z_i = z_{i+1} + x_{2i} + x_{2i-1} \tag{4.23b}$$

hold for $i \in \{2, \dots, k'-1\}$, and for the bottom adder we get

$$2y_1 + y' = z_2 + x_2 + x_1 \ . \tag{4.23c}$$

By adding the encoding of all 1-bit adders, i.e., the equalities (4.23a)–(4.23c), we obtain

$$\sum_{i=1}^{k'} 2y_i + y' + \sum_{i=2}^{k'} z_i = \sum_{i=2}^{k'} z_i + \sum_{i=1}^{2k'+1} x_i \ , \tag{4.24}$$

where the sums $\sum_{i=2}^{k'} z_i$ on each side cancel to produce the equality constraint (4.16) as desired.

Step 2: The final step is to fix the value of y' in order to go from (4.16) to our final goal (4.3). That is, writing $y'(b) = y'$ if $b = 1$ and $y'(b) = \overline{y}'$ if $b = 0$, we wish to derive the PB constraint

$$y'(b) \geq 1 \tag{4.25}$$

forcing $y' = b$. We will do so by considering all possible truth value assignments ρ to the variables x_i, $i \in [k]$. In order to present the formal derivation, we first need to set up some notation.

For any assignment ρ to a subset of the variables x_i, $i \in [k]$, let $\mathcal{F}(\rho)$ and $\mathcal{T}(\rho)$ be the indices of variables set to false and true by ρ, respectively, as defined in (4.7) and (4.8). Let us write $C_{\neg\rho}$ to denote the unique clausal constraint

$$\sum_{i \in \mathcal{F}(\rho)} x_i + \sum_{j \in \mathcal{T}(\rho)} \overline{x}_j \geq 1 \tag{4.26}$$

over all variables assigned by ρ that is falsified by this assignment. We also extend this notation in the natural way to let $C_{\neg(\rho \cup \{y'(b) \mapsto 0\})}$ denote the clausal constraint

$$y'(b) + \sum_{i \in \mathcal{F}(\rho)} x_i + \sum_{j \in \mathcal{T}(\rho)} \overline{x}_j \geq 1 \tag{4.27}$$

that is falsified by ρ if in addition $y'(b)$ is set to false, i.e., y' is given the value $1 - b$.

For any assignment ρ such that $\sum_{i \in [k]} x_i \neq b \pmod 2$, we postulated above that the clause $C_{\neg\rho}$ is in the formula, but for our argument here we only need the slightly weaker assumption that this clause can be obtained by reverse unit propagation on the constraints derived so far. Assuming that this holds, we can certainly derive $C_{\neg(\rho \cup \{y'(b) \mapsto 0\})}$ by RUP for all such assignments ρ. If instead ρ is such that

$\sum_{i\in[k]} x_i = b \pmod 2$, then extending ρ by setting $y'(b) = 0$ means that (4.16) can no longer be satisfied, since ρ assigns different parities to the left-hand and right-hand sides of this equality, and no assignment to the y_i-variables in $\sum_{i\in[\lfloor k/2 \rfloor]} 2y_i$ can change this. For such ρ we can therefore proceed as in Section 4.3 to derive the clause $C_{\neg(\rho\cup\{y'(b)\mapsto 0\})}$ explaining why the assignment $\rho \cup \{y'(b) \mapsto 0\}$ is inconsistent.

So far, we have shown how to derive clauses $C_{\neg(\rho\cup\{y'(b)\mapsto 0\})}$ in (4.27) for any assignment ρ to all the x_i-variables. But once we have these clauses, the rest is routine. Let ρ_{k-1} be any partial assignment to the $k-1$ first variables x_i, $i \in [k-1]$. Taking the previously derived constraints $C_{\neg(\rho_{k-1}\cup\{x_k\mapsto 0,\, y'(b)\mapsto 0\})}$ and $C_{\neg(\rho_{k-1}\cup\{x_k\mapsto 1,\, y'(b)\mapsto 0\})}$, which is what we write by mild abuse of notation to denote the clausal constraints

$$y'(b) + x_k + \sum_{i\in\mathcal{F}(\rho_{k-1})} x_i + \sum_{j\in\mathcal{T}(\rho_{k-1})} \overline{x}_j \geq 1 \tag{4.28a}$$

and

$$y'(b) + \overline{x}_k + \sum_{i\in\mathcal{F}(\rho_{k-1})} x_i + \sum_{j\in\mathcal{T}(\rho_{k-1})} \overline{x}_j \geq 1 \ , \tag{4.28b}$$

respectively (which agree on all literals except that the variable x_k appears with opposite signs), adding these constraints, and then dividing by 2 yields

$$y'(b) + \sum_{i\in\mathcal{F}(\rho_{k-1})} x_i + \sum_{j\in\mathcal{T}(\rho_{k-1})} \overline{x}_j \geq 1 \ , \tag{4.29}$$

i.e., the clause $C_{\neg(\rho_{k-1}\cup\{y'(b)\mapsto 0\})}$.[4] We can eliminate the variable x_k in this way by deriving clauses (4.29) for all assignments ρ_{k-1} to x_i, $i \in [k-1]$. (A technical side note is that the constraint (4.29) follows by reverse unit propagation on (4.28a) and (4.28b), and so we could avoid a syntactic derivation by just claiming it as a RUP constraint. However, when there is a simple explicit derivation like above it is often preferable to use such a derivation instead, since this tends to make proof verification faster, and as we will see in Section 5 there is an elegant way of chaining all derivations of this type together on a single proof line.)

Next, we consider all assignments ρ_{k-2} to x_i, $i \in [k-2]$, and repeat the derivation of clauses (4.29) from (4.28a) and (4.28b) to obtain clauses $C_{\neg(\rho_{k-2}\cup\{y'(b)\mapsto 0\})}$ for all ρ_{k-2} (where we replace x_k by x_{k-1} in (4.28a) and (4.28b)). Continuing in this fashion, we eliminate the variables $x_k, x_{k-1}, \ldots, x_1$ one by one, until the process terminates with the desired constraint (4.25). Since we also know $y'(b) \leq 1$ (which is a literal axiom), we now have the equality $y' = b$ in (4.18), so that we can add together (4.16) and (4.18).

This concludes our derivation of the pseudo-Boolean encoding (4.3) of the XOR constraint (4.1) from a CNF encoding.

4. For readers knowledgeable in proof complexity, what we are doing here is just the cutting planes simulation of a resolution step resolving the two clauses (4.28a) and (4.28b) over x_k to obtain the clause (4.29). And, jumping ahead a bit, the whole derivation presented here is an adaptation of the standard resolution derivation of contradiction from the 2^k clauses $C_{\neg\rho}$ for all assignments ρ to a set of k variables.

5. A Worked-Out Proof Logging Example

In this section, we present a concrete (toy) application of the methods developed in Section 4, using this example to also illustrate the syntax used in *VeriPB* proof logging files.

Suppose that we have a CNF formula with two parity constraints $x_1 \oplus x_2 \oplus x_3 = 0$ and $x_2 \oplus x_3 \oplus x_4 = 1$, which are encoded as sets of clauses

$$\{\overline{x}_1 \vee x_2 \vee x_3,\ x_1 \vee \overline{x}_2 \vee x_3,\ x_1 \vee x_2 \vee \overline{x}_3,\ \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3\} \tag{5.1a}$$

and

$$\{x_2 \vee x_3 \vee x_4,\ x_2 \vee \overline{x}_3 \vee \overline{x}_4,\ \overline{x}_2 \vee x_3 \vee \overline{x}_4,\ \overline{x}_2 \vee \overline{x}_3 \vee x_4\}\ , \tag{5.1b}$$

respectively. To present this formula to *VeriPB*, we write the constraints in pseudo-Boolean form in an input file as

```
* #variable= 4 #constraint= 8
+1 ~x1 +1  x2 +1  x3 >= 1 ;
+1  x1 +1 ~x2 +1  x3 >= 1 ;
+1  x1 +1  x2 +1 ~x3 >= 1 ;
+1 ~x1 +1 ~x2 +1 ~x3 >= 1 ;
+1  x2 +1  x3 +1  x4 >= 1 ;
+1  x2 +1 ~x3 +1 ~x4 >= 1 ;
+1 ~x2 +1  x3 +1 ~x4 >= 1 ;
+1 ~x2 +1 ~x3 +1  x4 >= 1 ;
```

using the standard OPB file format (Roussel & Manquinho, 2016).[5] In the *proof log* file presented to the *VeriPB* verifier, the start of the file

```
pseudo-Boolean proof version 1.1
f 8
```

instructs the verifier to read this input file, and to expect to see 8 constraints. The verifier maintains a database of pseudo-Boolean constraints, and keeps track of the constraints in the database by assigning to each constraint a *constraint identifier*, which is a positive integer. Upon reading the input file above, the verifier will assign the pseudo-Boolean constraints in the file identifiers 1 through 8 and store them in the constraints database.

When the SAT solver execution starts, the solver reads the formula consisting of all of these clauses from file,[6] and then runs an algorithm to detect clausal encodings of parities. Once the SAT solver detects a parity constraint, it generates a derivation of the pseudo-Boolean encoding of this constraint and writes it to the proof log file. For this translation from CNF to pseudo-Boolean form it is necessary to introduce fresh variables using redundance-based strengthening. For each application of the redundance-based strengthening rule, the proof log will contain a line of the form

5. In fact, *VeriPB* uses a slight extension of the OPB format, which among other things provides greater flexibility in choosing variable names, but since this is not really relevant for this discussion we ignore such details.
6. Although the SAT solver would instead expect its input to be formatted according to the standard DIMACS format used in the SAT competitions. Translating a CNF formula from DIMACS format to OPB format is a simple syntactic operation, and we ignore this detail here.

```
red [constraint C] ; [assignment omega]
```

where `red` identifies the line as a redundance-based strengthening step, followed by the constraint C to be added and the witness substitution ω. The substitution ω is specified by listing each variable in the domain of ω followed by the value or literal it should be substituted by, optionally separated by "`->`".

The translation of the clausal encoding of the parity $x_1 \oplus x_2 \oplus x_3 = 0$ into linear pseudo-Boolean form starts with the reification $y_1 \Leftrightarrow x_1 + x_2 + x_3 \geq 2$ for the fresh variable y_1. The two PB constraints in the reification (which are of the form (3.4a) and (3.4b)) are derived by the two lines

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

in the proof file, which can be checked using Algorithm 1 as shown in Proposition 3.2. After the verifier has succeeded in validating these redundance-based strengthening steps, it adds the new constraints

$$(\text{ID: } 9) \qquad 2\overline{y}_1 + x_1 + x_2 + x_3 \geq 2 \tag{5.2a}$$

$$(\text{ID: } 10) \qquad 2y_1 + \overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 2 \tag{5.2b}$$

to the constraint database (where we note that the constraints are assigned identifiers 9 and 10). In a completely analogous fashion, the reification $y_2 \Leftrightarrow x_1 + x_2 + x_3 - 2y_1 \geq 1$ can be derived by the lines

```
red +3 ~y2 +1  x1 +1  x2 +1  x3 +2 ~y1 >= 3 ; y2 -> 0
red +3  y2 +1 ~x1 +1 ~x2 +1 ~x3 +2  y1 >= 3 ; y2 -> 1
```

in the proof log, which adds the constraints

$$(\text{ID: } 11) \qquad 3\overline{y}_2 + x_1 + x_2 + x_3 + 2\overline{y}_1 \geq 3 \tag{5.3a}$$

$$(\text{ID: } 12) \qquad 3y_2 + \overline{x}_1 + \overline{x}_2 + \overline{x}_3 + 2y_1 \geq 3 \tag{5.3b}$$

to the database of the verifier.

Once these proof logging steps have been performed, the variables y_1 and y_2 correspond to the output carry and sum bit, respectively, of a single full adder with inputs x_1, x_2, x_3. Since the parity is only over three variables in this example, we do not need to derive a chain of multiple full adders as explained in Section 4.4. It is important to note that the SAT solver database will not contain any of these constraints—indeed, the CDCL algorithm does not even know what a "pseudo-Boolean constraint" is—but that the PB constraints only exist in the verifier constraint database for proof logging purposes. However, it is important that the verifier maintains a separate name space for auxiliary variables like y_1 and y_2, so that the SAT solver will not try to use the same variables for any preprocessing or inprocessing steps. If this happens, this will most likely result in an incorrect proof, making the verifier reject.

The next step in our proof logging example is to combine the constraints we just derived with identifiers 9 through 12 via a sequence of cutting planes rule applications. The version

of cutting planes in *VeriPB* is slightly different from (but equivalent to) what we described in Section 2 in that the derivation rules are addition, scalar multiplication, and division.[7] Such operations are written in postfix notation (also known as reverse polish notation) in the following way:

- To use a literal axiom $y \geq 0$ or $\overline{y} \geq 0$, we simply write "`y`" or "`~y`", respectively.
- To add two constraints with identifiers *id1* and *id2*, we write "`id1 id2 +`".
- To multiply a constraint with identifier *id* by a positive integer c, we write "`id c *`".
- To divide a constraint *id* by a positive integer c, we write "`id c d`".

Arbitrary combinations of such derivation steps can be performed using the reverse polish notation rule in *VeriPB*, written on a line in the proof log prefixed by `p` (or `pol`), where the semantics is that any operands (constraint identifiers or factors/divisors) are pushed on a stack, and operators pop the top two elements from this stack and then push back the result of the operation. The final constraint resulting from a sequence of operations is stored with the next available constraint identifier number. In our example, the next lines in the proof log will be

```
p 11  9 2 * + 3 d
p 12 10 2 * + 3 d
```

where the first line starts with constraint number 11 and adds 2 times the constraint 9, after which the result is divided by 3 (and rounded up). The same operations are done in the second line but with the constraints with identifiers 12 and 10. The two lines derive the constraints

$$(\text{ID: } 13) \qquad x_1 + x_2 + x_3 + 2\overline{y}_1 + \overline{y}_2 \geq 3 \tag{5.4a}$$

$$(\text{ID: } 14) \qquad \overline{x}_1 + \overline{x}_2 + \overline{x}_3 + 2y_1 + y_2 \geq 3 \tag{5.4b}$$

encoding an equality of the form (4.16).

The inequalities in (5.4a)–(5.4b) do not yet enforce any parity constraint on the variables x_1, x_2, x_3, since the fresh variables y_1 and y_2 are unconstrained and can be made to satisfy the constraints for any values assigned to x_1, x_2, x_3. To address this, we need to fix the value of y_2, which can be done by generating a brute-force derivation of $\overline{y}_2 \geq 1$ as described in Section 4.4. Since in our example we are dealing only with 3-XORs, i.e., parity constraints over only three variables, we can take a little shortcut when generating the missing clausal constraints of the form (4.27). If we assign the variables x_1, x_2, x_3 so that the parity is even but set $y_2 = 1$, then the constraints (5.4a)–(5.4b) will propagate to contradiction since there is only a single variable y_1 left. This means that we can use reverse unit propagation steps

```
rup +1 ~y2 +1  x1 +1  x2 +1  x3 >= 1 ;
rup +1 ~y2 +1  x1 +1 ~x2 +1 ~x3 >= 1 ;
rup +1 ~y2 +1 ~x1 +1  x2 +1 ~x3 >= 1 ;
rup +1 ~y2 +1 ~x1 +1 ~x2 +1  x3 >= 1 ;
```

7. And there is also an additional saturation rule, just as described in Section 2, but we will not read the saturation rule for this proof logging example.

to derive the clausal constraints that we need (where each `rup`-line claims that adding the negation of the specified constraint as in (2.3) to the current database will cause unit propagation to contradiction, which is checked by the verifier before the constraint is added to the database), and we list below these new constraints 15–18 together with the relevant input constraints

$$(\text{ID: } 15) \qquad \overline{y}_2 + x_1 + x_2 + x_3 \geq 1 \tag{5.5a}$$
$$(\text{ID: } 3) \qquad x_1 + x_2 + \overline{x}_3 \geq 1 \tag{5.5b}$$
$$(\text{ID: } 2) \qquad x_1 + \overline{x}_2 + x_3 \geq 1 \tag{5.5c}$$
$$(\text{ID: } 16) \qquad \overline{y}_2 + x_1 + \overline{x}_2 + \overline{x}_3 \geq 1 \tag{5.5d}$$
$$(\text{ID: } 1) \qquad \overline{x}_1 + x_2 + x_3 \geq 1 \tag{5.5e}$$
$$(\text{ID: } 17) \qquad \overline{y}_2 + \overline{x}_1 + x_2 + \overline{x}_3 \geq 1 \tag{5.5f}$$
$$(\text{ID: } 18) \qquad \overline{y}_2 + \overline{x}_1 + \overline{x}_2 + x_3 \geq 1 \tag{5.5g}$$
$$(\text{ID: } 4) \qquad \overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 1 \tag{5.5h}$$

to get an overview of the clauses involved in the derivation fixing y_2 to false. In the proof log, we can write a single long `p`-line

```
p 15 3 + 2 d 2 16 + 2 d + 2 d 1 17 + 2 d 18 4 + 2 d + 2 d + 2 d
```

to implement the procedure described at the end of Step 2 in Section 4.4. repeating derivations of the clause (4.29) from (4.28a) and (4.28b) for partial assignments over subsets of variables of decreasing size. First, the variable x_3 is eliminated by performing addition followed by division by 2 for the clause pair (5.5a) and (5.5b), the pair (5.5c) and (5.5d), the pair (5.5e) and (5.5f), and the pair (5.5g) and (5.5h), respectively. This yields four new clauses, for which addition followed by division is performed in the same order to eliminate x_2. In the final step, the two clauses $\overline{y}_2 + x_1 \geq 1$ and $\overline{y}_2 + \overline{x}_1 \geq 1$ are added and the result divided by 2 to yield the clause

$$(\text{ID: } 19) \qquad \overline{y}_2 \geq 1 \tag{5.6}$$

as desired. (A further slight optimization could be to only add the clauses together, without any intermediate division steps, and then finally divide by a large enough number—the number of clauses involved in the brute-force derivation will always be enough—but we opted here for keeping all intermediate constraints clausal for simplicity.)

The constraint (5.6) can then be added to (5.4b) to remove y_2. To eliminate y_2 from (5.4a) we can simply use the literal axiom $y_2 \geq 0$, which as mentioned above is referred to as "`y2`" in the `p`-rule. Repeating this in formal notation, the proof lines

```
p 13 y2 +
p 14 19 +
```

derive the inequalities

$$(\text{ID: } 20) \qquad x_1 + x_2 + x_3 + 2\overline{y}_1 \geq 2 \tag{5.7a}$$
$$(\text{ID: } 21) \qquad \overline{x}_1 + \overline{x}_2 + \overline{x}_3 + 2y_1 \geq 3 \tag{5.7b}$$

encoding an equality $x_1 + x_2 + x_3 = 2y_1$ of the form (4.3).

For the second parity constraint $x_2 \oplus x_3 \oplus x_4 = 1$, we perform analogous derivations steps

```
red +2 ~y3 +1  x2 +1  x3 +1  x4 >= 2 ; y3 -> 0
red +2  y3 +1 ~x2 +1 ~x3 +1 ~x4 >= 2 ; y3 -> 1
red +3 ~y4 +1  x2 +1  x3 +1  x4 +2 ~y3 >= 3 ; y4 -> 0
red +3  y4 +1 ~x2 +1 ~x3 +1 ~x4 +2  y3 >= 3 ; y4 -> 1
p 24 22 2 * + 3 d
p 25 23 2 * + 3 d
```

to obtain

$$(\text{ID: } 26) \qquad x_2 + x_3 + x_4 + 2\overline{y}_3 + \overline{y}_4 \geq 3 \tag{5.8a}$$

$$(\text{ID: } 27) \qquad \overline{x}_2 + \overline{x}_3 + \overline{x}_4 + 2y_3 + y_4 \geq 3 \tag{5.8b}$$

after which we fix y_4 to true by writing

```
rup +1 y4 +1  x2 +1  x3 +1 ~x4 >= 1 ;
rup +1 y4 +1  x2 +1 ~x3 +1  x4 >= 1 ;
rup +1 y4 +1 ~x2 +1  x3 +1  x4 >= 1 ;
rup +1 y4 +1 ~x2 +1 ~x3 +1 ~x4 >= 1 ;
p 5 28 + 2 d 29 6 + 2 d + 2 d 30 7 + 2 d 8 31 + 2 d + 2 d + 2 d
```

yielding the constraint

$$(\text{ID: } 32) \qquad y_4 \geq 1 \tag{5.9}$$

on the last line. We finally derive the pseudo-Boolean constraints

$$(\text{ID: } 33) \qquad x_2 + x_3 + x_4 + 2\overline{y}_3 \geq 3 \tag{5.10a}$$

$$(\text{ID: } 34) \qquad \overline{x}_2 + \overline{x}_3 + \overline{x}_4 + 2y_3 \geq 2 \tag{5.10b}$$

encoding the PB equality $x_2 + x_3 + x_4 = 2y_3 + 1$ by the derivation steps

```
p 26  32 +
p 27 ~y4 +
```

and it is straightforward to verify that the constraints (5.10a)–(5.10b) indeed enforce that the parity of the variables x_2, x_3, x_4 is odd. This concludes the proof logging done after detecting parities.

We remark that in the implementation of SAT solving with Gaussian elimination that we made for the purposes of the experiments in this paper, the detection of parities and the proof generation for pseudo-Boolean constraints encoding such parities is done only once at the start of the solver execution. In principle, however, similar detection and derivation steps could also be performed later during the solver search.

Suppose now that that the solver decides on the assignment $x_1 = 0$. Note that adding the two parity constraints $x_1 \oplus x_2 \oplus x_3 = 0$ and $x_2 \oplus x_3 \oplus x_4 = 1$ encoded by our input

formula yields $x_1 \oplus x_4 = 1$, and hence x_4 should propagate to 1. This will be detected when the XOR propagator runs Gaussian elimination.

In order to justify this propagation, in the proof file the solver first needs to derive the new parity constraint by adding pairwise the pseudo-Boolean inequalities encoding the original parity constraints, which is done by inserting the lines

```
p 20 33 +
p 21 34 +
```

producing the new constraints

$$(\text{ID: } 35) \qquad x_1 + x_4 + 2x_2 + 2x_3 + 2\overline{y}_1 + 2\overline{y}_3 \geq 5 \tag{5.11a}$$

$$(\text{ID: } 36) \qquad \overline{x}_1 + \overline{x}_4 + 2\overline{x}_2 + 2\overline{x}_3 + 2y_1 + 2y_3 \geq 5 \tag{5.11b}$$

that imply $x_1 \oplus x_4 = 1$ by the observation made in in Section 4.1.

Once the constraints (5.11a)–(5.11b) have been added to the constraints database of the verifier, the solver also needs to provide a proof that the reason clause $x_1 + x_4 \geq 1$ provided by the XOR propagator is valid. The assignment falsifying this reason clause is $\rho = \{x_1 \mapsto 0, x_4 \mapsto 0\}$. Following the approach in Section 4.3, we derive $x_1 + x_4 \geq 0$ and add to constraint 36 in (5.11b) to get $2\overline{x}_2 + 2\overline{x}_3 + 2y_1 + 2y_3 \geq 3$, after which division by 2 followed by multiplication by 2 yields $2\overline{x}_2 + 2\overline{x}_3 + 2y_1 + 2y_3 \geq 4$. If we add constraint 35 in (5.11a) to this, then the terms $2\overline{x}_2 + 2\overline{x}_3 + 2y_1 + 2y_3$ and $2x_2 + 2x_3 + 2\overline{y}_1 + 2\overline{y}_3$ cancel, leaving a constant 8, and so if we write the line

```
p 36 x1 x4 + + 2 d 2 * 35 +
```

in the proof log, then this yields the clause

$$(\text{ID: } 37) \qquad x_1 + x_4 \geq 1 \tag{5.12}$$

proving that the propagation is valid. Observe that in contrast to the other constraints derived in the proof logging steps above, the reason clause $x_1 \vee x_4$ in (5.12) is also stored in the SAT solver clause database and can can be used in the ensuing CDCL search in the same way as any other clause in this database.

Whenever the XOR propagator detects a propagation or conflict, the solver will need to write derivation steps analogous to the ones leading to constraints (5.11a)–(5.11b) and (5.12) to the proof file. After this, the clause (5.12) can be used either for propagation or as the starting point for CDCL conflict analysis.

6. Implementation and Evaluation

We have extended the pseudo-Boolean proof format (*PBP*) of the *VeriPB* tool[8] with a redundance-based strengthening rule, which the proof checker validates as described in Algorithm 1, and have implemented our proof logging approach for XOR reasoning in a library together with an XOR engine using Gaussian elimination mod 2 to detect XOR

8. *VeriPB* is available at `https://gitlab.com/MIAOresearch/VeriPB`.

propagations.[9] We integrated this library into *MiniSat*[10] to call the XOR propagation method every time clausal propagation terminated. If the library detects a propagation or conflict, a callback is used to notify *MiniSat*, but the reason clause is only generated when needed in conflict analysis. This *lazy reason generation* technique (Soos, Gocht, & Meel, 2020) is crucial to minimize the proof logging overhead, since it avoids generating proofs for reasons that are not used.

In order to be able to compare to approaches using *DRAT*, we have also implemented in our library *DRAT* proof logging for XOR constraints as described by Philipp and Rebola-Pardo (2016). We remark that we did not study the more recent *DRAT*-based approach by Chew and Heule (2020), which combines long parity constraints by sorting the involved literals, because it does not seem to be applicable to the kind of formulas that are relevant for our comparison with *DRAT*. The parity constraints in the formulas we consider will only contain few variables, or else the clausal encoding that we are looking for to detect these parities will blow up the formulas exponentially. Also, when we operate on intermediate parity constraints generated during Gaussian elimination, such parities are guaranteed to be sorted already.

In the results reported below, all running times were measured on an Intel Core i5-1145G7 @2.60GHz × 4 with a memory limit of 8GiB, disk write speed of roughly 200 MiB/s, and read speed of 2 GiB/s. The used tools, benchmarks, data and evaluation scripts are available at `https://doi.org/10.5281/zenodo.7083485`.

Importantly, our goal was not to study whether XOR reasoning is useful or not—this has already been investigated—but to provide efficient proof logging for such reasoning. Therefore, we focused on benchmarks from the SAT competition from 2016 to 2020 that could be solved by *MiniSat* with our XOR propagator but not by *Kissat*,[11] the winner of the 2020 SAT competition. There were 39 such instances, and they could be solved in 0.01 seconds on average by *MiniSat* with the XOR propagator. With our new proof logging the average running time increased to 0.02 seconds and unsatisfiability could be verified in 1.29 seconds on average. For *DRAT* proof logging, on the other hand, the average solving time jumped to 2.72 seconds and verification took 1092 seconds on average.

In order to get systematic measurements for the performance of our new proof logging technique, we ran experiments on the so-called *Tseitin formulas*[12] introduced by Tseitin (1968), including some formula instances that have been studied before in the applied SAT community in the context of proof logging. Tseitin formulas consist of large inconsistent sets of parity constraints, and can thus be viewed as a worst case for XOR reasoning. To the best of our knowledge, the shortest *DRAT* proofs for these formulas obtained so far[13] are based on hand-crafted so-called *propagation redundancy* (PR) proofs, which have been translated to *DRAT* using the tool *PR2DRAT* (Kiesl, Rebola-Pardo, & Heule, 2018). Table 1 shows the disk space required for the proofs of Tseitin formulas by Kiesl et al. (2018). The pseudo-Boolean proofs obtained by *MiniSat* with the XOR propagator are

9. The code for the XOR engine is available at `https://gitlab.com/MIAOresearch/xorengine`.
10. *MiniSat* is available at `http://minisat.se/`.
11. `http://fmv.jku.at/kissat/`.
12. Somewhat confusingly, and as can be seen from the instance names in Table 1, these formulas are sometimes also referred to as *Urquhart formulas* in the applied SAT community, perhaps because Urquhart (1987) was the first to establish strong hardness results for these formulas.
13. The proofs and instances can be found at `https://github.com/marijnheule/drat2er-proofs`.

Instance	*MiniSat* + XOR (*PBP*)	(*DRAT*)	*PR2DRAT*
Urquhart-s5-b1	80.8	3033.1	3878.4
Urquhart-s5-b2	84.0	2844.4	3575.2
Urquhart-s5-b3	123.5	7584.0	7521.0
Urquhart-s5-b4	99.8	5058.6	5271.5

Table 1: Proof sizes (KiB) for some previously studied Tseitin formulas.

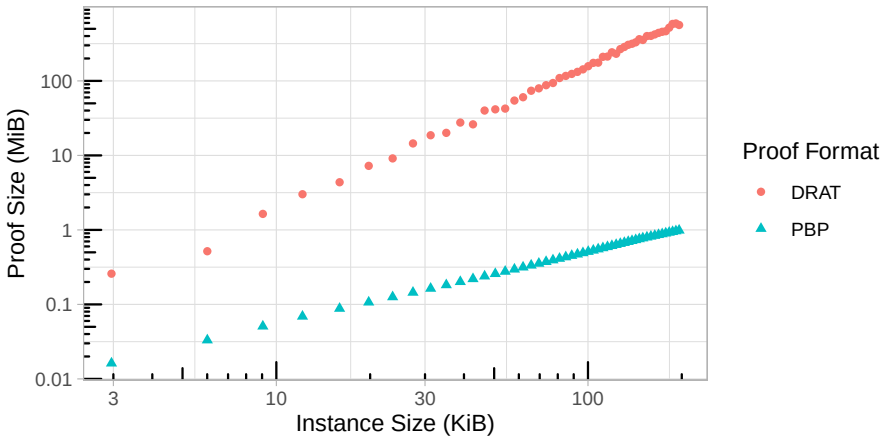


Figure 2: Proof sizes for larger Tseitin formulas using *DRAT* and PB proof logging.

dramatically smaller than the *DRAT* proofs produced by the same tool, and the size of our *DRAT* proofs are similar to that of the best previously known *DRAT* proofs.

The formulas in Table 1 contain only 50 XOR constraints over about 100 variables, which is very small by modern standards, and they are solved and verified in less than one second. To get a sense of the asymptotic behaviour of the proof logging, we also considered 50 new, larger Tseitin formulas with up to 500 XORs over up to 1250 variables. These formulas were generated with the tool *CNFgen* (Lauria et al., 2017) using random regular graphs of degree 5, which produces formulas with clausal encodings of 5-XOR constraints. It was shown by Urquhart (1987) that Tseitin formulas are hard for *resolution*, the reasoning method underlying conflict-driven clause learning, if the graph from which the formula is generated is an expander, and it is well known that random graphs are expanders with extremely high probability (see, e.g., (Hoory, Linial, & Wigderson, 2006)). Thus, we can be confident that the generated formulas are hard for CDCL solvers and require additional reasoning methods, such as Gaussian elimination, to be solved efficiently.

In Figure 2 we compare the proof size for *DRAT* proof logging and our pseudo-Boolean *VeriPB* proof logging. Notice that both proof logging approaches result in straight lines in the log-log plot, which is a strong indication that they are both scaling polynomially. Studying the slopes of the lines yields the estimates that *DRAT* produces quadratic-size proofs while the proof size of the pseudo-Boolean proof is linear in the size of the formula. In Figure 3 we compare the running time (system time plus user time) of solving and

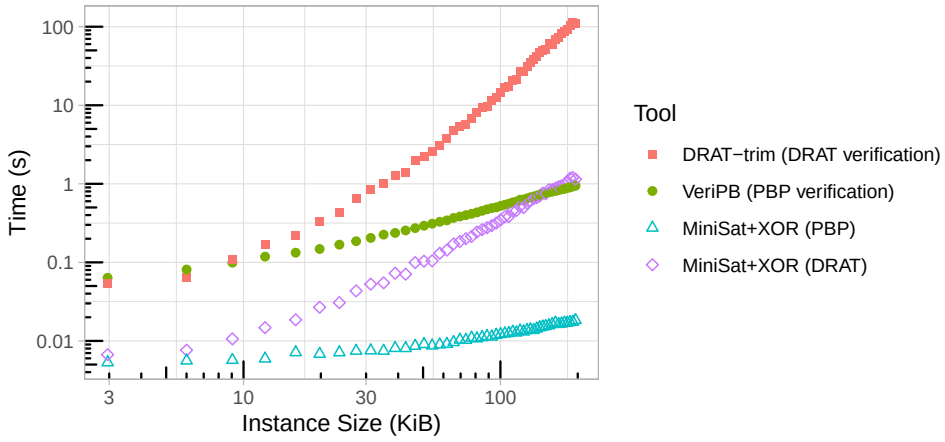


Figure 3: Solving and verification time for Tseitin formulas.

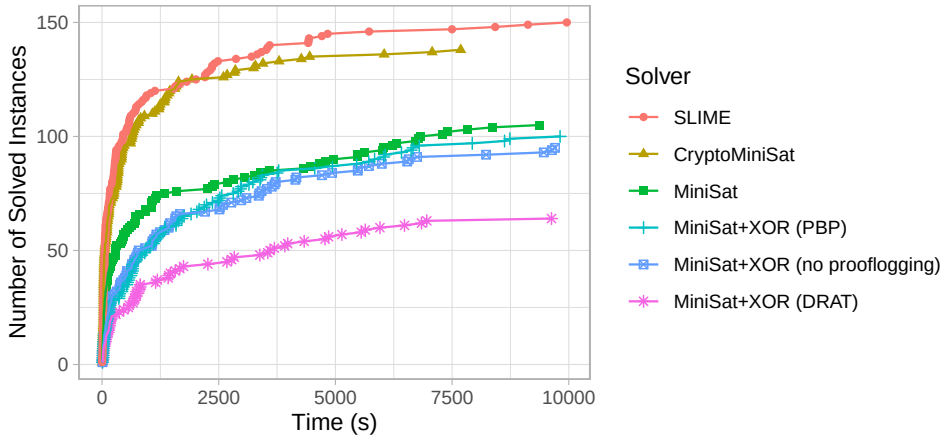


Figure 4: Cumulative plot for the crypto track of the SAT Competition 2021.

producing the proof, as well as time spent on proof verification (where it can be noted that running times below one second should be interpreted with some care since the running time might be dominated by start-up overhead). It is clear that the larger proof size required for *DRAT* proofs does not only increase verification time, but also causes a clearly increased time overhead during solving.

To get a wider range of practically relevant formulas, we additionally evaluated our tools on cryptographic benchmarks, which often contain parity constraints, from the crypto track of the 2021 SAT competition. Figure 4 compares the performance of different solvers on this benchmark set, including *SLIME*,[14] the winner of the crypto track, and *CryptoMiniSat*,[15] arguably the most well-established modern solver with integrated parity reasoning. Notably, *SLIME* and *CryptoMiniSat* significantly outperform *MiniSat*, showing the advancements made over the last decades. Somewhat surprisingly, our integration of parity reasoning

14. `https://maxtuno.github.io/slime-sat-solver/`.
15. `https://github.com/msoos/cryptominisat/`.

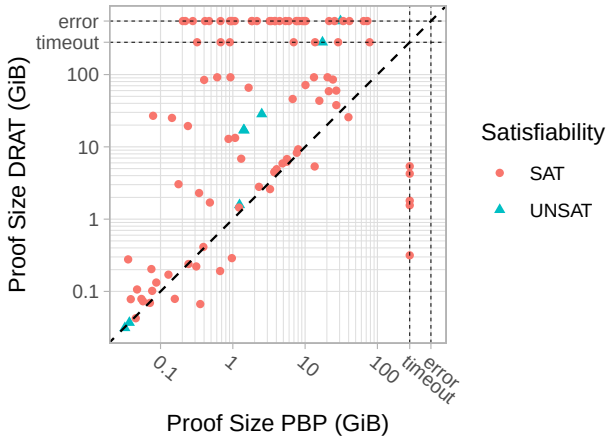


Figure 5: *DRAT* and PB and proof sizes for the crypto track of the SAT Competition 2021.

does not seem to benefit *MiniSat* on this set of benchmarks. However, if one insists on that the solver with parity reasoning should also support proof logging, then it is clear that more instances can be solved if we use pseudo-Boolean proof logging instead of *DRAT*. One reason for this is that the proof sizes are much larger when using DRAT proof logging, as shown in Figure 5. The generated *DRAT* proofs can quickly exceed the disk limit of roughly 100GB, causing the SAT solver to terminate with an error.

While the tendency of the plot in Figure 5 is clear, it should be noted that the difference shown is due not only to different proof logging methods, but also to the particular way in which we implemented proof logging for *MiniSat*, in which the introduction of new variables for proof logging affects the *MiniSat* search. This can be observed in different statistics such as the number of decisions or conflicts.[16] For example, consider the instance in the bottom right of Figure 5 that requires a proof of a few hundred MiB in *DRAT* but times out for pseudo-Boolean proof logging. This instance is solved with $541,928$ conflicts with no proof logging or *DRAT* proof logging, but requires more than 19 million conflicts for pseudo-Boolean proof logging. It should be emphasized, however, that this difference is completely irrelevant in that it is not in any way related to pseudo-Boolean proof logging per se, but only to a peculiar choice in the implementation we used for our experiments, as explained in the footnote above.

In Figure 6 we can see the time required for solving a benchmark versus verifying the result. In practice, it would be sufficient to only verify the final solution for satisfiable

16. In principle, the CDCL proof search should be completely oblivious to whether proof logging is being carried out or not, since no proof logging steps have any bearing on how the search algorithm is executed. However, in our implementation we use the variable handling interface in *MiniSat* to manage the auxiliary variables introduced during proof logging. In more technical detail, the proof logging routines introduces fresh variables by adding them to the solver and marking them as non-decision variables. The mere existence of these additional variables seems to cause a slight change in the search. The difference can only be observed when variables were added before preprocessing. With hindsight, it would most likely be better to let the proof logging code manage additional variables only used for the pseudo-Boolean derivations separately from the solver, However, our ambition was not to deliver a production-grade SAT solver with Gaussian elimination, but to provide a competitive implementation that can serve as a basis for meaningful experiments.

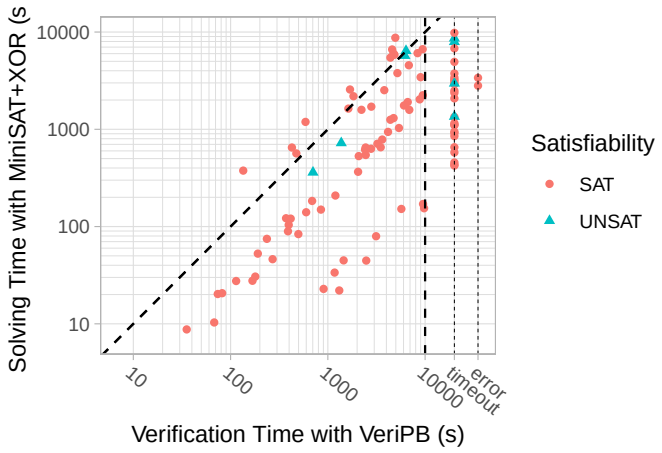


Figure 6: Time required for solving and verifying instances.

instances. However, as there are few solved unsatisfiable instances, we verified that every constraint derived by the solver is correct even for satisfiable ones. For most formulas the verification overhead is roughly a factor 10, but there are also cases where verification is much slower, which demonstrates that further optimizations in the *VeriPB* proof verification code would be desirable. We are aware of several possibilities for this, such as improving RUP checks and borrowing ideas like backward trimming of proofs from *DRAT-trim*.[17] However, the main focus of this work was not on such engineering questions, but rather on developing new mathematical methods for efficient proof logging, and we would argue that the potential for vast improvements in proof logging efficiency should be clear from the experimental results reported in this section.

7. Conclusion

In this work, we present an efficient proof logging method for conflict-driven clause learning (CDCL) solvers equipped with parity reasoning, which has been a long-standing challenge in SAT solving. Our approach circumvents the prohibitive overhead of previous *DRAT*-based proof logging methods for parity reasoning such as the one developed by Philipp and Rebola-Pardo (2016) by instead using the cutting planes method operating on pseudo-Boolean inequalities in the *VeriPB* tool and adding a rule for introducing extension variables. An experimental evaluation shows that this makes the proof logging overhead, the size of the proof, and the time required for verification all go down by an order of magnitude or more compared to *DRAT*. While there is certainly ample room for further improvements, our first proof-of-concept implementation already shows the power of this approach.

In terms of weaknesses, one significant disadvantage of our method is that the proof verification time is still considerably larger than the time required for solving with proof logging, especially if many XOR constraints are involved. There are at least two explanations for this. One reason is that the algorithm for XOR reasoning can make use of bit-level parallelism. The verifier cannot do so easily, because it has to be able to deal with arbitrary

17. `https://github.com/marijnheule/drat-trim`.

linear constraints and not just XORs. Another reason is that we introduce fresh variables to encode the XOR constraints. On the solver side, these auxiliary variables can essentially be ignored, except that they are printed in fairly standardized proof logging templates, but they play a crucial role in the calculations on the proof checker side when the proof is verified. It should be said, though, that although verification overhead is larger than proof logging overhead, this is only by a constant factor. In other words, if we are willing to pay a constant-factor increase in running time, then this will allow us to not only use parity reasoning but also obtain a formal proof establishing that this parity reasoning has been performed correctly. It seems fair to argue that the benefits from fully verified solutions could outweigh the disadvantage of this limited increase in total execution time.

By construction, the pseudo-Boolean proof logging method in *VeriPB* can also be used to solve another task that has remained very challenging for *DRAT*, namely efficient proof logging for cardinality detection and reasoning. We have not investigated this in the current paper, since this is mostly an engineering question rather than a research problem in view of the methods that have already been developed by Biere, Le Berre, Lonca, and Manthey (2014), Elffers and Nordström (2020). Symmetry handling, a third notorious problem for proof logging, appears to be much more difficult, but when it comes to adding symmetry breaking constraints our method can do at least as well as Heule et al. (2015), since it is a strict generalization of *DRAT*. In a later work (Bogaerts et al., 2022) appearing after the conference version of this paper, the *VeriPB* proof logging system has been extended further with a so-called *dominance-based strengthening*, providing for the first time efficient proof logging support for fully general symmetry breaking. It is an interesting open question, however, whether this new dominance rule is necessary, or whether the redundance-based strengthening rule introduced in the current work is sufficient to provide efficient derivations of symmetry-breaking constraints.

The fact that no efficient proof logging support has previously been available for enhanced SAT solving techniques such as parity reasoning, cardinality detection, and symmetry handling means that SAT solvers making crucial use of such techniques have not been able to take part in the main track of the SAT competition, where proof logging is mandatory. Somewhat paradoxically, this seems to have the effect that the proof logging requirements, which have played such an important role for the development of the field, now risk becoming a barrier to further solver developments. Since the *VeriPB* tool can now support all of parity reasoning, cardinality detection, and (as of (Bogaerts et al., 2022)) also symmetry breaking, and does so with very limited overhead compared to *DRAT*, it seems natural to propose that this should be an allowed proof logging format in future SAT competitions.

However, we believe that the potential benefit of pseudo-Boolean proof logging with extension variables goes well beyond the context of the SAT competitions. *VeriPB* has been shown to be capable of efficient justification of important constraint programming techniques (Elffers et al., 2020; Gocht et al., 2022), and can also provide proof logging for a wide range of graph problem solvers (Gocht et al., 2020b, 2020a). Furthermore, Gocht et al. (2022), Vandesande et al. (2022) have used *VeriPB* to develop proof logging methods that seem to have the potential to support a range of SAT-based optimization approaches using maximum satisfiability (MaxSAT) solvers. The pseudo-Boolean rules for reasoning with 0-1 linear constraints provide a simple yet very expressive formalism, and it

does not seem out of the question to hope that they could be extended to deal with proof logging for mixed integer programming (MIP). Thus, we believe that the ultimate goal of this line of research should be to design a unified proof logging approach for as wide as possible a range of combinatorial optimization paradigms. In addition to furnishing efficient machine-verifiable proofs of correctness, proof logging could also serve as a valuable tool for debugging and empirical performance analysis during solver development. Furthermore, the proofs produced could in principle provide auditability by third parties using independently developed software, and/or be a stepping stone towards explainability by showing, e.g., why certain solutions are optimal. We view our paper as only one of the first steps on this long but exciting road.

Acknowledgments

We are grateful to Bart Bogaerts and Ciaran McCreesh for many stimulating conversations on proof logging in general and *VeriPB* in particular. We also want to thank Kuldeep Meel and Mate Soos for helpful discussions on how to implement Gaussian elimination modulo 2. A special thanks goes to Andy Oertel for helping us track down mistakes in our worked-out example in Section 5. Finally, we have benefited greatly from the interactions with and feedback from many colleagues taking part in the semester program *Satisfiability: Theory, Practice, and Beyond* in the spring of 2021 at the Simons Institute for the Theory of Computing at UC Berkeley.

The authors were supported by the Swedish Research Council grant 2016-00782, and Jakob Nordström also received funding from the Independent Research Fund Denmark grant 9040-00389B.

References

Akgün, Ö., Gent, I. P., Jefferson, C., Miguel, I., & Nightingale, P. (2018). Metamorphic testing of constraint solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, Vol. 11008 of *Lecture Notes in Computer Science*, pp. 727–736. Springer.

Alekhnovich, M., Ben-Sasson, E., Razborov, A. A., & Wigderson, A. (2002). Space complexity in propositional calculus. *SIAM Journal on Computing*, *31*(4), 1184–1211. Preliminary version in *STOC '00*.

Allouche, D., André, I., Barbe, S., Davies, J., Givry, S. d., Katsirelos, G., O'Sullivan, B., Prestwich, S., Schiex, T., & Traoré, S. (2014). Computational protein design as an optimization problem. *Artificial Intelligence*, *212*(1), 59–79.

Baek, S., Carneiro, M., & Heule, M. J. H. (2021). 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)*, Vol. 12651 of *Lecture Notes in Computer Science*, pp. 59–75. Springer.

Bayardo Jr., R. J., & Schrag, R. (1997). Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97)*, pp. 203–208.

Biere, A. (2006). Tracecheck. `http://fmv.jku.at/tracecheck/`.

Biere, A., Le Berre, D., Lonca, E., & Manthey, N. (2014). Detecting cardinality constraints in CNF. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, Vol. 8561 of *Lecture Notes in Computer Science*, pp. 285–301. Springer.

Biró, P., van de Klundert, J., Manlove, D. F., Pettersson, W., Andersson, T., Burnapp, L., Chromy, P., Delgado, P., Dworczak, P., Haase, B., Hemke, A., Johnson, R., Klimentova, X., Kuypers, D., Costa, A. N., Smeulders, B., Spieksma, F. C. R., Valentín, M. O., & Viana, A. (2021). Modelling and optimisation in European kidney exchange programmes. *European Journal of Operational Research*, *291*(2), 447–456.

Bogaerts, B., Gocht, S., McCreesh, C., & Nordström, J. (2022). Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, pp. 3698–3707.

Bryant, R. E. (2022). TBUDDY: a proof-generating BDD package. EasyChair preprint 8471. Available at `https://easychair.org/publications/preprint/DbRN`.

Bryant, R. E., Biere, A., & Heule, M. J. H. (2022). 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)*, Vol. 13243 of *Lecture Notes in Computer Science*, pp. 443–461. Springer.

Buss, S. R., & Nordström, J. (2021). Proof complexity and SAT solving. In Biere, A., Heule, M. J. H., van Maaren, H., & Walsh, T. (Eds.), *Handbook of Satisfiability* (2nd edition)., Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, chap. 7, pp. 233–350. IOS Press.

Buss, S. R., & Thapen, N. (2019). DRAT proofs, propagation redundancy, and extended resolution. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, Vol. 11628 of *Lecture Notes in Computer Science*, pp. 71–89. Springer.

Chai, D., & Kuehlmann, A. (2005). A fast pseudo-Boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *24*(3), 305–317. Preliminary version in *DAC '03*.

Chew, L., & Heule, M. J. H. (2020). Sorting parity encodings by reusing variables. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, Vol. 12178 of *Lecture Notes in Computer Science*, pp. 1–10. Springer.

Clegg, M., Edmonds, J., & Impagliazzo, R. (1996). Using the Groebner basis algorithm to find proofs of unsatisfiability. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC '96)*, pp. 174–183.

Cook, W., Coullard, C. R., & Turán, G. (1987). On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, *18*(1), 25–38.

Cook, W., Koch, T., Steffy, D. E., & Wolter, K. (2013). A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, *5*(3), 305–344.

Cruz-Filipe, L., Heule, M. J. H., Hunt, W. A., Kaufmann, M., & Schneider-Kamp, P. (2017a). Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, Vol. 10395 of *Lecture Notes in Computer Science*, pp. 220–236. Springer.

Cruz-Filipe, L., Marques-Silva, J., & Schneider-Kamp, P. (2017b). 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)*, Vol. 10205 of *Lecture Notes in Computer Science*, pp. 118–135. Springer.

Delorme, M., García, S., Gondzioa, J., Kalcsics, J., Manlove, D., & Pettersson, W. (2019). Mathematical models for stable matching problems with ties and incomplete lists. *European Journal of Operational Research*, *277*(2), 426–441.

Dixon, H. E., Ginsberg, M. L., & Parkes, A. J. (2004). Generalizing Boolean satisfiability I: Background and survey of existing work. *Journal of Artificial Intelligence Research*, *21*, 193–243.

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

Elffers, J., Gocht, S., McCreesh, C., & Nordström, J. (2020). Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pp. 1486–1494.

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

Elffers, J., & Nordström, J. (2020). A cardinal improvement to pseudo-Boolean solving. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pp. 1495–1503.

Gange, G., & Stuckey, P. (2019). Certifying optimality in constraint programming. Presentation at KTH Royal Institute of Technology. Slides available at `https://www.kth.se/polopoly_fs/1.879851.1550484700!/CertifiedCP.pdf`.

Gillard, X., Schaus, P., & Deville, Y. (2019). SolverCheck: Declarative testing of constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, Vol. 11802 of *Lecture Notes in Computer Science*, pp. 565–582. Springer.

Gocht, S., Martins, R., Nordström, J., & Oertel, A. (2022). Certified CNF translations for pseudo-Boolean solving. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, Vol. 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 16:1–16:25.

Gocht, S., McBride, R., McCreesh, C., Nordström, J., Prosser, P., & Trimble, J. (2020a). 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)*, Vol. 12333 of *Lecture Notes in Computer Science*, pp. 338–357. Springer.

Gocht, S., McCreesh, C., & Nordström, J. (2020b). Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pp. 1134–1140.

Gocht, S., McCreesh, C., & Nordström, J. (2022). An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, Vol. 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 25:1–25:18.

Gocht, S., & Nordström, J. (2021). Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pp. 3768–3777.

Gocht, S., Nordström, J., & Yehudayoff, A. (2019). On division versus saturation in pseudo-Boolean solving. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI '19)*, pp. 1711–1718.

Goldberg, E., & Novikov, Y. (2003). Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pp. 886–891.

Han, C., & Jiang, J. R. (2012). When Boolean satisfiability meets Gaussian elimination in a simplex way. In *Proceedings of the 24th International Conference on Computer Aided Verification, (CAV '12)*, Vol. 7358 of *Lecture Notes in Computer Science*, pp. 410–426. Springer.

Heule, M. J. H., Hunt Jr., W. A., & Wetzler, N. (2013a). Trimming while checking clausal proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pp. 181–188.

Heule, M. J. H., Hunt Jr., W. A., & Wetzler, N. (2013b). Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, Vol. 7898 of *Lecture Notes in Computer Science*, pp. 345–359. Springer.

Heule, M. J. H., Hunt Jr., W. A., & Wetzler, N. (2015). Expressing symmetry breaking in DRAT proofs. In *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*, Vol. 9195 of *Lecture Notes in Computer Science*, pp. 591–606. Springer.

Heule, M. J. H., Kiesl, B., & Biere, A. (2017). Short proofs without new variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, Vol. 10395 of *Lecture Notes in Computer Science*, pp. 130–147. Springer.

Hoory, S., Linial, N., & Wigderson, A. (2006). Expander graphs and their applications. *Bulletin of the American Mathematical Society*, *43*(4), 439–561.

Kiesl, B., Rebola-Pardo, A., & Heule, M. J. H. (2018). Extended resolution simulates DRAT. In *Proceedings of the 9th International Joint Conference on Automated Reasoning (IJCAR '18)*, Vol. 10900 of *Lecture Notes in Computer Science*, pp. 516–531. Springer.

Laitinen, T., Junttila, T., & Niemelä, I. (2012a). Conflict-driven XOR-clause learning. In *Proceedings of the 15th International Conference on Theory and Applications of*

Satisfiability Testing (SAT '12), Vol. 7317 of *Lecture Notes in Computer Science*, pp. 383–396. Springer.

Laitinen, T., Junttila, T., & Niemelä, I. (2012b). Extending clause learning SAT solvers with complete parity reasoning. In *Proceedings of the IEEE 24th International Conference on Tools with Artificial Intelligence (ICTAI '12)*, pp. 65–72.

Lauria, M., Elffers, J., Nordström, J., & Vinyals, M. (2017). CNFgen: A generator of crafted benchmarks. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, Vol. 10491 of *Lecture Notes in Computer Science*, pp. 464–473. Springer.

Le Berre, D., & Parrain, A. (2010). The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, *7*, 59–64.

Leyton-Brown, K., Milgrom, P., & Segal, I. (2017). Economics and computer science of a radio spectrum reallocation. *Proceedings of the National Academy of Sciences*, *114*(28), 7202–7209.

Manlove, D. F. (2016). Hospitals/residents problem. In Kao, M.-Y. (Ed.), *Encyclopedia of Algorithms*, pp. 926–930. Springer New York.

Manlove, D. F., McBride, I., & Trimble, J. (2017). "Almost-stable" matchings in the hospitals / residents problem with couples. *Constraints*, *22*(1), 50–72.

Manlove, D. F., & O'Malley, G. (2012). Paired and altruistic kidney donation in the UK: Algorithms and experimentation. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA '12)*, Vol. 7276 of *Lecture Notes in Computer Science*, pp. 271–282. Springer.

Mann, M., Will, S., & Backofen, R. (2008). CPSP-tools – Exact and complete algorithms for high-throughput 3D lattice protein studies. *BMC Bioinformatics*, *9*, 230:1–230:8.

Marques-Silva, J. P., & Sakallah, K. A. (1999). GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, *48*(5), 506–521. Preliminary version in *ICCAD '96*.

McConnell, R. M., Mehlhorn, K., Näher, S., & Schweitzer, P. (2011). Certifying algorithms. *Computer Science Review*, *5*(2), 119–161.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 530–535.

Pardalos, P. M., Du, D.-Z., & Graham, R. L. (Eds.). (2013). *Handbook of Combinatorial Optimization* (2nd edition). Springer.

Philipp, T., & Rebola-Pardo, A. (2016). DRAT proofs for XOR reasoning. In *Proceedings of the 15th European Conference on Logics in Artificial Intelligence (JELIA '16)*, Vol. 10021 of *Lecture Notes in Computer Science*, pp. 415–429. Springer.

Roussel, O., & Manquinho, V. M. (2016). 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`.

Sinz, C., & Biere, A. (2006). Extended resolution proofs for conjoining BDDs. In *Proceedings of the 1st International Computer Science Symposium in Russia (CSR '06)*, Vol. 3967 of *Lecture Notes in Computer Science*, pp. 600–611. Springer.

Soos, M., & Bryant, R. E. (2022). Combining CDCL, Gauss-Jordan elimination, and proof generation. EasyChair preprint 8497. Available at `https://easychair.org/publications/preprint/4rGK`.

Soos, M., Gocht, S., & Meel, K. S. (2020). Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV '20)*, Vol. 12224 of *Lecture Notes in Computer Science*, pp. 463–484. Springer.

Soos, M., Nohl, K., & Castelluccia, C. (2009). Extending SAT solvers to cryptographic problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, Vol. 5584 of *Lecture Notes in Computer Science*, pp. 244–257. Springer.

Tseitin, G. (1968). On the complexity of derivation in propositional calculus. In Silenko, A. O. (Ed.), *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pp. 115–125. Consultants Bureau, New York-London.

Urquhart, A. (1987). Hard examples for resolution. *Journal of the ACM*, *34*(1), 209–219.

Vandesande, D., Wulf, W. D., & Bogaerts, B. (2022). QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, Vol. 13416 of *Lecture Notes in Computer Science*, pp. 429–442. Springer.

Veksler, M., & Strichman, O. (2010). A proof-producing CSP solver. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '10)*, pp. 204–209.

Wetzler, N., Heule, M. J. H., & Hunt Jr., W. A. (2014). 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)*, Vol. 8561 of *Lecture Notes in Computer Science*, pp. 422–429. Springer.

Yang, J., & Meel, K. S. (2021). Engineering an efficient PB-XOR solver. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP '21)*, Vol. 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 58:1–58:20.