Proof Logging for Smart Extensional Constraints

Matthew J. McIlree
University of Glasgow, UK

Ciaran McCreesh
University of Glasgow, UK

Abstract

Proof logging provides an auditable way of guaranteeing that a solver has produced a correct answer using sound reasoning. This is standard practice for Boolean satisfiability solving, but for constraint programming, a challenge is that every propagator must be able to justify all inferences it performs. Here we demonstrate how to support proof logging for a wide range of previously uncertified global constraints. We do this by showing how to justify every inference that could be performed by the propagation algorithms for two families of generalised extensional constraint: "Smart Table" and "Regular Language Membership".

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Discrete optimization

Keywords and phrases Constraint programming, proof logging, extensional constraints

Digital Object Identifier 10.4230/LIPIcs.CP.2023.26

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.8132457`

Funding Part of this work was done while the authors were participating in a programme at the Simons Institute for the Theory of Computing.
Ciaran McCreesh: Supported by a Royal Academy of Engineering research fellowship.

Acknowledgements The authors would like to thank Jakob Nordström for several helpful discussions regarding pseudo-Boolean encodings and unit propagation.

1 Introduction

A proof log for a problem-solving algorithm provides a verifiable certificate that the result is correct, and also an auditable record of the steps taken to obtain that result. In the field of Boolean satisfiability (SAT), proof-logging has become an expected capability of modern solvers, with a series of proof formats including DRAT [12, 13], LRAT [7], and FRAT [2] widely accepted for independent verification. A similar standard practice has not yet been adopted for Constraint Programming (CP) due to the difficulties of creating easily verifiable proofs for the more expressive formulations and reasoning present in this paradigm. However, recent work by Gocht et al. [10] has shown how the VeriPB proof system [3, 9] can be used to certify the reasoning carried out for several important expressive global constraints, offering a strong candidate for a complete, general CP proof logging method. In this setting, every propagator for a global constraint must be able to do two things: describe its semantics in a lower-level *pseudo-Boolean* format, and justify any reasoning it carries out using either *cutting planes* [6] or *reverse unit propagation* (RUP) [11] reasoning. Describing a constraint's semantics is a well-understood problem, but justifying propagation is not. Gocht et al. demonstrated proof logging for a range of global constraints and propagation strengths, including bounds-consistent integer linear inequalities and domain-consistent table constraints, but it remains an open question whether *every* global constraint propagation algorithm can be justified in this manner.

This paper shows that the reasoning carried out by domain-consistent propagators for the `SmartTable` [16] and `Regular` [19] constraints can similarly be proof logged efficiently inside the VeriPB proof system. As well as being useful in their own right, these two constraints provide the necessary building blocks for implementing many others, since they allow for efficient strong propagation for extensional constraints that cannot be expressed efficiently as a conventional table. For example, `SmartTable` can be used to implement the `Lex`, `AtMostOne` and `NotAllEqual` constraints [16], whilst `Regular` can be used to implement `Stretch`, `Geost` and `DiffN` [15, 19]. Together, these two new constraints bring us a lot closer to fully auditable combinatorial solving, particularly in areas such as workforce scheduling where legal restrictions apply, and where algorithmic decisions can have a large effect upon people's livelihoods.

2 An Overview of How Proof Logging Works

The VeriPB proof system is built upon pseudo-Boolean (PB) problems. A PB model is a restricted constraint satisfaction problem, where all variables x_i have domain $\{0, 1\}$ and all constraints are integer linear inequalities of the format $\sum_i c_i \ell_i \geq A$ where $c_i, A \in \mathbb{Z}$, and each ℓ_i is a pseudo-Boolean *literal*: either a variable x or its negation $\bar{x} = 1 - x$. For clarity and by convention we will always refer to CP variables with upper-case symbols and PB variables with lower-case symbols. Additionally, to save space and make the meaning clearer we will make use of *reified* PB constraint shorthands:

- $(\ell_1 \wedge \cdots \wedge \ell_m) \implies \sum_i c_i x_i \geq A$ means $\sum_{i=1}^n c_i x_i + J\bar{\ell}_1 + \cdots + J\bar{\ell}_m \geq A$ where J is chosen to be suitably large, e.g. $J = A - \sum_{i=1}^n \min(c_i, 0)$.
- $\sum_i c_i x_i \geq A \implies (\ell_1 \wedge \cdots \wedge \ell_m)$ means $\{\sum_{i=1}^n -c_i x_i + K\bar{\ell}_t \geq -A + 1 \;:\; 1 \leq t \leq m\}$ where K is chosen to be suitably large, e.g. $K = -A + 1 - \sum_{i=1}^n \min(c_i, 0)$.
- $(\ell_1 \wedge \ldots \wedge \ell_m) \iff \sum_i c_i x_i \geq A$ means both of the above together.

PB constraints can also be viewed as a superset of conjunctive-normal form (CNF) for SAT problems, since a clause such as $x_1 \vee \bar{x}_2 \vee x_3$ is always equivalent to a PB constraint such as $x_1 + \bar{x}_2 + x_3 \geq 1$. This means that it is straightforward to generalise known SAT encodings of CP constraints to PB. The concept of *unit propagation* from SAT can also be generalised to a PB setting. We say a PB constraint *propagates* a literal ℓ if it cannot possibly be satisfied unless $\ell = 1$. For the "at least 1" constraints that can be viewed as clauses, this is the same as SAT unit propagation, but in general it is enforcing integer bounds consistency [5] on the PB inequalities. The (generalised) notion of unit propagation for PB constraints is therefore repeatedly setting any literals that propagate until either a contradiction or a fixed point is reached.

An auditable constraint solver following the proof-logging methodology of Gocht et al. [10] must be able to produce a pseudo-Boolean representation of the problem being solved, and then justify its backtracking and inference steps in a proof log by outputting further PB constraints derivable by one of the proof rules allowed by the *VeriPB* proof checker. The full proof system associated with VeriPB is described in detail by Bogaerts et al. [3], but for the purposes of this paper the only rule that is needed is the *reverse unit propagation* (RUP) rule, which says that a PB constraint D is derivable from a set of PB constraints F if applying the pseudo-Boolean generalisation of unit propagation (repeated integer bounds consistency) to $F \cup \bar{D}$ results in a contradiction. Intuitively, it is a constraint that "obviously" follows once it is pointed out to the verifier.

This paper will omit discussion of how the variables and constraints of CP problem can be represented as PB constraints and variables in general, focussing specifically on how `Regular` and `SmartTable` can be represented. We will assume that we have access to a collection of properly defined PB variables $x_{b\text{neg}}, x_{b0}, x_{b1}, x_{b2} \ldots$ representing bits in the two's-complement encoding of a CP variable X (with suitable constraints to enforce the domain), but also that we can freely use flags of the form $x_{=v}$, which are defined to equal 1 if and only if $X = v$. Explanation of how these encodings can be achieved is given by Gocht et al. [10].

Similarly, we will omit detailed discussion of how complete proofs of solutions, unsatisfiability, optimality, or enumeration are achieved. We simply note that each proof is essentially a description of the solver's backtracking search tree. Any time the solver backtracks, a RUP constraint that encodes the negated conjunction of the currently guessed assignments is emitted. A new propagation algorithm can fit into this existing framework by ensuring that the encoding of any specific filtering and failure inferences it makes are visible to the verifier on reverse unit propagation of the backtrack constraint. So for example, if a solver has guessed $A = 2, B = 1$; and a propagator is able to infer $C \neq 1$ then the proof logging methodology requires that the PB constraint

$$\bar{a}_{=2} + \bar{b}_{=1} + \bar{c}_{=1} \geq 1 \tag{1}$$

is somehow derived, which can be interpreted as

$$\bar{a}_{=2} \lor \bar{b}_{=1} \lor \bar{c}_{=1} \quad \text{i.e.} \quad A = 2 \land B = 1 \implies C \neq 1. \tag{2}$$

This is in many ways similar to how lazy clause generating solvers work [18], except that the new constraints introduced must be justified, rather than asserted.

3 Proof Logging for Smart Table Constraints

A smart table constraint generalises the idea of a table constraint to allow wildcards and comparisons, allowing for compact representations for a much larger set of relations, whilst still retaining an efficient domain-consistent propagator [16, 20, 21]. There are several ways to define `SmartTable` (also called `HybridTable`), depending on which restriction types are allowed within tuples [4]. In this work we restrict our focus to unary comparisons, binary comparisons, and set membership, and define `SmartTable` as follows.

▶ **Definition 1.** *Smart Table Constraint*
Let $\mathbf{X}$ *be a sequence of finite-domain variables* $\langle X_1, \ldots, X_n \rangle$. *A smart entry constraint is a unary or binary constraint in one of three possible forms:*

1. *<var> <op> a*
2. *<var>* $\in S$ *or <var>* $\notin S$
3. *<var> <op> <var>*

where a is an integer constant, S is a set of integer constants, <op> denotes an operator in the set $\{<, \leq, =, \neq, \geq, >\}$, *and <var> denotes a variable. A smart tuple is a set of these smart entry constraints, and a smart table is a set of smart tuples. When a variable does not appear in a given tuple, it is implicitly unrestricted (equivalent to a wildcard entry).*

For a given smart table T, *where the scope of every smart entry constraint is a subset of* $\mathbf{X}$, *a smart table constraint* `SmartTable`$(\mathbf{X}, T)$ *requires that there is at least one smart tuple in* T *where every smart entry constraint holds. It can hence be thought of as a special case of a disjunction of conjunctions of simple constraints.*

3.1 PB Encodings for Smart Table Constraints

Recall that in order to support proof logging for a constraint, we must be able to describe its semantics in a lower-level PB form. Let T be a smart table with smart tuples $\sigma_1, \ldots, \sigma_k$. We can encode the `SmartTable` constraint by making use of some auxiliary PB variables. For each smart tuple σ_i in the table we will have a selector PB variable s_i that controls whether the tuple is active (i.e. not yet infeasible). Then for each smart entry $C_j \in \sigma$ we will have a PB variable e_{ij} that is set to 1 if and only the constraint C_j is satisfied, which we denote by $\texttt{satisfied}(C_j)$.

Since smart entries are themselves simple binary and unary constraints, with all of them having PB encodings that already well understood, the process of encoding this `satisfied` property is straightforward. Inequalities on one or two variables can be encoded by imposing the inequality on the evaluation of the bit encodings. For example, for two CP variable A and B both with domain $\{1, \ldots 7\}$, and recalling that PB variables a_{bi} and b_{bi} denote the i_{th} bits in the encoding of the variables A and B, we would have

$$\texttt{satisfied}(A < B) := -a_{b0} - 2a_{b1} - 4a_{b2} + b_{b0} + 2b_{b1} + 4b_{b2} \geq 1. \tag{3}$$

Other inequality constraints can be handled similarly, adding or subtracting constants from the right-hand side as necessary. For $\in$ and $\notin$ it is simply a case of imposing "at least n" constraints on the flags representing the disallowed values, e.g.

$$\texttt{satisfied}(A \in \{1,3,5\}) := \bar{a}_{=2} + \bar{a}_{=4} + \bar{a}_{=6} + \bar{a}_{=7} \geq 4, \tag{4}$$

$$\texttt{satisfied}(B \notin \{1,3,5\}) := \bar{b}_{=1} + \bar{b}_{=3} + \bar{b}_{=5} \geq 3. \tag{5}$$

Finally, for $=$ and $\neq$ we can make direct use of the $x_{=v}$ flag if the right-hand side is a value. If the right-hand side is instead another variable we make use of further intermediate flags to express the relation in terms of strict or non-strict inequalities, so:

$$\texttt{satisfied}(A = B) := f_{A \geq B} + f_{A \leq B} \geq 2, \tag{6}$$

$$\texttt{satisfied}(A \neq B) := f_{A > B} + f_{A < B} \geq 1, \tag{7}$$

where the flags are enforced with two-way implication, e.g.

$$f_{A<B} \iff \texttt{satisfied}(A < B). \tag{8}$$

Obviously, the negation of these encodings can be expressed by encoding the negation of the constraint, e.g.

$$\neg\texttt{satisfied}(A < B) := \texttt{satisfied}(A \geq B), \tag{9}$$

$$\neg\texttt{satisfied}(A \neq B) := \texttt{satisfied}(A = B). \tag{10}$$

With the procedures for each of these smart entry constraint encodings in place, and recalling that we know how to reify arbitrary PB constraints on a literal, the PB model for smart table can then be created according to the following specification.

▶ **Encoding 1.** *Smart Table PB Encoding*

```
1: for all $\sigma_i \in T$
2:     for all $C \in \sigma_i$
3:         $e_C \iff \texttt{satisfied}(C)$
4:     $s_i \iff \sum_{C \in \sigma_i} e_C \geq |\sigma_i|$
5: $s_1 + s_2 + \cdots + s_{|T|} \geq 1$
```

3.2 Justifying Smart Table Propagation

The goal of the propagator for the smart table constraint, as with many global constraint propagators, is to achieve *domain consistency*. As a running example, suppose we had a smart table constraint on three variables, A, B, C, all with domains $\{1, 2, 3\}$, defined by the following table T with two tuples:

$$T := \{\sigma_1, \sigma_2\}; \qquad \sigma_1 = \{(A < B), (A \in \{1, 2\}), (C = 3)\}, \tag{11}$$

$$\sigma_2 = \{(A = B), (A \neq 1), (B \geq C)\}. \tag{12}$$

By inspection, we can observe that the literal $B = 1$ does not have any support on this constraint, as it would contradict $A < B$ on the first tuple and $A = B, A \neq 1$ together on the second. Therefore, a propagation algorithm that achieves domain consistency should immediately remove the value 1 from the domain of B.

The propagation algorithm described by Mairy et al. [16] uses a “Simple Tabular Reduction” strategy to eliminate such unsupported values. Essentially it initialises a set *sl* (for “supportless”) with every variable value/pair that is possible given the current domains of variables, and then iterates through the tuples, removing any values that they in turn support. The literals that are left in *sl* are not supported by any tuple and hence the ones that should be eliminated. A key observation is that each smart tuple σ, as a conjunction of simple constraints, can be thought of as a small CSP P_σ in its own right, and so the supported values are just all the solutions for this problem. It is shown that as long as the constraint graph of P_σ is acyclic it can be effectively filtered as a collection of tree CSPs via a two pass filtering process. Due to the restricted structure of the smart tuples, the result of this is a collection of copies of the domains where the only values present are those that appear in some solution (global consistency for P_σ), and so this can be used to remove values from *sl* in polynomial time.

For proof logging, the main concern is then how to justify the elimination of these unsupported values. In particular, for a sequence of guesses $\mathcal{G}$ and newly derived unsupported assignment $X \neq v$, we would like to log the PB constraint

$$\wedge\mathcal{G} \implies \bar{x}_{=v} \geq 1. \tag{13}$$

In some situations this might follow directly by reverse unit propagation, as is always the case for proof-logging the classical table constraint as shown by Gocht et al. [10]. For smart tables, however, we can show that more work is required in general. Going back to the previous example and following Encoding 1, we note that if we try to derive $\bar{b}_{=1} \geq 1$ by reverse unit propagation, we do not reach a contradiction after propagating its negation $b_{=1}$. This is despite $B = 1$ lacking support in the table. To see why this is the case, note that the only way unit propagation of a non-auxiliary literal such as $b_{=1}$ could falsify the whole model would be for it to eventually result in all the selectors s_i being falsified. In turn, the only way that could happen is for there to be at least one e_{iC} falsified for every $i \in \{1, \ldots, |T|\}$. We can observe that this does not happen on propagation of $b_{=1}$. The smart entry constraints that are responsible for eliminating $(B = 1)$'s support are $A < B$ in the first tuple and both $A = B, A \neq 1$ in the second. From Encoding 1, the first of these is present in the PB model in the form of two constraints:

$$e_{A<B} \implies b_{b0} + 2b_{b1} - a_{b0} - 2a_{b1} \geq 1; \tag{14}$$

$$\bar{e}_{A<B} \implies -b_{b0} - 2b_{b1} + a_{b0} + 2a_{b1} \geq 1. \tag{15}$$

Assuming the bit encodings are correctly defined, propagating $b_{=1}$ should propagate b_{b0} and $\bar{b}_{b1}$, meaning (14) is reduced to

$$e_{A<B} \implies 1 - a_{b0} - 2a_{b1} \geq 1. \tag{16}$$

Now it might seem that $\bar{e}_{A<B}$ would then propagate, as we know $a_{b0} + 2a_{b1}$ is at least 1 due to the domain constraints of the variable A and so the right-hand side of the implication in (16) must be false. However, because the falsity does not follow from a contradiction intrinsic to the PB itself – in isolation it can be satisfied by having both b_{b0} and b_{b1} equal to 0 – unit propagation alone is not strong enough to determine that $\bar{e}_{A<B}$ must follow. A similar problem holds for the other tuple, where the contradiction arises due to two constraints being incompatible rather than just one being falsified.

Fortunately, we can ensure that the desired constraints do follow by RUP by first explicitly deriving some intermediate proof steps. For a sequence of guesses $\mathcal{G}$ and newly derived unsupported assignment $X \neq v$, instead of simply logging the PB constraint (13) as above, we first log

$$\wedge\mathcal{G} \implies \bar{x}_{=v} + \bar{s}_k \geq 1 \tag{17}$$

for each $k \in \{1, \dots, |T|\}$, i.e. first show by RUP that no tuple selector can be set to true without contradiction, and then derive from these the desired constraint (13). So in our running example, we would log

$$\bar{b}_{=1} + \bar{s}_1 \geq 1; \qquad \bar{b}_{=1} + \bar{s}_2 \geq 1; \qquad \bar{b}_{=1} \geq 1. \tag{18}$$

This is now sufficient for most cases, however, it is still possible to construct examples where even these constraints in the form of (17) do not follow by RUP. Suppose we have four variables W, X, Y, Z, all with domain $\{-2, -1, 0\}$, and then a single smart tuple of the form

$$(W < X), (X < Y), (Y < Z). \tag{19}$$

Clearly this is unsatisfiable, and so in particular $W = -2$ should be unsupported by the tuple. But the constraint

$$\bar{w}_{=-2} + \bar{s}_1 \geq 1 \tag{20}$$

does not unit propagate to contradiction. This can be seen by considering the PB encoding of the first smart entry constraint,

$$e_{W<X} \iff -2x_{b\text{neg}} + x_{b0} + 2w_{b\text{neg}} - w_{b0} \geq 1. \tag{21}$$

The negation of (20) will lead to $e_{W<X}, w_{b\text{neg}}$, and $\bar{w}_{b0}$ being propagated, which would mean the two PB constraints encoding both implication directions in (21) are reduced to

$$-2x_{b\text{neg}} + x_{b0} \geq -1; \qquad \text{and} \qquad 2x_{b\text{neg}} - x_{b0} \geq -1. \tag{22}$$

But then no further propagation results from these constraints, which can be seen either by calculating the *slack* value (see Elffers et al. [8]), or simply by observing that the value sets $\{0, 0\}$, $\{1, 1\}$, $\{0, 1\}$ would each satisfy both constraints if assigned to $\{x_{b\text{neg}}, x_{b0}\}$ and so both 0 and 1 are both supported possibilities for each variable. None of the other constraints contain bit variables for W and so it is clear that no propagation will result from these either, and hence no contradiction will be reached.

This weak propagation is not surprising given the context given by Gocht et al. [10], where integer linear inequality constraints need to log all the inferences made when enforcing domain consistency and cannot rely on the disallowed values being implicit. Fortunately for us, these same inferences are being made in the solver as part of the smart table propagator anyway, at the tree filtering stage, and so the desired constraints in the form of (17) and then (13) can still be logged providing we have first logged the inferences for binary constraints explicitly. The only difference in proof logging these and the proof logging of equality and inequality constraints described by Gocht et al. [10] is that each inference will be conditional on a tuple selector. For example, if a variable B has domain with maximum value l and the filtering process for $A < B$ updates the domain copy for A so that $A < l$, then we would log

$$\wedge\mathcal{G} \implies \bar{s}_1 + a_{<l} \geq 1; \tag{23}$$

where the flag $a_{<l}$ can be introduced via reification with corresponding constraints on the bit variables.

We can now be confident we have a complete proof logging procedure. We can show by an inductive argument on the binary representation that the PB constraints that encode the filtering inferences made by simple binary constraints such as (23) will always be RUP, providing they are logged in the same order as they are made by the solver. Then, any unsupported value must be unsupported in every tuple, and hence removed in the filtering process for some smart entry constraint. Given the negation of this removal, i.e. a variable being equal to some unsupported value, if it was removed due to a binary constraint then we have already logged a corresponding contradicting restriction and hence will unit propagate to contradiction. Otherwise, if it was removed due to a unary constraint it will propagate to contradiction immediately as the assignment will contradict a single PB constraint.

3.3 A Complete Worked Example for Smart Table Propagation

To demonstrate how this procedure works on a single round of the domain-consistent smart table propagator we will now show all the steps required for the running example. Firstly, we will have the PB constraints encoding that the of the domains of A, B, C are all $\{1, 2, 3\}$, i.e.

$$1 \leq a_{b0} + 2a_{b1} + 4a_{b2} \leq 3; \quad 1 \leq b_{b0} + 2b_{b1} + 4b_{b2} \leq 3; \quad 1 \leq c_{b0} + 2c_{b1} + 4c_{b2} \leq 3. \tag{24}$$

Next we can follow Encoding 1 to encode the smart table constraint. First we have the encodings of the four necessary auxiliary smart entry flags

$$e_{A<B} \iff b_{b0} + 2b_{b1} + 4b_{b2} - a_{b0} - 2a_{b1} - 4a_{b2} \geq 1; \tag{25a}$$

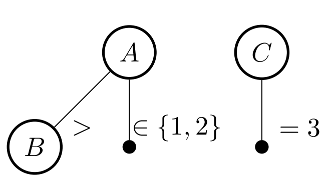
$$e_{A\in\{1,2\}} \implies \bar{a}_{=3} \geq 1; \tag{25b}$$
$$\bar{e}_{A\in\{1,2\}} \implies \bar{a}_{=1} + \bar{a}_{=2} \geq 2; \tag{25c}$$

$$f_{A>B} \iff a_{b0} + 2a_{b1} + 4a_{b2} - b_{b0} - 2b_{b1} - 4b_{b2} \geq 1; \tag{25d}$$
$$f_{A<B} \iff b_{b0} + 2b_{b1} + 4b_{b2} - a_{b0} - 2a_{b1} - 4a_{b2} \geq 1; \tag{25e}$$
$$e_{A=B} \iff \bar{f}_{A>B} + \bar{f}_{A<B} \geq 2; \tag{25f}$$

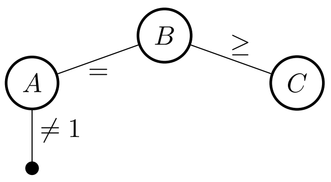
$$e_{B\geq C} \iff b_{b0} + 2b_{b1} + 4b_{b2} - c_{b0} - 2c_{b1} - 4c_{b2} \geq 0; \tag{25g}$$

Then we can use these to encode the two tuples

$$s_0 \iff e_{A<B} + e_{A\in\{1,2\}} + c_{=3} \geq 3; \tag{25h}$$
$$s_1 \iff e_{A=B} + \bar{a}_{=1} + e_{B\geq C} \geq 3; \tag{25i}$$



(a) The graph for P_{σ_1}, which consists of two trees.



(b) The graph for P_{σ_2}, which consists of a single tree.

Figure 1 The constraint graphs for the two sub-CSPs P_{σ_1} and P_{σ_2} represented by the smart tuples σ_1 and σ_2. Both graphs are acyclic and thus composed of trees.

And then finally we require

$$s_0 + s_1 \geq 1. \tag{25j}$$

Constraints that define the $x_{=v}$ flags are also included but are omitted here for brevity.

Now, for the domain-consistent propagator itself, we note that constraint networks of the smart tuples do indeed form acyclic graphs and so the procedure is applicable. This can be seen in Figure 1, where unary constraints are represented as edges to a dummy node. The first tuple is clearly made up of two small trees, and the second consists of a single tree.

Initially all variables have $\{1, 2, 3\}$ in their domains and the set of unsupported values sl is set to:

$$sl = \{A : 1, 2, 3; \quad B : 1, 2, 3; \quad C : 1, 2, 3\} \tag{26}$$

To find the set of values supported by P_{σ_1} we find the values supported by each tree making it up. This involves two filtering passes over copies of the domain, working first from the root outwards, and then back again.

Taking the left tree first, on the first pass 3 is removed from the domain copy for A and 1 from the domain of B due to $A < B$. So we would log

$$\bar{s}_0 + \bar{a}_{<3} \geq 1; \qquad \bar{s}_0 + \bar{b}_{>1} \geq 1; \tag{27}$$

as these inferences were made due to a binary constraint. No further inferences are made on the second pass. This leaves the values

$$\{A : 1, 2; \quad B : 2, 3\} \tag{28}$$

supported by the first tree, and so these are removed from sl. Filtering the other tree simply reduces the domain copy for C to $\{3\}$, and so $C = 3$ is removed from sl. No inferences need to be logged here as this is a unary constraint. We now have

$$sl = \{A : 3; \quad B : 1; \quad C : 1, 2\} \tag{29}$$

Moving on to the second tuple, no inference occurs on the first pass due to $A = B$ nor $B \geq C$, but 1 is removed from the (fresh) domain copy for A due to the unary $A \neq 1$ entry. Then on the second pass, 1 is now removed from the domain copy for B due to $A = B$, and so we log

$$\bar{s}_1 + \bar{b}_{=1} \geq 1; \tag{30}$$

and then no further inference occurs. So the values occurring in some solution for this tree are

$$\{A : 2, 3; \quad B : 2, 3; \quad C : 1, 2, 3\} \tag{31}$$

and so these can be removed from sl, leaving

$$sl = \{B : 1\}. \tag{32}$$

Ultimately, $B = 1$ is the only assignment lacking support in the table after this initial execution of the domain-consistent propagator and the only value to be removed from an actual variable domain. We are then able to log the three constraints in (18) as discussed above, and thus we have successfully maintained the required invariant for pseudo-Boolean proof logging of this propagator.

4 Proof Logging for Regular Language Membership Constraints

A regular language provides another compact way of representing a constraint extensionally [19]. It is defined as follows.

▶ **Definition 2.** *Regular Language Membership Constraint*
Let $\mathbf{X}$ *again be a sequence of finite-domain variables* $\langle X_1, \dots, X_n \rangle$ *and let* $M = (Q, \Sigma, \delta, q_0, F)$ *denote a deterministic finite automaton (DFA), where*

- Q *is a set of states;*
- $q_0 \in Q$ *is the initial state and* $F \subseteq Q$ *is the set of accepting states;*
- Σ *is a set of symbols having the domain of every variable in* $\mathbf{X}$ *as a subset; and*
- $\delta : (Q \times \Sigma) \to Q$ *is the transition function.*

A regular language membership constraint `Regular`$(\mathbf{X}, M)$ *requires that any sequence of values taken by the variables of* $\mathbf{X}$ *must belong to the regular language recognised by* M*.*

Throughout this section we will assume that the DFA is specified explicitly by a trusted source. Verified compilation to create the constraint from a regular expression is left as future work.

4.1 PB Encodings for Regular Language Membership Constraints

The `Regular` constraint is similarly easy to encode using auxiliary PB variables. If we let $M = (Q, \Sigma, \delta, q_0, F)$ denote a deterministic finite automaton (DFA) and we impose a regular constraint using this on n variables, we simply need to define flags r_{ij} that are set to true if and only if the automaton is in the j_{th} state at position before the $(i+1)_{th}$ symbol of the sequence is processed (including an additional set of flags r_{nj} to denote the final state).

▶ **Encoding 2.** *Regular Language Membership PB Encoding*

```
1: for all $i \in \{0, \dots, n\}$
2:     $r_{i0} + \cdots + r_{i|Q|-1} \geq 1$
3:     $-r_{i0} - \cdots - r_{i|Q|-1} \geq -1$
4: for all $i \in \{0, \cdots n-1\}$, $q \in Q$, $v \in \Sigma$
5:     if $\delta(q, v)$ is an allowed transition
6:         $r_{iq} \wedge x_{i=v} \implies r_{i+1\delta(q,v)} \geq 1$
7:     else
8:         $\bar{r}_{iq} + \bar{x}_{i=v} \geq 1$
9: $r_{00} \geq 1$
10: $\sum_{f \in F} r_{(n)f} \geq 1$
```

This is a simpler version of the stronger SAT encoding for `Regular` given by Bacchus [1].

4.2 Justifying Regular Propagation

As with `SmartTable`, the `Regular` propagator enforces domain consistency, and so our goal once again is to log

$$\wedge\mathcal{G} \implies \bar{x}_{i=v} \geq 1 \tag{33}$$

where $\mathcal{G}$ is the current sequences of guesses made by the solver, and $X_i = v$ is an assignment that is not supported by the regular language membership constraint. In particular, $X_i = v$ is not supported when there exists no sequence of symbols belonging to the language recognised by M that has v as the i_{th} symbol.

It is trivial to establish that this inference will not immediately follow by unit propagation, which is unsurprising given the relative simplicity of the encoding. Just as we did for `SmartTable`, we will have to log some additional facts during the execution of the propagation algorithm.

We will consider here the original domain-consistent `Regular` propagator by Pesant [19], on a constraint over a sequence of n variables $\mathbf{X} = \langle X_1 \ldots, X_n \rangle$ and associated with a DFA $M = (Q, \Sigma, \delta, q_0, F)$. This works by maintaining a layered, directed multigraph that has $n+1$ layers, with the i_{th} layer having $|Q|$ nodes $q_0^i, \ldots q_{|Q|-1}^i$. The edges in this graph are labelled with values in Σ, and are required to respect several properties that are satisfied from the outset and then maintained through propagation. These are:

1. Edges can only appear between nodes in consecutive layers.
2. An edge labelled with the value v can only be included between nodes q_k^i and nodes q_l^{i+1} if there exists a transition between the states numbered k and l in the DFA for a v, with v currently in the domain of X_i.
3. Furthermore, every edge included must appear in a path from the node q_0^0 (representing the initial state) to a node representing a final state in the last layer.

The graph is constructed at the start of the solving process, and then updated incrementally during propagation to reflect changes in the variable domains and preserve the required properties. It is clear then that once this is achieved, an assignment $X_i = v$ loses support on the regular constraint exactly when there are no edges labelled with v between nodes in the i_{th} and $(i+1)_{th}$ layers.

The strategy for proof logging is then as follows. Every time an edge (q_k^i, q_l^{i+1}) labelled with the symbol v is removed by the propagation algorithm (or excluded when the graph is first built), we log the PB constraint

$$\wedge\mathcal{G} \implies \bar{r}_{ik} + \bar{x}_{i=v} \geq 1, \tag{34}$$

which can be interpreted as saying: "given these guesses, we can't both be in state k after we've processed the first $i-1$ symbols, and have the i_{th} symbol be v". If we do this, then when an assignment $X_i = v$ loses support we will have logged a constraint in the form of (34) for all r_{iq} where $\delta(q, v)$ is an allowed transition. Thus, the negation of (33) will result in $\bar{r}_{iq}$ being propagated for all $q \in Q$, and then this will contradict one of the constraints specified by the second line of Encoding 2, which says that at least one r_{iq} must be set to 1. So with these constraints in place we would be able to log our desired constraint (33) by RUP.

The problem is therefore reduced to that of deriving the constraints in the form of (34). It turns out that these will either follow immediately by RUP, or require additional justification depending on how the corresponding edge elimination is inferred by the propagation.

Edges are removed by the `Regular` propagator in one of three ways. Firstly, when an assignment $X_i = v$ has already been removed, either because another value has been guessed or because it has been eliminated by a propagator for another CP constraint, all the edges extending from the i_{th} layer labelled with value v are also removed. In this case the corresponding constraints in (34) can be logged by RUP, as $x_{i=v}$ will immediately contradict the guesses or previous inferences.

Secondly, when a particular node loses all of its outgoing edges, none of its incoming edges can be part of a valid path and so should be removed. These removals occur recursively in a backward pass (Algorithm 3 from Pesant [19]), so if a removed incoming edge was the last outgoing edge of a node in a previous layer then the incoming edges of that previous node are also removed, and so on. In this case too, the constraints in the form of (34) follow by RUP. This can be shown inductively. Suppose all of the outgoing edges for a node q_l^i have been removed and the corresponding constraints in the form of (34) have already been logged. Then for a given incoming edge (q_k^{i-1}, q_l^i) labelled with the value v, the negation of the corresponding constraint in the form of (34) will entail r_{ik}, and since $\delta(q_k, v)$ must be q_l, r_{il} will then unit propagate by one of the constraint specified by line 6 of Encoding 2. Then, since we have already eliminated all possible outgoing edges, we will propagate $\bar{x}_{i=v'}$ for all $v' \in \Sigma$, which will contradict the encoding of the variable X_i, as it must take at least one value.

Finally, in the other direction, when a particular node loses all of its incoming edges, similarly none of its outgoing edges can be part of a valid path and so should be removed. These removals also occur recursively, this time in a forward pass (Algorithm 4 from Pesant[19]). In contrast to the previous two cases, this inference requires more justification as the desired constraint will not always follow by RUP, as is the case for (45) in the worked example below. So when a state q_l^i loses all of its incoming edges we first need to log the constraints

$$\wedge\mathcal{G} \implies \bar{r}_{i-1k} + \bar{r}_{il} \geq 1 \tag{35}$$

for each $k \in \{1, \ldots, k\}$, which intuitively say that no previous state could lead to this state, before logging constraints in the form of (34) for each outgoing edge. These constraints in the form of (35) will all follow by RUP, since for every $v \in \Sigma$ we either have a constraint

$$\wedge\mathcal{G} \implies \bar{r}_{i-1k} + \bar{x}_{i-1=v} \geq 1; \tag{36}$$

from those in the form of (34) logged at the previous stage of the recursion; or there is a constraint

$$\wedge\mathcal{G} \implies \bar{x}_{i-1=v} \geq 1; \tag{37}$$

logged during earlier search/propagation (when this node was the first layer of the recursion); or else there is a constraint

$$\bar{r}_{i-1k} + x_{i-1=v} \geq 1; \tag{38}$$

present in the PB model. In all cases, the negation of (35) will result in $\bar{x}_{i-1=v}$ for every $v \in \Sigma$, giving a contradiction.

Once all the constraints in the form of (35) have been derived, the desired constraint in the form of (34) will definitely follow by RUP, as its negation would then propagate $\bar{r}_{i-1k}$ for each $k \in Q$, contradicting one of the constraints on the second line of Encoding 2.

Putting all of this together, we can emit full justification using RUP for every edge elimination performed by `Regular`, and thus we are able to derive the constraint in the form of (33) required by the proof logging invariant.

It should be noted that the same justifications apply when building the graph for the first time, as it can be conceptually viewed as eliminating edges from the complete layered graph with $n+1$ layers and $|Q|$ nodes in each layer. The deletion occurs in two passes. Firstly it eliminates all outgoing edges from every node in layer 0 that does not correspond to the initial state, recursing for each subsequent layer. Then secondly, it eliminates all incoming edges from every node in layer $n+1$ that does not correspond to a final state, similarly recursing for each previous layer.

Note finally that stronger encodings of the `Regular` constraint are possible, and these would not require as much proof logging during search. However, these encodings are not so obviously correct, which is a drawback since the constraint encoding process is not (currently) verified. It is not even clear that such encodings would give more efficient proof verification, since they would involve having a larger set of active PB constraints for the entire verification process.

4.3 A Complete Worked Example for Regular Propagation

Once again, to demonstrate how the proof logging procedure works more concretely, we will now show a worked example, including the initial building of the graph and a round of domain-consistent propagation. We will take the simple Example 1 from Pesant [19], of a regular language membership constraint on five variables $X_1, \ldots, X_5$ each with domain $\{0, 1, 2\}$, and using the DFA M as shown in Figure 2a.

For the PB encoding, we will omit the constraints encoding the domains and equals flags and focus on the constraints present in Encoding 2. We can see that we have five states numbered $0 \ldots 4$, and five variables in the sequence, so the model would first define:

$$r_{00} + \cdots + r_{04} \geq 1; \qquad \ldots \qquad r_{50} + \cdots + r_{54} \geq 1; \tag{39}$$

$$-r_{00} - \cdots - r_{04} \geq -1; \qquad \ldots \qquad -r_{50} - \cdots - r_{54} \geq -1; \tag{40}$$

saying that we have to have exactly one of the state-position flags set for each position. Next for each position i we would define eight PB constraints corresponding to the eight valid transitions:

$$\begin{aligned}
&\bar{r}_{i0} + \bar{x}_{i=0} + r_{i+11} \geq 1; \quad \bar{r}_{i0} + \bar{x}_{i=2} + r_{i+14} \geq 1; \quad \bar{r}_{i1} + \bar{x}_{i=0} + r_{i+11} \geq 1;\\
&\bar{r}_{i1} + \bar{x}_{i=1} + r_{i+12} \geq 1; \quad \bar{r}_{i2} + \bar{x}_{i=0} + r_{i+13} \geq 1; \quad \bar{r}_{i2} + \bar{x}_{i=1} + r_{i+12} \geq 1;\\
&\bar{r}_{i3} + \bar{x}_{i=0} + r_{i+13} \geq 1; \quad \bar{r}_{i4} + \bar{x}_{i=2} + r_{i+14} \geq 1;
\end{aligned} \tag{41}$$

along with constraints of the form

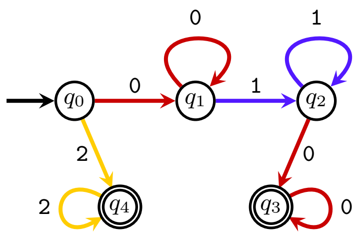
$$\bar{r}_{iq} + \bar{x}_{i=v} \geq 1; \tag{42}$$

for the remaining seventeen invalid transitions. Finally, we would have constraints saying that we have to start in an initial state and end in a final state:

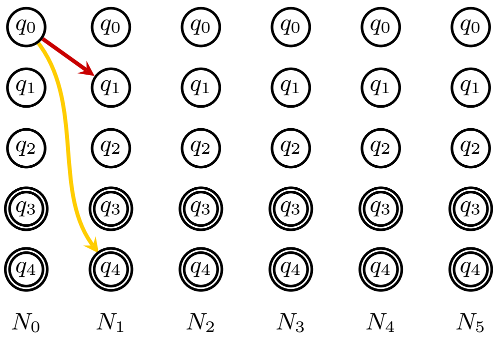
$$r_{00} \geq 1; \qquad r_{53} + r_{54} \geq 1. \tag{43}$$

Propagation depends on us having first built a valid graph, with nodes as shown in Figure 2b. As mentioned, this occurs in two passes, with the forward pass collecting nodes that can be reached from the initial state. This corresponds to removing outgoing edges only reachable from nodes other than the initial state.

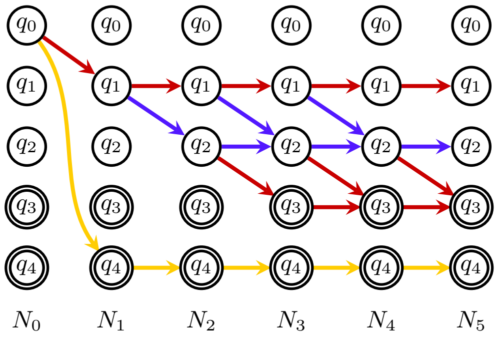
So to begin with, we traverse the two edges incident to node q_0^0, reaching nodes q_1^1 and q_4^1 and thereby eliminating all other edges. These are the two edges shown in Figure 2b. Since we are eliminating outgoing edges we are effectively in the third case of the proof logging



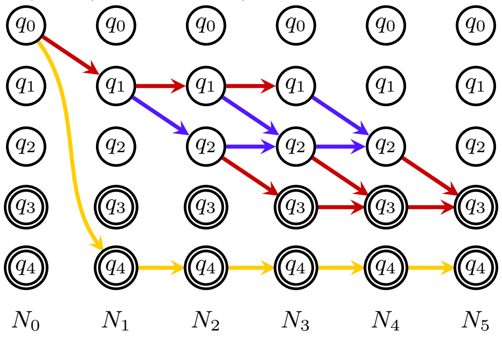
(a) A DFA M recognising the regular expression `00*11*00*+2*`. Double circles indicate accepting states.



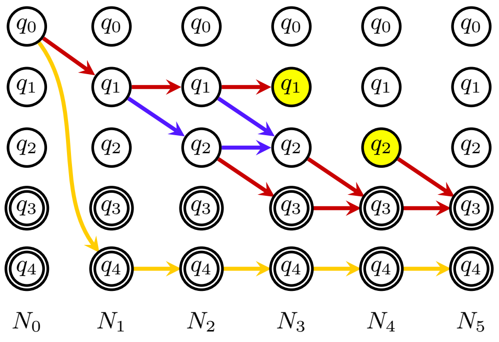
(b) The first traversal in the forward pass for building the layered multigraph used in propagation of `Regular`$(X_1, \ldots, X_5, M)$.



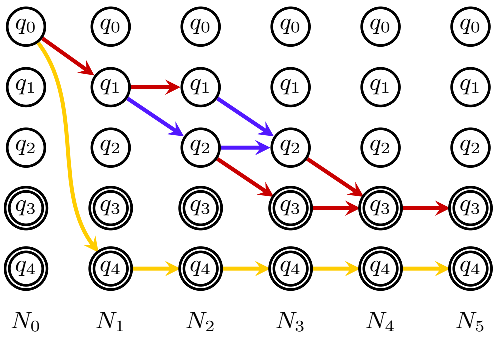
(c) The same multigraph after the forward pass is complete.



(d) The same multigraph after the backwards pass is complete.



(e) After inferring that $X_3 \neq 1$, the multigraph is in an invalid state.



(f) The same multigraph after its consistency properties have been re-established.

Figure 2 Propagation of the regular expression `00*11*00*+2*` on the variables $X_1, \ldots, X_5$, showing the initial state, and then the effect of propagating $X_3 \neq 1$.

procedure from the previous section, however, as there are no previous states to eliminate we simply need to log constraints in the form of (34) for each of the six eliminated edges:

$$\begin{aligned}&\bar{r}_{01} + \bar{x}_{0=0} \geq 1; \quad \bar{r}_{01} + \bar{x}_{0=1} \geq 1; \quad \bar{r}_{02} + \bar{x}_{0=0} \geq 1;\\ &\bar{r}_{02} + \bar{x}_{0=1} \geq 1; \quad \bar{r}_{03} + \bar{x}_{0=0} \geq 1; \quad \bar{r}_{04} + \bar{x}_{0=2} \geq 1;\end{aligned} \tag{44}$$

and these will all follow by RUP because we have $r_{00} \geq 1$ in the PB model. Continuing, we recursively collect edges reachable from the initial node, consequently eliminating all outgoing edges for any node that is not reached. The result of this is shown in Figure 2c.

At each layer, we can follow this same proof logging procedure for eliminating outgoing edges. For example, we do not collect (q_3^2, q_3^3) at layer 2, since q_3^2 is not reached as part of the forward pass. So we would like to log

$$\bar{r}_{23} + \bar{x}_{2=0} \geq 1; \tag{45}$$

for this eliminated edge, but this does not follow by RUP. Despite us having logged constraints in the form of (34) for each eliminated incoming edge to q_2^3, none of these constraints contain either r_{23} or $x_{2=0}$, as they concern the previous layer, and so no further propagation takes place from the negation of (45). Hence, as discussed, we first log

$$\bar{r}_{10} + \bar{r}_{23} \geq 1; \quad \bar{r}_{11} + \bar{r}_{23} \geq 1; \quad \bar{r}_{12} + \bar{r}_{23} \geq 1; \quad \bar{r}_{13} + \bar{r}_{23} \geq 1; \quad \bar{r}_{14} + \bar{r}_{23} \geq 1; \tag{46}$$

which all do follow by RUP, and then our constraint (45) can be logged.

After this process is complete, the backwards pass takes place, first eliminating incoming edges from any nodes in the last layer corresponding to non-final states, and then recursively eliminating all incoming edges from any node that lost all of its outgoing edges. The complete, correctly initialised state of the graph after this backwards pass is shown in Figure 2d.

Following the proof logging procedure for eliminating incoming edges, in this direction we simply need to log a constraint in the form of (34) for each, and these will follow by RUP, as discussed.

At this point, some basic inferences about variable value pairs can already be made, such as $X_0 \neq 1$ and $X_4 \neq 1$. Due to all the constraints that have already been logged, eliminating all possible corresponding edges from consideration, the required proof logging invariant constraint for each will follow by RUP. This concludes the graph initialisation for the `Regular` propagator, and from this point onwards the structure can be updated incrementally and restored upon backtrack, as described in detail by Pesant [19].

We will now demonstrate one such incremental update, corresponding to an execution of domain-consistent propagation for `Regular`. Suppose through the course of the computation, the assignment $X_3 = 1$ loses support. This means the edges (q_1^3, q_2^4) and (q_2^3, q_2^4) are immediately removed from the graph, and we log the RUP constraints:

$$\bar{r}_{13} + \bar{x}_{3=1} \geq 1; \qquad\qquad \bar{r}_{23} + \bar{x}_{3=1} \geq 1. \tag{47}$$

At this point the state of the graph is as shown in Figure 2e, with the two highlighted nodes having lost all of their outgoing and incoming edges respectively.

The recursive incremental update is then executed for each of these, further removing the edge (q_1^2, q_1^3) in a backwards pass, and so logging the RUP constraint

$$\wedge\mathcal{G} \implies \bar{r}_{31} + \bar{x}_{2=0}. \tag{48}$$

The edge (q_2^4, q_3^5) is also removed, but because this is removed in a forward pass, we first log

$$\wedge\mathcal{G} \implies \bar{r}_{30} + \bar{r}_{42} \geq 1; \quad \wedge\mathcal{G} \implies \bar{r}_{31} + \bar{r}_{42} \geq 1; \quad \dots \quad \wedge\mathcal{G} \implies \bar{r}_{34} + \bar{r}_{42} \geq 1; \qquad (49)$$

before logging the required

$$\wedge\mathcal{G} \implies \bar{r}_{42} + \bar{x}_{2=0}. \qquad (50)$$

No further nodes lose their last incoming or outgoing edge as a result of this, and so the required properties (and hence domain consistency) have been re-established on the graph. The final consistent state is shown in Figure 2f.

5 Implementation and Validation

We have implemented both of these proof logging propagation algorithms inside the open-source *Glasgow Constraint Solver* [17]. Our implementation adds two new constraints `Regular` and `SmartTable`, with specific functionality to produce the required encodings and justifying propagators. They can therefore be used in conjunction with the rest of the constraint types already available in the solver.

We validated the implementations by first modelling some key examples. For `SmartTable`, these included the representation of lexicographic ordering problems, and problems where at most one variable takes a value, as given by Mairy et al. [16], as well as the illustrative examples from Section 3. For `Regular`, we implemented both examples 1 and 2 from Pesant [19].

We carried out further experimental validation by generating random (acyclic) smart tables, and random DFAs on sequences of up to five variables and solving the corresponding single-constraint problems. In all tests and examples, the solver produced an OPB model and proof files, and we checked these using the VeriPB proof checker. We also found that although proof logging incurs an obvious performance cost, the observed overheads were not unreasonable, giving a slowdown factor of between 2 and 10. As proofs are written currently written directly to disk, this can be very hardware dependent, and also dependent on how optimised the propagator implementation is. We decided to leave further engineering and optimisation of proof writing to future work.

During development some very subtle bugs in both propagator implementations that had eluded conventional testing were caught immediately by proof logging. For example, in an incremental version of the `Regular` propagator, an unsound inference was being made due to mixing up variable names, but in such a way that a situation where this unsound inference would actually lead to an incorrect solution was extremely rare (only in specially constructed instances, created once we were made aware of the bug due to proof logging). After correcting bugs such as these, all proofs were certified by VeriPB as being correct.

6 Conclusion

We have shown that we can efficiently justify all the reasoning that could possibly be carried out for two families of smart extensional constraints. An interesting observation is that justifying this reasoning required only the RUP rule, and this rule was only to provide hints of algorithmic steps that were already being carried out by the propagators. In effect, we are logging a sequence of "lookahead to see immediate contradiction" steps. We did not require any explicit cutting planes derivations, and although we relied upon pseudo-Boolean constraints to make it simple to express reifications and negations, in principle everything we did should also be possible in a weaker proof encoding and proof system such as CNF and DRAT. The only caveat is that we *do* rely upon strong propagation properties for encoded integer variables, which would limit approaches based upon Boolean satisfiability to integer variables with very small domains. This is in contrast to constraints like `AllDifferent` and `Linear`, which cannot be logged efficiently in resolution-based approaches [8, 10].

We expect that other global constraints, even those with complex propagation algorithms will be similarly feasible to proof log using this technique. Richer smart tables, such as those with offset or ternary restrictions [4], also fit into the framework given in Section 3.2, although the justification of the filtering inferences for each restriction may require additional cutting planes steps. Furthermore, the propagation of "Multi-Valued Decision Diagram"-based constraints [14] can be viewed as a generalisation of the techniques used for the regular language membership constraint, and so it seems very plausible that a similar proof logging methodology as demonstrated in this paper could work here. This bodes well for being able to provide auditable solving for most global constraints that might occur in a modern constraint solver.

References

1 Fahiem Bacchus. GAC Via Unit Propagation. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, Lecture Notes in Computer Science, pages 133–147, Berlin, Heidelberg, 2007. Springer. `doi:10.1007/978-3-540-74970-7_12`.

2 Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part I*, pages 59–75, Berlin, Heidelberg, 2021. Springer-Verlag. `doi:10.1007/978-3-030-72016-2_4`.

3 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified Symmetry and Dominance Breaking for Combinatorial Optimisation, March 2022. `doi:10.48550/arXiv.2203.12275`.

4 Frédéric Boussemart, Christophe Lecoutre, and Cédric Piette. XCSP3: an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016. `arXiv:1611.03398`.

5 Chiu Wo Choi, Warwick Harvey, J. H. M. Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In *AI 2006: Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006, Proceedings*, pages 49–58, 2006. `doi:10.1007/11941439_9`.

6 W. Cook, C.R. Coullard, and Gy. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987. `doi:10.1016/0166-218X(87)90039-4`.

7 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 220–236, Cham, 2017. Springer International Publishing. `doi:10.1007/978-3-319-63046-5_14`.

8 Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying All Differences Using Pseudo-Boolean Reasoning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02):1486–1494, April 2020. `doi:10.1609/aaai.v34i02.5507`.

9 Stephan Gocht. *Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, Sweden, 2022. URL: `https://lup.lub.lu.se/record/3550cb96-83d5-4fc7-9e62-190083a3c10a`.

10 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An Auditable Constraint Programming Solver. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.CP.2022.25`.

11 E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 886–891, 2003. `doi:10.1109/DATE.2003.1253718`.

12 Marijn J. H. Heule, Warren A. Hunt, and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, pages 345–359, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. `doi:10.1007/978-3-642-38574-2_24`.

13 Marijn J.H. Heule, Warren A. Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *2013 Formal Methods in Computer-Aided Design*, pages 181–188, 2013. `doi:10.1109/FMCAD.2013.6679408`.

14 Samid Hoda, Willem-Jan van Hoeve, and J. N. Hooker. A systematic approach to MDD-Based constraint programming. In David Cohen, editor, *Principles and Practice of Constraint Programming – CP 2010*, pages 266–280, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. `doi:10.1007/978-3-642-15396-9_23`.

15 Mikael Z Lagerkvist and Gilles Pesant. Modeling irregular shape placement problems with regular constraints. In *First workshop on bin packing and placement constraints BPPC'08*, 2008.

16 Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. The Smart Table Constraint. In Laurent Michel, editor, *12th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2015)*, Lecture Notes in Computer Science, pages 271–287, Cham, 2015. Springer International Publishing. `doi:10.1007/978-3-319-18008-3_19`.

17 Ciaran McCreesh and Matthew McIlree. The Glasgow Constraint Solver. GitHub repository, 2023. URL: `https://github.com/ciaranm/glasgow-constraint-solver`.

18 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007. `doi:10.1007/978-3-540-74970-7_39`.

19 Gilles Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In Mark Wallace, editor, *10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, Lecture Notes in Computer Science, pages 482–495, Berlin, Heidelberg, 2004. Springer. `doi:10.1007/978-3-540-30201-8_36`.

20 Hélène Verhaeghe. *The extensional constraint.* PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2021.

21 Hélène Verhaeghe, Christophe Lecoutre, Yves Deville, and Pierre Schaus. Extending Compact-Table to Basic Smart Tables. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming*, volume 10416, pages 297–307. Springer International Publishing, Cham, 2017. `doi:10.1007/978-3-319-66158-2_19`.