Documentation of VeriPB and CakePB for the SAT Competition 2023

Bart Bogaerts Ciaran McCreesh Magnus O. Myreen Jakob Nordström
Andy Oertel Yong Kiam Tan

March 1, 2023

Abstract

This is the documentation for the pseudo-Boolean proof checker VeriPB together with its formally verified backend CakePB as proposed for usage in the SAT competition 2023. If any questions arise regarding corner cases not covered by this documentation, or regarding how to use pseudo-Boolean proof logging to certify correctness of different forms of reasoning, any inquiries are are welcome and may be directed to `jn@di.ku.dk`.

Contents

1 Introduction

The pseudo-Boolean proof format used for the proof checker VERIPB [Ver] supports proof logging for decision, enumeration, and optimization problems, as well as problem reformulations, all in a unified format. So far, VERIPB has been used for proof logging of enhanced SAT solving techniques [GN21, BGMN22], pseudo-Boolean CDCL-based solving [GMNO22], constraint programming [EGMN20, GMN22], subgraph solving [GMN20, GMM+20], and MaxSAT solving [VDB22], and this list of applications is expected to keep growing. In this paper we present a recently revised version of the proof format, focusing on how it can be used to certify unsatisfiability of CNF formulas in the SAT competition 2023.

The full proof format makes it possible to specify *different types of proofs*, where an important consideration is that it might only be known towards the end of the proof what kind of proof was produced (e.g., if a preprocessor did not just reformulate a problem but actually solved it). It also supports *composition of proofs*, so that different solvers can collaborate simultaneously or in sequence on solving a problem. However, in this document we focus on the restricted version of the format that is proposed to be supported in the SAT competition 2023.

2 Quickstart Guide for Boolean Satisfiability (SAT) Proof Logging

This section contains the bare minimum of information needed to use VERIPB and CAKEPB as proof checkers for Boolean satisfiability (SAT) solvers with pseudo-Boolean proof logging. A good way to learn more (in addition to reading this document) might be to study the example files in the directory `tests/integration_tests/correct/` in the repository [Ver] and run VERIPB with the options `--trace --useColor`, which will output detailed information about the proofs and the proof checking.

2.1 Running the Proof Checkers

If a SAT solver with pseudo-Boolean proof logging has solved the instance `input.cnf`, the generated proof `input.pbp` can be checked by VERIPB and CAKEPB by runnning the following commands:

```
# Translate to kernel format proof
veripb --cnf --proofOutput translated.pbp input.cnf input.pbp
# Check the kernel proof
cake_pb_cnf input.cnf translated.pbp
```

The first command recompiles the pseudo-Boolean proof `input.pbp` into a more restricted "kernel-format" proof `translated.pbp` using VERIPB, after which the kernel proof is checked using CAKEPB. In case of successful recompilation, VERIPB will output:

```
# Running veripb as shown above
...
Verification succeeded
```

Upon successful proof checking, CAKEPB will report success on the standard output stream:

```
# Running cake_pb_cnf as shown above
s VERIFIED UNSAT
```

All errors are reported on standard error.

2.2 Proof Format

The syntactic format of a pseudo-Boolean proof of unsatisfiability for a CNF formula as expected by the version of VeriPB proposed for the SAT competition 2023 is

```
pseudo-Boolean proof version 2.0
f ⟨N⟩
```
Derivation section
```
output NONE
conclusion UNSAT : ⟨id⟩
end pseudo-Boolean proof
```

where $\langle N \rangle$ should be the number of clauses in the formula and *Derivation section* should contain the actual proof which derives contradiction as the pseudo-Boolean constraint with constraint ID $\langle id \rangle$.

In pseudo-Boolean format, a disjunctive clause like

$$x_1 \vee \overline{x}_2 \vee x_3 \tag{2.1a}$$

is represented as the inequality

$$x_1 + \overline{x}_2 + x_3 \geq 1 \tag{2.1b}$$

claiming that at least one of the literals in the clause is true (i.e., takes value 1), and this inequality is written as

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

in the OPB format [RM16] used by VeriPB. The proof checker can also read CNF formulas in the DIMACS and WDIMACS formulas used for SAT solving and MaxSAT solving, respectively. For such files, VeriPB will parse a clause

```
1 -2 3 0
```

to be identical to (2.1b), and the variables should be referred to in the pseudo-Boolean proof file as `x1`, `x2`, `x3`, et cetera.

DRAT proofs can be transformed into valid VeriPB proofs by simple syntactic manipulations. Most of the proof resulting from a CDCL SAT solver is the ordered sequence of clauses learnedduring conflict analysis. Since all such clauses are guaranteed to be *reverse unit propagation (RUP)* clauses, in pseudo-Boolean proofs such learned clauses can be derived most easily by writing lines like

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

(assuming that the learned clause was (2.1a)) in the derivation section of the pseudo-Boolean proof (as explained in more detail in Section 5.2.2).

If instead the clause (2.1a) is a *resolution asymmetric tautology (RAT)* clause that is RAT on x_1, then this is written as

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

in the pseudo-Boolean proof using the more general *redundance-based strengthening* rule (discussed in Section 5.4.1). And if the RAT literal would instead have been $\overline{x}_2$, this would have been indicated by ending the proof line above by `x2 -> 0` instead.

Finally, in order to delete the clause (2.1a), the deletion command

```
del spec +1 x1 +1 ~x2 +1 x3 >= 1 ;
```

is issued. (This and other deletion rules are covered in Section 5.6.) An important difference from DRAT proofs is that deletion is made also for unit clauses, i.e., clauses containing only a single literal—DRAT proof checkers typically ignore such deletion commands. Another crucial difference is that all clauses learned during CDCL execution need to be written down in the proof log, including unit clauses. If unit clauses are missing in a DRAT proof, the proof checkers will typically be helpful and silently infer and add the missing clauses. No such patching of formally incorrect proofs is offered by VERIPB.

It should be noted, though, that if all the reasoning performed by some particular SAT solver can efficiently be captured by standard DRAT proof logging, then there is no real reason to use pseudo-Boolean proof logging for that solver. Pseudo-Boolean proof logging becomes relevant if the solver uses more advanced techniques such as, for instance, cardinality reasoning, Gaussian elimination, or symmetry breaking. We refer the reader to [GN21] and [BGMN22], respectively, for detailed descriptions of how to do efficient pseudo-Boolean proof logging for the latter two techniques.

3 Pseudo-Boolean and Proof Checker Preliminaries

In this section, we briefly review some pseudo-Boolean preliminaries and general principles for how pseudo-Boolean proof checking works.

3.1 Pseudo-Boolean Notation and Terminology

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). A *pseudo-Boolean (PB) inequality* is a 0–1 linear inequality

$$C \doteq \sum_i a_i \ell_i \geq A\,, \tag{3.1}$$

where a_i and A are integers (and where we write $\doteq$ to denote syntactic equality). We can assume without loss of generality that pseudo-Boolean constraints are *normalized*; i.e., that all literals ℓ_i are over distinct variables and that the *coefficients* a_i and the *degree (of falsity)* A are non-negative. This is how constraints are represented internally in the proof checker, but most of the time there is no need to worry about this, and the proof checker accepts constraints written in non-normalized form.

A *pseudo-Boolean formula* is a conjunction $F \doteq \bigwedge_j C_j$ of pseudo-Boolean inequalities, which we can also think of as the set $\bigcup_j \{C_j\}$ of inequality constraints in the formula. Since a *(disjunctive) clause* $\ell_1 \vee \cdots \vee \ell_k$ is equivalent to the pseudo-Boolean constraint $\ell_1 + \cdots + \ell_k \geq 1$, formulas in *conjunctive normal form (CNF)* can be viewed as special cases of pseudo-Boolean formulas.

To introduce some further convenient notation, we write equality $\sum_i a_i \ell_i = A$ as syntactic sugar for the pair of pseudo-Boolean inequalities $\sum_i a_i \ell_i \geq A$ and $\sum_i -a_i \ell_i \geq -A$. The *negation* $\neg C$ of the constraint C in (3.1) can be represented as the pseudo-Boolean inequality

$$\neg C \doteq \sum_i -a_i \ell_i \geq -A + 1\,, \tag{3.2}$$

and the fact that the set of pseudo-Boolean inequalities is closed under negation is quite convenient for proof logging purposes. If z is a Boolean variable and $\sum_i a_i \ell_i \geq A$ is a pseudo-Boolean inequality in normalized form with $\sum_i a_i = M$, then we write

$$z \Rightarrow \sum_i a_i \ell_i \geq A \;\doteq\; A \cdot \overline{z} + \sum_i a_i \ell_i \geq A \tag{3.3a}$$

to denote the *right reification* and

$$z \Leftarrow \sum_i a_i \ell_i \geq A \;\doteq\; (M - A + 1) \cdot z + \sum_i a_i \overline{\ell}_i \geq M - A + 1 \tag{3.3b}$$

for the *left reification* of $\sum_i a_i \ell_i \geq A$. As the notation suggests, the constraints (3.3a)–(3.3b) enforce that z is true if and only if $\sum_i a_i \ell_i \geq A$ holds.

ρ	$slack(C;\rho)$	Remark
$\{\}$	8	
$\{\overline{x}_5\}$	3	C propagates $\overline{x}_4$ (coefficient $>$ slack)
$\{\overline{x}_5, \overline{x}_4\}$	3	Propagation does not change slack
$\{\overline{x}_5, \overline{x}_4, \overline{x}_3, x_2\}$	-2	Conflict (slack is negative)

Figure 1: Example slack calculations for the constraint $C \doteq x_1 + 2\overline{x}_2 + 3x_3 + 4\overline{x}_4 + 5x_5 \geq 7$.

If for an optimization problem with objective function $\sum_i w_i\ell_i$ to be minimized the solver finds a solution α, then we refer to

$$\sum_i w_i\ell_i \leq -1 + \sum_i w_i \cdot \alpha(\ell_i) \tag{3.4}$$

as a *objective-improving constraint* enforcing solutions yielding a strictly better value of the objective function during the rest of the search.

If ρ is a (possibly partial) assignment that is (provably) sufficient to uniquely specify a solution to an enumeration problem, then we refer to

$$\sum_{\ell_i\,:\,\rho(\ell_i)=0} \ell_i \geq 1 \tag{3.5}$$

as a *solution-excluding constraint* enforcing that different solutions will be found in the rest of the search.

3.2 Assignments, Substitution, Slack, and Unit Propagation

A *(partial) assignment* ρ is a (partial) function from variables to $\{0,1\}$; a *substitution* ω can also map variables to literals. These are extended from variables to literals in the natural way by respecting the meaning of negation. We also identify a partial assignment ρ with the set of literals set to true by ρ, so that $\ell \in \rho$ if and only if $\rho(\ell) = 1$. We can write $x \mapsto b$ when $\rho(x) = b$, for b a literal or truth value.

We write $\rho \circ \omega$ to denote the composed substitution resulting from applying first ω and then ρ, i.e., $\rho \circ \omega(x) = \rho(\omega(x))$. As an example, for $\omega = \{x_1 \mapsto 0, x_3 \mapsto \overline{x}_4, x_4 \mapsto x_3\}$ and $\rho = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 0, x_4 \mapsto 0\}$ we have $\rho \circ \omega = \{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 1, x_4 \mapsto 0\}$. Applying ω to a pseudo-Boolean inequality C as in (3.1) yields

$$C{\restriction_\omega} \doteq \sum_i a_i\omega(\ell_i) \geq A\,, \tag{3.6}$$

substituting literals or values as specified by ω. For a formula F we define $F{\restriction_\omega} \doteq \bigwedge_j C_j{\restriction_\omega}$.

The (normalized) pseudo-Boolean inequality C in (3.1) is *satisfied* by ρ if $\sum_{\ell_i \in \rho} a_i \geq A$. A pseudo-Boolean formula F is satisfied by ρ if all constraints in it are, in which case it is *satisfiable*. If there is no satisfying assignment, F is *unsatisfiable*. Two formulas are *equisatisfiable* if they are both satisfiable or both unsatisfiable. We also consider optimisation problems, where in addition to F we are given an integer linear objective function $f \doteq \sum_i w_i\ell_i$ and the task is to find an assignment that satisfies F and minimizes f. (To deal with maximization problems we can just negate the objective function.) 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 also satisfies C, and write $F \models F'$ if $F \models C'$ for all $C' \in F'$.

The *slack* of a normalized pseudo-Boolean inequality C as in (3.1) with respect to ρ measures how far ρ is from falsifying C, and is defined formally as

$$slack\big(\textstyle\sum_i a_i\ell_i \geq A; \rho\big) = \sum_{\overline{\ell}_i \notin \rho} a_i - A\,. \tag{3.7}$$

The constraint C is *conflicting* under ρ if $slack(C;\rho) < 0$. If ρ does not assign ℓ_j but $0 \leq slack(C;\rho) < a_j$, then C *propagates* ℓ_j under ρ, meaning that if the literal ℓ_j is falsified, then the constraint will become conflicting. See Figure 1 for some example calculations of slack. Note that constraint can be conflicting though not all variables in it have been assigned.

During *unit propagation* on F under ρ, the assignment ρ is extended iteratively by any propagated literals until an assignment ρ' is reached under which no constraint $C \in F$ is propagating, or under which

some constraint C is conflicting as described above. We refer to this latter scenario as a *conflict*, and say that ρ' *violates* the constraint C in this case. We say that F implies C by *reverse unit propagation (RUP)*, and that C is a *RUP constraint* with respect to F, if $F \wedge \neg C$ unit propagates to conflict under the empty assignment. Just as is the clause for clausal proof logging systems, the concept of RUP constraints will be important in pseudo-Boolean proof logging.

3.3 Syntax for Pseudo-Boolean Inequalities in Proofs

VERIPB expects pseudo-Boolean inequalities to be written in the OPB format [RM16], for which some examples are provided in Section 2.1. The proof checker also supports an extension to the OPB format that allows to use arbitrary variable names instead of only `x1`, `x2`, `x3`, ... Variable names in this extended format should:

- start with a letter in `A-Z` or `a-z`;
- continue with characters in `A-Z`, `a-z`, `0-9`, or `[]{}-_^` (square and curly brackets, hyphen, underscore, and caret);
- contain at least two characters.

Variable names cannot contain spaces. The proof checker has a reserved namespace of variable names starting with _ (underscore) for internal variables. Support for additional characters in variable names may be added but is implementation-specific. Unsupported characters may generate a proof checker error message, but can never generate erroneous verification results.

3.4 General Principles for the Proof Checker

At any step in a proof the proof checker maintains a (multi-)set $\mathcal{C}$ of *core constraints* and a (multi-)set $\mathcal{D}$ of *derived constraints*, where all constraints are pseudo-Boolean inequalities. At the outset, $\mathcal{C}$ contains the constraints in the input formula. All derived constraints are added to $\mathcal{D}$, but constraints can be moved from $\mathcal{D}$ to $\mathcal{C}$. New constraints are derived from $\mathcal{C} \cup \mathcal{D}$ using the *cutting planes* proof system as described more formally in [BN21] together with the *redundance-based strengthening* and *dominance-based strengthening* rules discussed in [BGMN22, GN21]. The syntactic form of the derivation rules and their exact semantic meaning is explained in Section 5.

A general design principle behind VERIPB is that the core set $\mathcal{C}$ should be equivalent to the input formula when it comes to satisfiability (for a decision problem) or optimal value of objective function (for an optimization problem), and therefore a constraints C should only be deleted from $\mathcal{C}$ if it can be proven that C can be recovered from $\mathcal{C} \setminus \{C\}$. A deletion from the core set that violates this is referred to as an *unchecked deletion*. SAT solvers that only use proofs to establish unsatisfiability can perform such unchecked deletions.

4 Overall Proof Structure

Let us now describe what different types of VERIPB proofs there are, although for the SAT competition only proofs for decision problems are of interest. The most general version of the pseudo-Boolean proof format also allows to compose proofs, but this is not supported for the proof checker in the SAT competition and so we will not discuss it further here.

4.1 Proof Sections

A single, atomic VERIPB proof consists of three sections:

1. A **derivation** section, where the input constraints are specified and derivations are performed on these constraints using the cutting planes and strengthening rules, and where solutions can also be logged.

2. An **output** section, where the constraints currently in the core set $\mathcal{C}$ of the constraint database can be specified (for verification/debugging purposes or as input to the next stage of the solving process).

3. A **conclusions** section, which specifies what if anything was established by the derivation in terms of satisfiability/unsatisfiability, optimality, or listing of solutions.

The syntactic format of the proof is

`pseudo-Boolean proof version 2.0`
`f` $\langle N \rangle$
Derivation section
`output` *output type*
Output section (optional)
`conclusion` *conclusion section (single line)*
`end pseudo-Boolean proof`

(where the reason for the final line `end pseudo-Boolean proof` is to make it very easy for experiment scripts do decide when parsing log files whether proof logging terminated successfully or not). Let us now discuss the different sections in more detail.

4.2 Derivation Section

The proof starts with the proof header and the "load formula" `f` command, after which the derivations from the input follow in the derivation section. The precise syntax is:

`pseudo-Boolean proof version 2.0`
`f` $\langle N \rangle$
Derivation section

where the parameter $\langle N \rangle$ is the number of pseudo-Boolean inequalities in the input formula. The derivation consists of the different cutting planes and strengthening rules as well as solution logging rules explained in the rest of this document.

4.3 Output Section

The output section, if non-empty, should be is a listing of the (potentially multi-set of) core constraints, where every core constraint should be listed according to its multiplicity, preceded by a description of what is guaranteed for this list of constraints:

1. `DERIVABLE` : The listed core set is derivable. This can be used to implement *finalization* as in [BCH21] by requiring that all constraints in the core set should be listed exactly once and that the derived set should be empty (which forces the solver to prove that it knows exactly what it has derived).

2. `EQUI-SATISFIABLE` : The listed core set is satisfiable if and only if the input problem at the start of this (atomic) proof is satisfiable. This option might only be relevant for decision problems, and it requires that the proof uses no unchecked deletion steps.[1]

3. `EQUI-OPTIMAL` : The optimal solution for the listed core set with respect to the objective function has the same value as the optimal solution for the input problem at the start of the proof, except if the derivation section of the proof has happened to log an optimal solution, in which case the core set is guaranteed to be unsatisfiable. This option is relevant for optimization problems, and requires that the proof uses no unchecked deletion steps.

[1] Note that standard SAT solvers should not necessarily be expected to follow this—they only use the proof to show unsatisfiability, whereas for satisfiable instance the proof log is ignored and instead the proposed solution output directly by the SAT solver is checked.

4. `EQUI-SOLVABLE` : There is a bijection between solutions to the input problem and solutions to the listed core constraints, except for any solutions found and logged during the course of the proof.

In terms of syntax, the output section has the format

`output` *output type*
Output section (optional)

where

```
output NONE
```

indicates that the proof has no output section (and this is the only supported option for the SAT competition).

If there is output, then *output type* should be a pair of parameters *guarantee* and *list type*, where *guarantee* is one of `DERIVABLE` , `EQUI-SATISFIABLE` , `EQUI-OPTIMAL` , or `EQUI-SOLVABLE` , and *list type* is one of `CONSTRAINTS` , `PERMUTATION` , or `IMPLICIT` . The output section should list any meta-information needed to connect the input formula and the output formula, such as how the input formula objective has been rewritten to the output formula objective, in case the objective has changed. After this, the output formula should be listed as a valid OPB file (possibly in the extended OPB format that VERIPB uses).

In slightly more detail, the output section format for an optimization problem should look like:

`output EQUI-OPTIMAL CONSTRAINTS`
$\langle Obj_{\text{input}} \rangle$ =$\langle Obj_{\text{output}} \rangle$
`*` `#variable=`⟨*number of variables*⟩ `#constraint=`⟨*number of constraints*⟩
`min:` $\langle Obj_{\text{output}} \rangle$

followed by an ordered sequence of pseudo-Boolean constraints listing every constraint in the core set $\mathcal{C}$ exactly once. Above, $\langle Obj_{\text{input}} \rangle$ and $\langle Obj_{\text{output}} \rangle$ are the original and reformulated objectives, respectively.

Regarding the ordered list of pseudo-Boolean constraints in the output formula, for large formulas it is desirable to allow a variant

`output EQUI-OPTIMAL PERMUTATION`
$\langle Obj_{\text{input}} \rangle$ =$\langle Obj_{\text{output}} \rangle$
`*` `#variable=`⟨*number of variables*⟩ `#constraint=`⟨*number of constraints*⟩
`min:` $\langle Obj_{\text{output}} \rangle$

which is followed by the constraint IDs in the core set in the desired order (with every ID of a constraint in the core set listed exactly once), as well as an even more concise variant

`output EQUI-OPTIMAL IMPLICIT`
$\langle Obj_{\text{input}} \rangle$ =$\langle Obj_{\text{output}} \rangle$
`*` `#variable=`⟨*number of variables*⟩ `#constraint=`⟨*number of constraints*⟩
`min:` $\langle Obj_{\text{output}} \rangle$

which is not followed by anything, and instead just implicitly assumes that we want the constraints in the core set listed in increasing order of constraint IDs.

The output from a preprocessor for a decision problem, where the problem instance has been reformulated while maintaining equi-satisfiability, should instead look like e.g.,

`output EQUI-SATISFIABLE CONSTRAINTS`
`*` `#variable=`⟨*number of variables*⟩ `#constraint=`⟨*number of constraints*⟩

followed by the list of constraints resulting from the preprocessing phase.

4.4 Conclusions Section

Proof logging is supported for the following types of problems and conclusions:

Decision problems: For decision problems without objective function, possible solver conclusions are:

1. `SAT` : The problem instance is satisfiable.

2. `UNSAT` : Problem instance is unsatisfiable/infeasible.
3. `NONE` : Result unknown.

Optimization problems: For problems with an objective function, possible solver conclusions are:

1. `BOUNDS` : The optimal solution to the problem instance lies in a specified interval $[LB, UB]$. If $LB = UB$, then optimality has been proven. We could also have $UB = \infty$, in which case no solution has been found, and/or LB being equal to the constant term in the (normalized) objective function, in which case no nontrivial lower bound has been proven.
2. `UNSAT` : Problem instance is unsatisfiable/infeasible.
3. `NONE` : Result unknown—in particular, no solution or lower bound has been found.

Enumeration problems: For enumeration problems (which should not have objective functions), possible solver conclusions are:

1. `UNSAT` : Problem instance is unsatisfiable/infeasible.
2. `ENUMERATION PARTIAL` : The proof has logged one or more solutions, but there is no derivation of contradiction ruling out the existence of further solutions.
3. `ENUMERATION COMPLETE` : The proof has logged one or more solutions and has also derived contradiction, ruling out the existence of further solutions.
4. `NONE` : Result unknown (in particular, no solution has been found).

The conclusion section has the format

`conclusion` *conclusion line*
`end pseudo-Boolean proof`

and after the keyword `conclusion` the proof should state what has been established. If there are no conclusions (as for, e.g., a pure reformulation of a problem), then

`conclusion NONE`

signals this. In the syntax descriptions below, square brackets are used to denote optional parts of the syntax.

For an unsatisfiability proof for a decision problem, the conclusion section should be

`conclusion UNSAT [:` $\langle id \rangle$ `]`

where $\langle id \rangle$ is an optional reference to a constraint ID with negative slack. Note that such a constraint ID is never needed for verification purposes—if indeed there is such a constraint, then $0 \geq 1$ is a reverse unit propagation (RUP) constraint, which can easily be checked—but it can be helpful for debugging purposes or to be able to reconstruct the actual proof found by a solver. If a solution to a decision problem is found, the conclusion section should be

`conclusion SAT [:` *assignment*`]`

where *assignment* is a list of variables with the values they are mapped to, but this is not supported for the SAT competition, since the proof log is not used to check the result if the instance is satisfiable.

None of the other conclusion types below are supported for the SAT competition, but are listed for completeness.

For an optimization problem, the syntactic format of the conclusion section is

`conclusion BOUNDS` $\langle LB \rangle$ $\langle UB \rangle$ `[:` $\langle id1 \rangle$ $\langle id2 \rangle$ `]`

for optional constraint IDs $\langle id1 \rangle$ and $\langle id2 \rangle$. The constraint ID $\langle id1 \rangle$ should refer to a constraint from which $Obj_{\text{input}} \geq LB$ follows by so-called syntactic implication (note that this constraint could be contradiction

$0 \geq 1$).[2] The constraint ID $\langle id2 \rangle$ should be the objective-improving constraint corresponding to when a solution with value UB was logged.

For an enumeration problem, the conclusion section should look like

`conclusion ENUMERATION PARTIAL` $\langle N \rangle$

for a partial enumeration of N solutions and

`conclusion ENUMERATION COMPLETE` $\langle N \rangle$ `[:` $\langle id \rangle$ `]`

for a complete enumeration of N solutions (with an optional reference id to a constraint with negative slack showing that the enumeration is complete).

5 Derivation Rules

In this section we describe the different rules or commands that can appear in a VERIPB proof. Every rule in a proof file has to be written on a single line, and no single line may contain more than one rule. Different rules can create different numbers of new constraints (or none).

The VERIPB proof checker accepts proofs in what we will refer to as an "augmented" proof format, including a rich collection of rules intended to make proof logging as convenient as possible. The verified proof checker CAKEPB only supports a "kernel" subset of these commands, where the intended workflow for formally verified proof checking is to use VERIPB as a preprocessor to compile augmented proofs into the kernel format to be formally checked by CAKEPB. In this section, we focus on describing the general augmented proof format, but we also mention the restrictions in the kernel format. A summary of the kernel format and the formally verified proof checker is presented in Section 6.

The expected setting in a VERIPB proof is that there is an input formula F in (linear) pseudo-Boolean form (and note that CNF formulas would be a special case of this). For a *decision problem*, the proof should establish whether F has satisfying assignments or is unsatisfiable. For an *optimization problem* the goal is to minimize a 0–1 integer linear objective function f subject to the constraints in F, or at least to prove as tightly matching upper and lower bounds on the objective function as possible (and decision problems can be viewed as a special cases of optimization problems by considering the objective function $f \doteq 0$ to be trivial). For an *enumeration problem* the solver should provide an (ideally exhaustive) list of solutions to F.

At all times, the VERIPB proof checker maintains a current state with the following information:

- The input formula F (which is immutable once loaded as described in Section 5.1).
- The objective function f (which would be the trivial function 0 for a decision problem).
- A constraint database consisting of a (multi-)set $\mathcal{C}$ of *core constraints* and a (multi-)set $\mathcal{D}$ of *derived constraints*, where all constraints are pseudo-Boolean inequalities. Each constraint is identified by a unique positive integer referred to as a *constraint ID*.
- A counter of the largest integer maxId used for any constraint ID in the proof so far.
- The best objective function value v achieved for any solution found so far (which is ∞ if no solution has been found).
- A (possibly empty) set of pseudo-Boolean inequalities $\mathcal{O}_{\preceq}(\vec{u}, \vec{v})$ over ordered sets of variables $\vec{u}$ and $\vec{v}$, which encodes a preorder, i.e., a reflexive and transitive relation.
- A sequence of literals $\vec{z}$ (normally, but not necessarily, distinct) on which the preorder $\mathcal{O}_{\preceq}$ should be applied.

[2] **Work in progress:** If no constraint ID is specified, VERIPB should loop over all constraints currently in the core set and check for each constraint whether it syntactically implies $Obj_{\text{input}} \geq LB$.

- The set of all variables that has appeared so far in the proof.
- The current proof level currLevel, offered as a convenience to help backtracking solvers keep track of which constraints should be erased when.

When a constraint is expected by some proof command, as a general rule a positive integer N is interpreted as referring to the constraint in the database with constraint ID N. Also, a negative integer $-N$ is interpreted in the augmented format as the constraint with ID maxId $+ 1 - N$, i.e., the Nth most recent constraint derived, but this convention is not supported in the kernel format.

5.1 Manipulation of Constraints Database

Load formula The command

`f` $\langle N \rangle$

loads the input formula (as specified in the invocation of the proof checker) and stores all constraints in the core set $\mathcal{C}$. The input formula is expected to contain N pseudo-Boolean inequalities, which will get constraint IDs $1, 2, \ldots, N$.

Note that any pseudo-Boolean equality $\sum_i a_i \ell_i = A$ in the input formula will be counted as two constraints $\sum_i a_i \ell_i \geq A$ and $\sum_i a_i \ell_i \leq A$ in this order. As an example, if the OPB file

```
* #variable= 4 #constraint= 2
1 x1 2 x2 >= 1 ;
1 x3 1 x4  = 1 ;
```

is loaded in a pseudo-Boolean proof file starting

```
pseudo-Boolean proof version 2.0
f 3
```

then the formula stored in the proof checker core set is

```
1:  1 x1  2 x2 >=  1 ;
2:  1 x3  1 x4 >=  1 ;
3: -1 x3 -1 x4 >= -1 ;
```

(with the constraint ID listed at the start of each line just for purposes of illustration).

Move to core Constraints can be moved from the derived set to the core set with the commands

`core id` $\langle id1 \rangle$ $\langle id2 \rangle$ $\langle id3 \rangle$ `...`
`core range` $\langle idStart \rangle$ $\langle idEnd \rangle$

where the first command variant specifies a list of one or more constraint IDs and the second variant specifies a range between $\langle idStart \rangle$ and $\langle idEnd \rangle$. All constraint IDs must be valid. Moving a constraint to the core that is already in the core has no effect. In the kernel proof format only the `core id` command is supported.

5.2 Implicational Rules

Implicational rules derive new pseudo-Boolean inequalities C that are guaranteed to be semantically implied by the constraint database $\mathcal{C} \cup \mathcal{D}$. We write $\mathcal{C} \cup \mathcal{D} \vdash C$ when a derivation of C from $\mathcal{C} \cup \mathcal{D}$ using the implicational derivation rules below can be exhibited.

5.2.1 Implicational Rules in Kernel Format: Reverse Polish Notation

Reverse polish notation The reverse polish notation rule

`pol` $\langle$*sequence of operations in reverse polish notation*$\rangle$

derives a new pseudo-Boolean inequality that receives constraint ID maxid $+ 1$ (after which maxid is incremented). The derivation is specified by a sequence of operations in the cutting planes proof system (as described in [BN21]). The sequence is given in reverse polish notation. That is, all operands in the sequence are pushed on a stack, and operations are performed on the top one or two elements (where $\langle op1 \rangle$ is below the top element $\langle op2 \rangle$ of the stack), after which the result is pushed on the stack:

- $\langle op1 \rangle$ $\langle op2 \rangle$ `+` adds two constraints.
- $\langle op1 \rangle$ $\langle op2 \rangle$ `*` multiplies the constraint $\langle op1 \rangle$ by the positive integer $\langle op2 \rangle$.
- $\langle op1 \rangle$ $\langle op2 \rangle$ `d` divides the constraint $\langle op1 \rangle$ by the positive integer $\langle op2 \rangle$ (with integer rounding).
- $\langle op \rangle$ `s` saturates the constraint $\langle op \rangle$.
- $\langle op1 \rangle$ $\langle op2 \rangle$ `w` weakens the constraint $\langle op1 \rangle$ by removing the variable $\langle op2 \rangle$ (assuming that it contributes so as to satisfy the constraint), where $\langle op2 \rangle$ should be a variable name without negation.

At the end of a reverse polish notation line the stack should contain a single constraint, which is the result of the `pol` rule application. If the stack is instead empty or contains more than one element, then this is an error. It is possible to write just `p` instead of `pol`.

As is the case in general for derivation rules, when a constraint is expected by some arithmetic operation a positive integer is interpreted as an absolute constraint ID and a negative integer $-N$ as referring to the constraint with ID maxid $+ 1 - N$. If instead a literal `var` or $\sim$`var` is encountered where a constraint is expected, then `var` is interpreted as the literal axiom constraint `var` ≥ 0 and $\sim$`var` is interpreted as $\sim$`var` ≥ 0 (or, equivalently, `var` ≤ 1). Note that since variables must consist of at least two characters they cannot be confused with arithmetic operations.

5.2.2 Shorthand Implicational Rules in Augmented Format

The implicational derivation rules listed below are provided in the augmented proof format for convenience, but are not supported in the kernel format used by the formally verified proof checker.

Reverse unit propagation (RUP) The rule

`rup` ⟨*pseudo-Boolean inequality C in OPB format*⟩

adds the specified pseudo-Boolean inequality C if it is implied by the constraint database $\mathcal{C} \cup \mathcal{D}$ by reverse unit propagation. That is, the proof checker temporarily adds $\neg C$ to the constraint database and performs unit propagation to check that this leads to conflict.

Syntactic implication The rule

`ia` ⟨*id*⟩ `:` ⟨*pseudo-Boolean inequality C in OPB format*⟩

adds the specified pseudo-Boolean inequality C if there is a single constraint in the database that syntactically implies C (as discussed in [BGMN22]). The argument $\langle id \rangle$ specifies the constraint ID of this syntactically implying constraint.[3]

5.3 Orders

A set of pseudo-Boolean inequalities $\mathcal{O}_{\preceq}$ encoding a preorder is introduced by using the following template, in which square brackets denote optional parts of the notation:

```
pre_order ⟨order name⟩
    vars
        left  ⟨ordered set of left variables⟩
        right ⟨ordered set of right variables (of same size)⟩
```

[3] **Work in progress:** The intention is to make the constraint ID argument optional in future revisions, but this has not yet been implemented. However, for efficiency reasons it will always be best to specify this argument explicitly.

```
      [ aux     ⟨list of auxiliary variables that are currently ignored⟩  ]
  end
  def
      ⟨list of pseudo-Boolean inequalities O_≤ defining order⟩
  end
  transitivity
      vars
          fresh_right  ⟨ordered set of extra variables (of same size)⟩
      end
      proof
          proofgoal  #1
              ⟨proof of transitivity of relation⟩
          end  ⟨id⟩
      end
  end
end
```

Let us describe in more detail the different parts of the definition of a preorder $\mathcal{O}_{\preceq}(\vec{u}, \vec{v})$ over sets of variables $\vec{u}$ and $\vec{v}$. The heading

```
pre_order  ⟨order name⟩
```

introduces the preorder and gives it a unique name (with the same naming conventions as those for variables discussed in Section 3.3), after which

```
  vars
      left   ⟨ordered set of left variables⟩
      right  ⟨ordered set of right variables (of same size)⟩
    [ aux    ⟨list of auxiliary variables that are currently ignored⟩  ]
  end
```

specifies the two sets of variables $\vec{u}$ and $\vec{v}$ used in $\mathcal{O}_{\preceq}$. The list of auxiliary variables is optional and is anyway ignored,[4] except that an empty `aux` line is required in the kernel format. Next, the section

```
  def
      ⟨list of pseudo-Boolean inequalities O_≤ defining order⟩
  end
```

presents the pseudo-Boolean formula claimed to encode a preorder defined by $\alpha \preceq \beta$ if and only if $\mathcal{O}_{\preceq}(\alpha, \beta)$ evaluates to true (except that the proof checker postulates $\alpha \preceq \alpha$ to hold by definition, regardless of $\mathcal{O}_{\preceq}$). The definition of the preorder is concluded by a *subproof*

```
  transitivity
      vars
          fresh_right  ⟨ordered set of extra variables (of same size)⟩
      end
      proof
          proofgoal  #1
              ⟨proof of transitivity of relation⟩
          end  ⟨id⟩
      end
  end
```

establishing that $\mathcal{O}_{\preceq}$ defines a transitive relation, i.e., that the implication $\mathcal{O}_{\preceq}(\vec{u}, \vec{v}) \wedge \mathcal{O}_{\preceq}(\vec{v}, \vec{w}) \vDash \mathcal{O}_{\preceq}(\vec{u}, \vec{w})$ provably holds.

As an example, lexicographic order over 3 bits defined by $(u_1, u_2, u_3) \preceq (v_1, v_2, v_3)$ when $\vec{u}$ viewed as a binary number is less than or equal to $\vec{v}$ can be expressed by the pseudo-Boolean inequality

$$-4u_1 + 4v_1 - 2u_2 + 2v_2 - u_3 + v_3 \geq 0 \tag{5.1}$$

[4] **Work in progress:** This syntax is to allow for the introduction of auxiliary variables in future revisions of the proof format in a backwards-compatible way.

and can be introduced as a preorder `ternarylex` as follows:

```
pre_order ternarylex
    vars
        left  u1 u2 u3
        right v1 v2 v3
        aux
    end
    def
        -4 u1 4 v1 -2 u2 2 v2 -1 u3 1 v3 >= 0 ;
    end
    transitivity
        vars
            fresh_right w1 w2 w3
        end
        proof
            proofgoal #1
                pol 1 2 + 3 +
            end -1
        end
    end
end
```

We will discuss subproofs in more detail in Section 5.5 below, but remark here that in the kernel format there must be explicit proofs for each proof goal in the transitivity proof. Also, the kernel format requires that the `aux` variable list is empty.

The `load_order` command loads the named order (which must previously have been successfully defined by a `pre_order` command). The command

```
load_order ternarylex x1 x2 x3
```

loads our example order `ternarylex` and specifies that it will be applied to the variables x_1, x_2, x_3. Calling `load_order` in this way in the middle of a proof also has the effect of moving all constraints currently in the derived set $\mathcal{D}$ to the core set $\mathcal{C}$.

The `load_order` command can also be called with no arguments, in which case the currently loaded order is unloaded. Derived constraints are not moved to the core for such an empty order command. We denote the empty preorder by $\mathcal{O}_\top$.

5.4 Strengthening Rules

Most proof logging steps for a solver trying to minimize f subject to the constraints in the the pseudo-Boolean formula F (or trying to solve the decision problem F, in which case we recall that we can consider the objective function $f \doteq 0$ to be trivial), are expected to be performed using the implicational rules in Section 5.2. However, we also need to allow *strengthening rules* deriving constraints C that are not semantically implied by the input formula. Adding such constraints C is in order as long as some optimal solution is maintained, i.e., a satisfying assignment to F that minimizes f. This idea was formalized in [BGMN22] by allowing the use of an additional pseudo-Boolean formula $\mathcal{O}_\preceq(\vec{u}, \vec{v})$ that, together with a sequence of variables $\vec{z}$, defines a relation on the set of truth value assignments.

Since we will need to make somewhat advanced use of substitutions below, let us discuss some notational conventions from [BGMN22]. If F is a pseudo-Boolean formula over variables $\vec{x} = \{x_1, \ldots, x_m\}$, we can write $F(\vec{x})$ to make explicit the set of variables $\vec{x}$ over which F is defined. For a substitution ω with domain (contained in) $\vec{x}$, the notation $F\big(\vec{x}\restriction_\omega\big)$ is understood to be a synonym of $F\restriction_\omega$. For the same formula F and $\vec{y} = \{y_1, \ldots, y_m\}$, the notation $F(\vec{y})$ is syntactic sugar for $F\restriction_\omega$ with ω denoting the substitution (implicitly) defined by $\omega(x_i) = y_i$ for $i = 1, \ldots, n$. Finally, for a formula $G = G(\vec{x}, \vec{y})$ over $\vec{x} \cup \vec{y}$ and substitutions α and β defined on $\vec{z} = \{z_1, \ldots, z_n\}$ (either of which could be the identity function), we write $G(\vec{z}\restriction_\alpha, \vec{z}\restriction_\beta)$ to denote $G\restriction_\omega$ for ω defined by $\omega(x_i) = \alpha(z_i)$ and $\omega(y_i) = \beta(z_i)$ for $i = 1, \ldots, n$.

Using this notation, we let $\mathcal{O}_\preceq$ define a relation $\alpha \preceq \beta$ that holds when $\mathcal{O}_\preceq(\vec{z}\restriction_\alpha, \vec{z}\restriction_\beta)$ evaluates to true, or when $\vec{z}\restriction_\alpha \doteq \vec{z}\restriction_\beta$, as also discussed in Section 5.3. Then $\preceq$ can be combined with the objective function f to define a preorder $\preceq_f$ on assignments by

$$\alpha \preceq_f \beta \quad \text{if} \quad \alpha \preceq \beta \text{ and } f\restriction_\alpha \leq f\restriction_\beta\,, \tag{5.2}$$

and we require that all strengthening rules should preserve some solution that is minimal with respect to this preorder $\preceq_f$.

5.4.1 Redundance-Based Strengthening

The format of the *redundance-based strengthening* rule is

`red` ⟨*pseudo-Boolean inequality C in OPB format*⟩ `;` ⟨*substitution ω*⟩ `[; begin`
⟨*subproofs*⟩
`end [` ⟨*id*⟩ `]]`

where we are again denoting optional parts of the rule with square brackets. This rule makes it possible to derive a constraint C from $\mathcal{C} \cup \mathcal{D}$ even if C is not implied, provided that the proof logger establishes that any assignment α that satisfies $\mathcal{C} \cup \mathcal{D}$ can be transformed into another assignment $\alpha' \preceq_f \alpha$ that satisfies both $\mathcal{C} \cup \mathcal{D}$ and C. (In case the order is empty, which we write as $\mathcal{O}_\preceq = \mathcal{O}_\top$, then the condition $\alpha' \preceq_f \alpha$ just means that the inequality $f\restriction_{\alpha'} \leq f\restriction_\alpha$ should hold—note that this is vacuously true for a decision problem). We remark that this rule is a generalized version of the RAT rule in [HHW13]. The redundance-based strengthening rule in the form we are using it here originated in [GN21], which in turn relies heavily on [HKB17, BT19].

More formally, if v is the best value for the objective function achieved by any solution so far (or ∞ if no solution has been found), then C can be derived by redundance-based strengthening, or just *redundance* for brevity, if there is a substitution ω (referred to as the *witness*) such that an explicit derivation

$$\mathcal{C} \cup \mathcal{D} \cup \{f \leq v - 1\} \cup \{\neg C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup C)\restriction_\omega \cup \{f\restriction_\omega \leq f\} \cup \mathcal{O}_\preceq(\vec{z}\restriction_\omega, \vec{z})\,. \tag{5.3}$$

can be provided. Intuitively, (5.3) says that if some assignment α satisfies $\mathcal{C} \cup \mathcal{D}$ but falsifies C, then the assignment $\alpha' = \alpha \circ \omega$ still satisfies $\mathcal{C} \cup \mathcal{D}$ and also satisfies C. In addition, the condition $f\restriction_\omega \leq f$ ensures that $\alpha \circ \omega$ achieves an objective function value that is at least as good as that for α.

In a redundance rule application, the witness ω is presented as a space-separated list

`var1 -> val1  var2 -> val 2  var3 -> val3 ...`

of variables in the domain of ω and what they are mapped to (i.e., truth values or literals). The arrow symbols `->` are optional in the augmented format, but not in the kernel format.

The ⟨*subproofs*⟩ part contains a derivation of every constraint on the right-hand side of (5.3), except that for, e.g., RUP constraints and syntactically implied constraints the proof checker can be asked to fill in the proof. How much automatic proof generation the proof checker will provide depends on whether the augmented or the kernel proof format is used, with less generous support provided in the kernel format. We will discuss subproofs in more detail in Section 5.5, but note here that the constraints on the right-hand of (5.3) for which subproofs are needed, and which are referred to as *proof goals*, are referred by labels constructed as follows:

1. For $(\mathcal{C} \cup \mathcal{D})\restriction_\omega$, each proof goal $D\restriction_\omega$ is labelled by the constraint ID of D in the database $\mathcal{C} \cup \mathcal{D}$.
2. The remaining constraints have special labels with a distinguishing prefix "#" as follows:
   (a) Label #1 refers to the proof goal $C\restriction_\omega$ for the constraint C being derived by the redundance rule application.
   (b) Labels #2, #3, …, #$N+1$ refer to proof goals for the order $\mathcal{O}_\preceq$ with $N = |\mathcal{O}_\preceq|$, i.e., there is one goal per constraint in the order (or $N = 0$ if no order is loaded).
   (c) Label #$N+2$ refers to the proof goal for the constraint $f\restriction_\omega \leq f$ saying that the objective function must not get worse, if the problem contains an objective function f.

5.4.2 Dominance-Based Strengthening

The *dominance-based strengthening* rule has the format

`dom` ⟨*pseudo-Boolean inequality C in OPB format*⟩ `;` ⟨*substitution ω*⟩ `[; begin`
⟨*subproofs*⟩
`end [` ⟨*id*⟩ `]]`

(with optional parts within square brackets), and to explain how it works we first make a quick detour to discuss order relations. For any preorder $\preceq$, we can define a strict order $\prec$ by postulating that $\alpha \prec \beta$ if $\alpha \preceq \beta$ and $\beta \not\preceq \alpha$ (which means that, as a special case, the empty preorder yields a "strict order" that does not relate any elements at all). The relation $\prec_f$ obtained in this way from the preorder (5.2) coincides with what is called a *dominance relation* in [CS15] in the context of constraint optimization, which is the reason for why the dominance-based strengthening rule has received its name.

Just as for the redundance rule, the dominance rule allows to derive a constraint C from $\mathcal{C} \cup \mathcal{D}$ even if C is not implied. A crucial difference, however, is that in the dominance rule an assignment α satisfying $\mathcal{C} \cup \mathcal{D}$ but falsifying C need only be mapped to an assignment α' that satisfies $\mathcal{C}$, but not necessarily $\mathcal{D}$ or C. On the other hand, the new assignment α' should satisfy the strict inequality $\alpha' \prec_f \alpha$ and not just $\alpha' \preceq_f \alpha$ as in the redundance rule.

To see that this is sound, we can argue by induction that if these conditions are satisfied, then it must be possible to construct an assignment that satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\}$, and achieves at least as good a value with respect to the objective function f, by iteratively applying the witness of the dominance rule. Note, however, that the derivation in the proof log does not actually carry out this construction, but just provides an existential proof that such a construction would be possible in principle. We sketch the proof of the soundness of this argument here, referring the reader to [BGMN22] for the missing details.

For the base case, if the assignment α' obtained from α satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\}$, we are done. Otherwise, since α' satisfies $\mathcal{C}$, and since $\mathcal{D}$ has previously been derived from $\mathcal{C}$, it can be shown that there exists an assignment α'' that satisfies $\mathcal{C} \cup \mathcal{D}$ and is such that $\alpha'' \prec_f \alpha' \prec_f \alpha$ holds. If α'' does not satisfy C, then this assignment satisfies exactly the same conditions as the assignment α that we started with, and the whole argument can be repeated to get $\alpha^{(4)} \prec_f \alpha^{(3)} \prec_f \alpha''$. Arguing by induction, we get a strictly decreasing sequence of assignments with respect to $\prec_f$. Since the set of possible assignments is finite, this sequence has to terminate eventually with an assignment α^* that satisfies all constraints in $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ and for which the inequality $f\restriction_{\alpha^*} \leq f\restriction_{\alpha}$ holds.

More formally, we would like to say that if v is the best value for the objective function achieved so far (or ∞), and if the preorder $\mathcal{O}_{\preceq}$ has been loaded to be applied to $\vec{z}$, then the pseudo-Boolean inequality C can be derived by dominance-based strengthening given a substitution ω such that

$$\mathcal{C} \cup \mathcal{D} \cup \{f \leq v-1\} \cup \{\neg C\} \vdash \mathcal{C}\restriction_\omega \cup \mathcal{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \cup \neg\mathcal{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega) \cup \{f\restriction_\omega \leq f\}\,, \tag{5.4}$$

where $\mathcal{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z})$ and $\neg\mathcal{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega)$ taken together imply that $\alpha \circ \omega \prec \alpha$ for any assignment α. A technical problem with this proposal is that the pseudo-Boolean formula $\mathcal{O}_{\preceq}$ may contain multiple constraints, so that the negation of it is not a set of pseudo-Boolean inequalities and thus is not in the correct syntactic format. To get around this, one can divide (5.4) into two separate conditions and move $\neg\mathcal{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega)$ to the premise of the implication, which eliminates the negation. After this rewriting step, we get the formal definition that C is derivable by the dominance-based strengthening rule if there is a substitution ω such that explicit derivations

$$\mathcal{C} \cup \mathcal{D} \cup \{f \leq v-1\} \cup \{\neg C\} \vdash \mathcal{C}\restriction_\omega \cup \mathcal{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \cup \{f\restriction_\omega \leq f\} \tag{5.5a}$$

$$\mathcal{C} \cup \mathcal{D} \cup \{f \leq v-1\} \cup \{\neg C\} \cup \mathcal{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega) \vdash \bot \tag{5.5b}$$

can be provided.

As for the redundance rule, ω should be given as a list `var1 -> val1  var2 -> val2 ...` of variables and what these variables are mapped to by ω, with the arrow symbols `->` being optional in the augmented format. The ⟨*subproofs*⟩ part contains a derivation for every constraint on the right-hand side

of (5.5b)–(5.5b) with the same conventions that simple proofs can be automatically filled in by the proof checker, and with the level of proof checker helpfulness depending on whether we use the augmented or kernel format. We will discuss the syntax and semantics of such subproofs next, after writing down for the record that the proof goals for a dominance application are labelled as follows:

1. For $\mathcal{C}\restriction_\omega$, each proof goal $D\restriction_\omega$ is labelled by the constraint ID of D in the core constraint set $\mathcal{C}$.
2. The remaining proof goals get labels with a distinguishing prefix "#" as follows:
   (a) Labels #1, #2, …, #N for $N = |\mathcal{O}_\preceq|$ refer to proof goals for the order $\mathcal{O}_\preceq$ with one proof goal per constraint in the order (or $N = 0$ if no order is loaded).
   (b) Label #$N + 1$ refers to the proof goal for the negated order in (5.5b). Note that this proof goal behaves slightly differently from the others in that it directly adds a list of assumptions from which contradiction must be derived.
   (c) Label #$N + 2$ refers to the proof goal for the inequality $f\restriction_\omega \leq f$ if applicable, i.e., if the problem contains an objective function f.

5.5 Subproofs

The rules for subproofs are arguably somewhat complex. A good way to understand the syntax is to run VeriPB on files such as `tests/integration_tests/correct/dominance/example.pbp` in the repository with the options `--trace --useColor`, which will (among other things) display the required proof goals.

If all proof goals for a strengthening rule application can be automatically derived by the proof checker, then there is no need for a list of subproofs and the strengthening rule can be stated as

⟨*strengthening rule*⟩ ⟨*derived constraint* C⟩ ; ⟨*substitution* ω⟩

on a single line, where ⟨*strengthening rule*⟩ is `red` or `dom`. In the general case, however, the proof checker will need to be provided with a list of explicit subproofs, and then the the format of the strengthening rules is

⟨*strengthening rule*⟩ ⟨*derived constraint* C⟩ ; ⟨*substitution* ω⟩ ; `begin`
  ⟨*list of subproofs*⟩
`end [` ⟨*id*⟩ `]`

The optional constraint ID at the end of the `red` or `dom` rule with subproofs, if provided, refers to a contradiction that is derived in the top-level subproof of these rules. As soon as such a constraint has been derived, there is no need to check any remaining proof goals.

In ⟨*list of subproofs*⟩ implicational derivation steps can be interleaved with proof goals, where the latter of which are formatted as follows:

`proofgoal` ⟨*pid*⟩
  `[` ⟨*list of implicational steps*⟩ `]`
`end` ⟨*id*⟩

Each proof goal is labeled with a proof goal ID ⟨pid⟩ which is on the form ⟨id⟩ or #⟨id⟩ as explained at the end of Sections 5.4.1 and 5.4.2 for redundance and dominance, respectively.

Proof checking proceeds through subproofs sequentially, populating the database according to the implicational commands, and checking proof goals with the accumulated database up to that proof goal. This is illustrated below with a commented trace, starting from a constraint database $\mathcal{S}$ which is the union of the core set and the derived set at this point in the overall proof.

`* Initial database` $\mathcal{S}$
⟨*list of implicational steps*⟩
`* Derived database` $\mathcal{S}'$ `from` $\mathcal{S}$ `following implicational steps`
`proofgoal` ⟨*pid*⟩
  `* Add constraint(s) from proof goal for` ⟨*pid*⟩ `to` $\mathcal{S}'$

```
  ⟨list of implicational steps⟩
  * Derived database S'' from S' implicationally
end ⟨id⟩
* Check if ⟨id⟩ is a contradiction
* Rewind to database S' and continue
...
```

If subproof checking succeeds, the argument at the end of the explicit subproofs specifies a constraint ID at which ⟨*list of subproofs*⟩ derives contradiction. If such a contradiction is derived, the proof of the redundancy or dominance step is complete. Otherwise, all proof goals for the given redundancy or dominance step are checked to either be explicitly covered by a proof goal in the subproofs or implicitly covered by automatic proof.

5.5.1 Automatically Generated Subproofs in Kernel Format

The kernel format uses the following rules to determine if a proof goal requires explicit proofs (assuming contradiction has not been derived):

1. All #⟨id⟩ proof goals must have explicit proofs.
2. For ⟨id⟩ proof goals, let C be the constraint corresponding to ⟨id⟩ in the database and ω be the substitution of the redundancy or dominance step. The proof goal ⟨id⟩ can be skipped if:[5]
   (a) C is untouched by the substitution;
   (b) ω only assigns literals to true in (the normalized form of) C;
   (c) C and $C\restriction_\omega$ are contradictory;
   (d) C is a duplicated constraint and one copy of it already has an explicit proof.

5.5.2 Automatically Generated Subproofs in Augmented Format

In addition to what is mentioned in Section 5.5.1, in the augmented format automatic checks for reverse unit propagation and syntactic implication from each constraint in the database can be performed.

5.6 Deletion Rules

Deletion is a complex topic, not least because the pseudo-Boolean proof format supports both *deleting by reference* (i.e., by specifying a constraint ID) and *deleting by specification* (i.e., describing the constraint to be deleted). The latter type of deletion is not a good fit for the proof format, but is supported for ease of integration with SAT solvers using standard DRAT-style proof logging.

As a further complication, a pseudo-Boolean constraint C in the core set $\mathcal{C}$ should ideally be deleted only if it can be proven that C can be recovered from $\mathcal{C} \setminus \{C\}$, which is referred to as *checked deletion* in [BGMN22]. In the version of VERIPB proposed for the SAT competition 2023 only unchecked deletion is supported, however.

5.6.1 Deletion Rules in Kernel Format

The kernel format only supports deleting constraints by referring to their constraint IDs, and in addition forces the proof logger to be aware of whether a core set constraint or a derived set constraint is being erased.

Deletion from derived set The command

```
del ⟨id1⟩ ⟨id2⟩ ⟨id3⟩ ...
```

[5] **Work in progress:** And additional case that is intended to be covered in the kernel format is when the constraint $C\restriction_\omega$ is already contained in the database, but this has not yet been implemented.

deletes a list of constraints with specified IDs from the derived set $\mathcal{D}$. It is an error if any constraint is instead in the core set $\mathcal{C}$ or is not in the constraint database at all.

Deletion from core set The unchecked core deletion command

`delc` ⟨*id1*⟩ ⟨*id2*⟩ ⟨*id3*⟩ ...

provided for the SAT competition 2023 first checks that all constraint IDs in the specified list identify constraints currently in the core set. Provided that no order (or the empty order) has been loaded, or alternatively that the derived set $\mathcal{D}$ is currently empty, all the specified constraints are deleted from the core. Otherwise the command generates an error.

Note that the fact that core deletion is unchecked means that the current database $\mathcal{C} \cup \mathcal{D}$ can turn satisfiable although the input formula is unsatisfiable.

5.6.2 Deletion Rules in Augmented Format

The augmented proof format supports a general deletion-by-reference command, where the proof logger is required to know the constraint ID but does not need to be aware of the difference between core set and derived set. It also provides a deletion-by-specification command that provides the encoding of the constraint to be deleted. Deletion by specification should be avoided if possible, but is supported as a convenience for SAT solvers already equipped with DRAT-style proof logging as well as for solvers in more powerful paradigms where the constraints in the proof constraints database might not match well what the solver is keeping track of in its own constraints database.

Deletion by reference The command

`del id` ⟨*id1*⟩ ⟨*id2*⟩ ⟨*id3*⟩ ...

has the same effect as checking the type of each constraint ⟨id⟩ in the list and then issuing a delete core command `delc` or delete derived command `deld` depending on the type of the constraint. All constraint IDs must be valid references to constraints currently in the database.

Deletion by specification The command

`del spec` ⟨*pseudo-Boolean inequality C in OPB format*⟩

is an error if there is no constraint C in the database. Otherwise, C is marked for deletion as per the description in Section 5.6.3.

5.6.3 Semantics in Augmented Format for Mixed Deletion by Reference and Specification

We implement a multiset deletion semantics for deletion by associating to constraint C in the database a delete-by-specification kill counter $K_s(C)$ and two lists $L_{\mathcal{D}}(C)$ and $L_{\mathcal{C}}(C)$ containing the constraint IDs with which C appears in the derived set $\mathcal{D}$ and core set $\mathcal{C}$, respectively.

When deletion by reference for a constraint C is encountered, the specified constraint ID is removed from the appropriate list $L_{\mathcal{D}}(C)$ or $L_{\mathcal{C}}(C)$. When deletion by specification of C is encountered, $K_s(C)$ is incremented by 1. After any of the above updates to $K_s(C)$, $L_{\mathcal{D}}(C)$, or $L_{\mathcal{C}}(C)$, the following procedure is run:

1. If $K_s(C) < |L_{\mathcal{D}}(C)| + |L_{\mathcal{C}}(C)|$, then there are still derived copies of C left, and no action is taken.
2. Else if $K_s(C) = |L_{\mathcal{D}}(C)| + |L_{\mathcal{C}}(C)|$, then as many copies of C as are available in the database have been deleted. Therefore the constraint C is completely removed from the database by issuing `deld` commands for all IDs in $L_{\mathcal{D}}(C)$ and `delc` commands for all IDs in $L_{\mathcal{C}}(C)$, after which we set $L_{\mathcal{D}}(C) = L_{\mathcal{C}}(C) = \emptyset$ and $K_s(C) = 0$.
3. The case $K_s(C) > |L_{\mathcal{D}}(C)| + |L_{\mathcal{C}}(C)|$ is impossible, as the system ensures $K_s(C) \leq |L_{\mathcal{D}}(C)| + |L_{\mathcal{C}}(C)|$.

6 Formally Verified Proof Checking

The kernel proof checker CAKEPB has been formally verified in the HOL4 theorem prover [SN08] using the CAKEML suite of tools for program verification, extraction, and compilation [TMK+19, GMKN17, MO14]. In this section, we present the verification guarantees for CAKEPB_CNF, a version of CAKEPB equipped with a DIMACS CNF parser frontend for UNSAT proof checking with pseudo-Boolean proof logging.

6.1 Summary of Kernel Format

We summarize the kernel format with reference to the derivation rules presented in Section 5. In general, rules in the kernel format have the same semantics on the proof state, but may have additional syntactic restrictions or requirements, e.g., requiring more explicit subproofs.

Constraint IDs The kernel format does not support the interpretation of a negative integer $-N$ as the constraint with ID $\mathsf{maxId} + 1 - N$.

Load formula The `f` command behaves identically in the kernel format. It must be the first command in the input pseudo-Boolean proof after the header (see Section 2.1 for an example).

Move to core Only the `core id` command for moving constraints to the core is supported.

Implicational rules Only the `pol` implicational rule is supported.

Orders The kernel format requires an empty `aux` line. In addition, there must be explicit proofs for all proof goal in the transitivity proof for the specified order.

Strengthening rules and subproofs The kernel format supports both `red` and `dom` strengthening commands, but requires more explicit subproofs. see Section 5.5.1 for the exact explicit subproof requirements in the kernel format.

Deletion rules Only the `deld` and `delc` deletion by ID commands are supported in kernel format. Notably, deletion by specification is not supported and must be compiled away.

Conclusions section In the kernel format, the conclusion section *must* be explicitly given all required IDs (cf. Section 4.4). In particular, for an unsatisfiability proof, the conclusion section

`conclusion UNSAT :` $\langle id \rangle$

must have a constraint ID specifying the contradictory constraint in the database.

6.2 Verified Correctness Theorem for CAKEPB_CNF

The end-to-end verified correctness theorem for CAKEPB_CNF is shown in Figure 2. This theorem can be intuitively understood in four parts, corresponding to the indicated lines (6.1)–(6.4):

- The theorem assumes (6.1) that the CAKEML-compiled machine code for CAKEPB_CNF is executed in an x64 machine environment set up correctly for CAKEML. The definition of cake_pb_cnf_run is shown below, where the first line (wfcl $cl \wedge$ wfFS $fs \wedge$ ...) says the command line cl and filesystem fs match the assumptions of CAKEML's FFI model. The second line says that the compiled code (cake_pb_cnf_code) is correctly set up for execution on an x64 machine.

$$\begin{aligned}&\mathsf{cake\_pb\_cnf\_run}\ cl\ fs\ mc\ ms \stackrel{\text{def}}{=}\\ &\quad \mathsf{wfcl}\ cl \wedge \mathsf{wfFS}\ fs \wedge \mathsf{STD\_streams}\ fs \wedge \mathsf{hasFreeFD}\ fs \wedge\\ &\quad \mathsf{installed\_x64}\ \mathsf{cake\_pb\_cnf\_code}\ mc\ ms\end{aligned}$$

- Under these assumptions, the CAKEPB_CNF program is guaranteed to never crash (6.2). However, it may run out of resources such as heap or stack memory (extend_with_resource_limit ...). In these cases, CAKEPB_CNF will fail gracefully and report out-of-heap or out-of-stack on standard error.

$$\vdash \text{cake\_pb\_cnf\_run } cl\ fs\ mc\ ms \Rightarrow \tag{6.1}$$

$$\left.\begin{array}{l}\text{machine\_sem } mc\ (\text{basis\_ffi } cl\ fs)\ ms \subseteq \\ \quad \text{extend\_with\_resource\_limit} \\ \quad\quad \{\ \text{Terminate Success } (\text{cake\_pb\_cnf\_io\_events } cl\ fs)\ \}\ \wedge\end{array}\right\} \tag{6.2}$$

$$\left.\begin{array}{l}\exists\, out\ err. \\ \quad \text{extract\_fs } fs\ (\text{cake\_pb\_cnf\_io\_events } cl\ fs) = \\ \quad\quad \text{SOME } (\text{add\_stdout } (\text{add\_stderr } fs\ err)\ out)\ \wedge\end{array}\right\} \tag{6.3}$$

$$\left.\begin{array}{l}\quad \text{if } out = \ll\text{s VERIFIED UNSAT\textbackslash n}\gg \text{ then} \\ \quad\quad \text{LENGTH } cl = 3 \wedge \text{inFS\_fname } fs\ (\text{EL } 1\ cl)\ \wedge \\ \quad\quad \exists fml. \\ \quad\quad\quad \text{parse\_dimacs } (\text{all\_lines } fs\ (\text{EL } 1\ cl)) = \text{SOME } fml\ \wedge \\ \quad\quad\quad \text{unsatisfiable } (\text{interp } fml) \\ \quad\quad \text{else } out = \ll\gg\end{array}\right\} \tag{6.4}$$

Figure 2: The end-to-end correctness theorem for the CakeML pseudo-Boolean proof checker with a CNF parser

- Upon termination, the CakePB_cnf program will output some (possibly empty) strings out and err to the standard output and standard error streams, respectively (6.3).
- The key verification guarantee (6.4) is that, whenever the string "s VERIFIED UNSAT" is printed to standard output, the input CNF file (first command line argument) parses in DIMACS format to a CNF which is unsatisfiable. No other output is possible on standard output; error strings are always printed to standard error.

Internally, CakePB_cnf transforms input CNF clauses (in DIMACS format) to normalized pseudo-Boolean constraints, as exemplified by 2.1a and 2.1b. This transformation is formally verified to preserve satisfiability as part of the end-to-end correctness theorem shown in Figure 2.

Note that the CakePB_cnf tool has an essentially identical correctness theorem to an existing verified Boolean unsatisfiability proof checking tool [THM21]. In fact, these tools share exactly the same definitions of DIMACS CNF parsing, Boolean satisfiability semantics, and all of the CakeML's standard assumptions.

6.3 Complexity and Empirical Evaluation

All of the commands in the kernel format are designed to minimize the need to search over the entire constraint database. For example, each implicational and deletion proof step can be performed in linear time with respect to the size of that step.

The only proof steps that scale linearly with respect to the size of the constraint database are redundancy and dominance-based strengthening steps. For either of these steps, the proof checker potentially needs to loop over the entire constraint database to check all the necessary proof goals. However, the maximum size of the database is linear in the size of the input formula and the proof. Therefore, the overall complexity of the verified proof checker is polynomial in the size of the input formula and proof, as required.

Table 1 shows an empirical evaluation of the verified proof checking pipeline on a selected suite of example proofs, generated using BreakID[Bre] [6] and Kissat[Kis][7] to solve SAT competition instances of the last years and theoretical instances.

[6] https://bitbucket.org/krr/breakid/src/veriPB/

[7] https://gitlab.com/MIAOresearch/tools-and-utilities/kissat_fork

Table 1: Example timings for verified proof checking using VERIPB and CAKEPB_CNF. All times are in seconds.

Benchmark	VeriPB Time (s)	CakePB Time (s)
queen14_14.col.14.cnf	6.5	52.3
harder-php-025-024.sat05-1191.reshuffled-07.cnf	9.3	30.5
Pb-chnl15-16_c18.cnf	13	43.2
tseitin_n104_d3.cnf	4.2	3.9
rphp_p6_r28.cnf	123	68.2

Acknowledgements

We wish to acknowledge the monumental contributions of Stephan Gocht [Goc22], without whom there would not have been any pseudo-Boolean proof checker.

References

[BCH21] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12651 of *Lecture Notes in Computer Science*, pages 59–75. Springer, March-April 2021.

[BGMN22] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, pages 3698–3707, February 2022.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 7, pages 233–350. IOS Press, 2nd edition, February 2021.

[Bre] BreakID. https://bitbucket.org/krr/breakid.

[BT19] Samuel R. Buss and Neil Thapen. DRAT proofs, propagation redundancy, and extended resolution. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, volume 11628 of *Lecture Notes in Computer Science*, pages 71–89. Springer, July 2019.

[CS15] Geoffrey Chu and Peter J. Stuckey. Dominance breaking constraints. *Constraints*, 20(2):155–182, April 2015. Preliminary version in *CP '12*.

[EGMN20] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.

[GMKN17] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In Hongseok Yang, editor, *ESOP*, volume 10201 of *LNCS*, pages 584–610. Springer, 2017.

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

[GMN20] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

[GMN22] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022.

[GMNO22] Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Certified CNF translations for pseudo-Boolean solving. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.

[GN21] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[Goc22] Stephan Gocht. *Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, Lund, Sweden, June 2022. Available at https://portal.research.lu.se/en/publications/certifying-correctness-for-combinatorial-algorithms-by-using-pseu.

[HHW13] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, June 2013.

[HKB17] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, August 2017.

[Kis] Kissat SAT solver. http://fmv.jku.at/kissat/.

[MO14] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014.

[RM16] Olivier Roussel and Vasco M. Manquinho. Input/output format and solver requirements for the competitions of pseudo-Boolean solvers. Revision 2324. Available at http://www.cril.univ-artois.fr/PB16/format.pdf, January 2016.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.

[THM21] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in CakeML. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *TACAS*, volume 12652 of *LNCS*, pages 223–241. Springer, 2021.

[TMK+19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019.

[VDB22] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.

[Ver] VeriPB: Verifier for pseudo-Boolean proofs. https://gitlab.com/MIAOresearch/software/VeriPB.