An Auditable Constraint Programming Solver

Stephan Gocht
Lund University, Sweden
University of Copenhagen, Denmark

Ciaran McCreesh
University of Glasgow, UK

Jakob Nordström
University of Copenhagen, Denmark
Lund University, Sweden

Abstract

We describe the design and implementation of a new constraint programming solver that can produce an auditable record of what problem was solved and how the solution was reached. As well as a solution, this solver provides an independently verifiable proof log demonstrating that the solution is correct. This proof log uses the VeriPB proof system, which is based upon cutting planes reasoning with extension variables. We explain how this system can support global constraints, variables with large domains, and reformulation, despite not natively understanding any of these concepts.

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

Keywords and phrases Constraint programming, proof logging, auditable solving

Digital Object Identifier 10.4230/LIPIcs.CP.2022.25

Supplementary Material Source code for the solver described in this paper can be found here:
Software (Source Code): `https://doi.org/10.5281/zenodo.6514093`

Funding *Stephan Gocht*: Supported by the Swedish Research Council grant 2016-00782.
Ciaran McCreesh: Supported by a Royal Academy of Engineering research fellowship.
Jakob Nordström: Supported by the Swedish Research Council grant 2016-00782 and the Independent Research Fund Denmark grant 9040-00389B.

1 Why Trust a Constraint Programming Solver?

Proof logging is now a standard practice in the Boolean satisfiability (SAT) community: alongside a solution, solvers are expected to produce a proof, in a standard format called DRAT [19, 18, 33], that can be verified independently to ensure that the correct answer was reached through legitimate reasoning. As well as reducing the number of bugs in solvers, this has been vital for the social acceptability of computer-generated mathematical proofs [21]. These successes mean that proof logging is now being considered in other areas, including mixed integer programming [9] and subgraph-finding algorithms [14, 13], and a similar paradigm known as *certifying algorithms* exists for polynomial-time solvable problems [23].

We believe that a practical proof logging system would also be extremely useful for the constraint programming (CP) community. In the 2021 MiniZinc challenge, at least 45 out of 3,500 claimed solutions were incorrect (either through falsely claiming unsatisfiability or optimality, or by providing infeasible "solutions"), and previous years saw similar rates. Furthermore, this was not limited to one solver, one problem, or one global constraint. Although this high error rate does not necessarily reflect what we might see in practice, it strongly suggests that we should not be complacent. And even if we are completely convinced that our solvers are correct, thanks to extensive testing using domain-specific methods [1, 12],

there are still benefits to be had from proof logging. When CP is used for life-affecting decision-making, having a solver that can produce an independently verifiable record of what the problem was and how it was solved would be much better for public confidence in algorithms than saying "trust us, we tested it carefully". In effect, we would be making the solving process *auditable*, and removing the need for trust.

In some applications, compiling a CP model to SAT and re-using SAT proof logging might be a viable approach for auditability. However, this is not a universal solution: even if the loss of solving power from switching representations is not a problem, why should we trust that a complex compilation process is correct? And what if we need to solve enumeration or optimisation problems, neither of which are supported by DRAT? Nor is it practical to make CP solvers output DRAT proofs, even for decision problems: attempts at expressing the strong reasoning carried out by simple global constraints like all-different have introduced intolerable overheads [28, 10], and DRAT does not seem well-suited even for the parity reasoning done by some modern SAT solvers [15]. One alternative would be to introduce a much stronger and more complex proof format, that is aware of the meaning of every global constraint and every kind of propagation that could be performed [31] and every kind of reformulation ever invented – but why should we trust such a complicated proof logging system, and how would we even know that it is consistent? This is not a trivial concern: even the relatively simple DRAT format has had issues in this respect [26].

This paper describes an alternative approach to proof logging which addresses all of these problems. It uses an existing proof verifier, VeriPB, which was designed for pseudo-Boolean models. VeriPB's proof format uses cutting planes reasoning [5] and redundance-based strengthening [15], which is only a small step up in complexity from the DRAT approach of using Boolean models and extended resolution. However, this small change in underlying proof system suddenly means that proof logging for many kinds of constraint becomes both efficient and easy to implement, despite the system not having any explicit notion of global constraints or even non-binary variables. Thanks to this, we have been able to implement a new prototype constraint programming solver which can produce proof logs for all of its reasoning, with support for global constraints like all-different, integer linear inequality (including for variables with very large domains), table, minimum / maximum of an array, element, and absolute value, as well as some simple automatic reformulation.

Our aim in this work is not to produce the world's fastest solver, but rather to explore the design decisions necessary to provide auditable solving when operating with diverse constraints, and to explain how to understand and adopt proof logging technology. The main differences between our solver and a basic conventional solver such as MiniCP [24] are:

- The solver can describe the semantics of variables and each constraint in a low-level format, which is discussed in Section 2.1. We give examples in Section 3. This is the only part of the process that is not directly auditable: we discuss this further in Section 5.1.
- Whenever the solver backtracks during search, it creates a proof step asserting that its current sequence of guesses "obviously" (but verifiably) implies a contradiction, as described in Section 2.3. When enumerating or optimising, solutions must also be logged.
- Any piece of code that potentially removes a value from a variable's domain (or instantiates it, or changes its bounds) must be able to provide a justification that can be added to the proof log. This justification can be "this is immediately obvious", "use reverse unit propagation" (Section 2.3), or occasionally, "use the following explicit proof steps" (Section 2.2). In many ways this resembles lazy clause generation solvers [25], except that justifications must be derived in a sound and verifiable manner, rather than being introduced from nowhere. We give examples in Section 4.
- Finally, some constraints make use of reformulations, which must also be justified; examples are in Section 4.5.

Together, these additions mean that if the solver ever produces an incorrect answer, this can be detected – even if it is due to a compiler or hardware fault rather than a solver bug. Our results demonstrate that proof logging for constraint programming is rapidly becoming a technologically viable approach, and one that will be worth adopting in other solvers. We conclude by outlining how we might realise this goal of auditable constraint solving.

2 The VeriPB Proof System

We begin with an overview of the relevant parts of the model and proof system used by VeriPB.[1] It is important to stress immediately that solvers do not need to understand or implement this proof system (in the same way that most SAT solvers do not "know" that they are searching for resolution proofs). Indeed, the prototype solver we describe later in this paper produces proofs through templates, never manipulates proof steps directly, and does not know enough about the VeriPB proof system to be able to verify its own proofs.

In this section we will primarily be talking about proofs of unsatisfiability. Both VeriPB and our solver also support optimisation, enumeration, and satisfiable decision problems, but the core of the proof system concerns unsatisfiability. The idea behind a proof is that we start off with known facts, which come from the input model. Then, at each step in the proof, we derive a new fact which is "obviously" a consequence of some combination of previous facts. We finish by deriving a fact which is clearly a contradiction, which in turn means it must be the case that the input is unsatisfiable.

2.1 Pseudo-Boolean Models

VeriPB takes as input a pseudo-Boolean model, which is a very restricted kind of constraint programming model. A pseudo-Boolean model is defined by a set $\{x_i\}$ of $\{0,1\}$ integer variables, and a set of integer linear inequalities $\sum_i c_i x_i \geq n$ for integers c_i and n. In this paper we will use lowercase variable names to refer to pseudo-Boolean variables, and uppercase variable names to refer to constraint programming variables. We will also write some constraints using $\leq$ instead of $\geq$, and will write $\sum_i c_i x_i = n$ as shorthand for two inequalities. We use the convention that $x = 0$ means false, and $x = 1$ means true; we write $\overline{x}$ to mean $1 - x$. Observe that Boolean satisfiability constraints in conjunctive normal form (CNF) can easily be written as pseudo-Boolean constraints, because e.g. $(x_1 \vee \overline{x}_2 \vee x_3)$ holds if and only if $(x_1 + \overline{x}_2 + x_3 \geq 1)$. For clarity we will sometimes mix logical and pseudo-Boolean notation, and write expressions like $(x_1 \wedge x_2 \wedge x_3) \rightarrow (2x_4 + 3x_5 + -4x_6 \geq 7)$ rather than the more cumbersome $11\overline{x}_1 + 11\overline{x}_2 + 11\overline{x}_3 + 2x_4 + 3x_5 + -4x_6 \geq 7$.

There is a standard textual file format for pseudo-Boolean models, known as OPB [27]. VeriPB supports this format, with extensions: for example, it allows variables to have descriptive names, which is convenient for readability, and can include implications to avoid the need for solver authors to calculate appropriate coefficients manually.

2.2 Cutting Planes

Alongside an OPB file, VeriPB takes a proof log file that claims to show that the pseudo-Boolean model is unsatisfiable, and checks the proof's validity. This proof log is a text file, which describes a sequence of steps using the cutting planes proof system [5]. In cutting

[1] `https://gitlab.com/MIAOresearch/VeriPB`

planes, we can add two constraints together, multiply a constraint by a non-negative integer constant, and divide existing constraints by a positive integer constant (with rounding); we may also assert that any literal is non-negative. The aim is to derive a constraint saying that $0 \geq 1$, which serves as a contradiction. The cutting planes proof system is complete for pseudo-Boolean models, in the same way that resolution is complete for Boolean models. However, it is exponentially stronger than resolution: for example, resolution requires exponential length proofs for all-different constraints, whereas cutting planes can justify Hall set reasoning in (small) polynomial length [10]. For more details on the theoretical background, see, e.g., the survey by Buss and Nordström [3].

2.3 Unit Propagation and Reverse Unit Propagation

For solver authors, working directly with cutting planes can be difficult, and would require every part of a solver to keep careful track of every operation carried out. This difficulty can be avoided through the use of *reverse unit propagation* (RUP) proof steps [16, 30, 10], which are in effect shorthand for a sequence of cutting planes steps.

For CNF clauses, *unit propagation* means identifying any clause where all but one of its literals has already been set the wrong way, and propagating the remaining literal to the value that avoids violating the clause, repeating until either a contradiction is reached or no further unit clauses exist. This notion generalises to pseudo-Boolean constraints, where unit propagation means achieving integer bounds consistency [4]. A constraint C is said to be RUP if asserting its negation leads to a contradiction via unit propagation; in such a case, it is obviously permissible to introduce C as a new constraint without altering whether the underlying model is satisfiable.

RUP steps in a Boolean setting form the core part of the DRAT proof format. This is useful for solver authors because for a typical CDCL SAT solver, every learned clause is RUP, and so writing a proof log requires only that a solver output every clause it learns in turn. In our constraint programming setting, RUP clauses will similarly form the backbone of the proofs we generate, with a RUP clause being written every time a solver backtracks. However, we will also use explicit cutting planes steps where necessary, to justify complex propagations. In one sense, RUP is purely a convenience for solver authors, in that with more work, cutting planes steps could be used instead; however, this would require substantially more book-keeping in the solver.

The following pieces of intuition may be helpful in what follows: a fact follows from unit propagation if it is so immediately obvious that it is not worth stating. A fact follows from reverse unit propagation if, once you have been told that it is a fact, it is obviously true (but that it might not be immediately obvious if you are not told). In some ways this resembles failed literal probing, or the difference between generalised arc consistency and singleton arc consistency; this intuition may become clearer following the example in Section 4.1.

2.4 Extension Variables and Redundance-Based Strengthening

An *extension variable* is a new variable introduced as part of a proof. In VeriPB, extension variables are supported using a rule called *redundance-based strengthening* (which, for readers familiar with SAT proof logging, is the natural analogue of the RAT rule in DRAT) [15]. We do not need the full power or definition of that rule for this paper. It suffices to say that, for an arbitrary pseudo-Boolean constraint C and a new variable y that has not previously appeared in the model or proof, we are allowed to introduce the reified constraints $y \leftrightarrow C$ at any point during the proof. As well as being extremely convenient for solver authors, extension variables also give an exponential increase in reasoning power [3].

2.5 Satisfiable Instances, Enumeration, Optimisation, and Deletions

For satisfiable decision problems, VeriPB supports solution checking by including a solution in a proof log. Enumeration problems may also be verified this way: whenever a solution is logged, VeriPB treats this as introducing a new constraint saying "but not this solution", and so a proof is effectively a proof by contradiction that there are no solutions other than the ones listed. Optimisation is handled similarly, via an optional objective expression in the OPB file. Finally, in practice it is important to delete constraints from the proof that will not be re-used later on. This is straightforward, but will not be discussed in this paper.

3 Encoding Constraint Programming Models

In the previous section, we learned that if we wish to use VeriPB to verify constraint programming proofs then we must provide two things: a pseudo-Boolean model in OPB format, and a proof log. We now discuss how the first of these two steps may be generated by a CP solver, looking first at how we turn CP variables into pseudo-Boolean variables, and then at how we represent constraints. When compiling CP to a lower level format for solving, selecting a good encoding involves considering propagation and consistency; in contrast, for proof logging we need only something that is simple and reasonably compact.

3.1 Variables

The most straightforward way of encoding an integer variable X with domain $\ell \dots u$ is to create $u - \ell + 1$ pseudo-Boolean variables $x_{=i}$, where $x_{=i}$ is true if and only if $X = i$, together with supporting constraints saying that $\sum x_{=i} = 1$. Such an approach was used for proof logging the all-different constraint by Elffers et al. [10]. However, this is impractical for variables with large domains that are only involved in bounds-consistent constraints such as integer linear inequalities. Instead, we define a binary encoding. Let h be the least strictly positive number such that $2^{h-1} \geq \max(1, |u|, |\ell|)$. Then we introduce variables $x_{\mathrm{b}i}$ for $i \in 0 \dots h-1$, and, if $\ell < 0$, we additionally introduce an x_{neg} variable to give us a two's complement style representation. The two constraints $\ell \leq -2^h x_{\mathrm{neg}} + \sum_{i=0}^{h-1} 2^i x_{\mathrm{b}i} \leq u$ then define the meaning and bounds of these variables (with the leading sum term omitted if $\ell \geq 0$). For example, if we have a constraint programming variable A with domain $\{-3 \dots 9\}$, we would define

$$-32a_{\mathrm{neg}} + 1a_{\mathrm{b}0} + 2a_{\mathrm{b}1} + 4a_{\mathrm{b}2} + 8a_{\mathrm{b}3} + 16a_{\mathrm{b}4} \geq -3 \text{ and}$$
$$32a_{\mathrm{neg}} + -1a_{\mathrm{b}0} + -2a_{\mathrm{b}1} + -4a_{\mathrm{b}2} + -8a_{\mathrm{b}3} + -16a_{\mathrm{b}4} \geq -9.$$

Although compact, experienced modellers know that such an encoding often leads to extremely poor propagation. This is a problem if the encoding is to be used for solving, but for proof logging this is not an issue because the encoding only restricts how we write a proof, not how a solution is reached. However, when expressing constraints or propagation, it is often useful to be able to use variables $x_{=i}$ and $x_{\geq i}$ for selected values of i. If these variables are used when the constraints appear in the pseudo-Boolean model, we can define them immediately. We have found the most convenient way of expressing these variables to be

$$x_{\geq i} \leftrightarrow -2^h x_{\mathrm{neg}} + \sum_{i=0}^{h-1} x_{\mathrm{b}i} \geq i$$

and similarly for $x_{\geq i+1}$. Additionally, we constrain that $x_{\geq i+1} \rightarrow x_{\geq i}$, and force $x_{\geq i}$ to be true or false respectively if i defines a lower or upper bound. We then define $x_{=i} \leftrightarrow x_{\geq i} \wedge \overline{x}_{\geq i+1}$.

However, what if these values are only used for branching or propagation, such as when dealing with linear inequalities (discussed below)? The whole point of using a binary encoding was to avoid having to define variables for values that never appear in a constraint or a proof. Fortunately, it is possible to introduce these additional variables as extension variables with the same defining constraints, so long as it is done in exactly the order described above. In such a case, we also introduce the RUP constraints $x_{\geq i} \rightarrow x_{\geq j}$ and $x_{\geq h} \rightarrow x_{\geq i}$ for the closest two values h and j that already have equality variables, if they exist. Finally, when propagating certain constraints such as all-different, it is also convenient to have an at-least-one constraint $\sum_{i=\ell}^{u} x_{=i} \geq 1$. If all the $x_{=i}$ variables are defined, then this constraint can also be introduced via RUP as needed, and does not need to be defined in the model. This can make the pseudo-Boolean model much more manageable: for example, for the implementation of the "cake" problem discussed in Section 5, our solver introduces a total of one hundred $x_{=i}$ or $x_{\geq i}$ variables in the proof, rather than defining several hundred thousand of them in the OPB file.

Note finally that the details of this encoding are largely irrelevant to most constraints. In particular, it is possible for the part of a solver that deals with proof logging to treat 0/1 variables separately with almost no impact on the rest of its code.

3.2 Constraints

Next, we must represent every constraint in pseudo-Boolean form. This topic is relatively well-understood, because pseudo-Boolean constraints are a superset of CNF – and again, it is not necessary to find a *good* encoding, only a simple and correct one. We now give some examples that illustrate general concepts.

Integer linear inequalities. Integer linear inequalities can easily be expressed in pseudo-Boolean form by adding multiples of the bit encodings together. For example, suppose we have the CP constraint $2A+3B+4C \leq 42$, where each of the variables has domain $\{-3 \dots 9\}$. This would be translated into

$$\begin{aligned} &-64a_{\text{neg}} + 2a_{\text{b0}} + 4a_{\text{b1}} + 8a_{\text{b2}} + 16a_{\text{b3}} + 32a_{\text{b4}} \\ &+ -96b_{\text{neg}} + 3b_{\text{b0}} + 6b_{\text{b1}} + 12b_{\text{b2}} + 24b_{\text{b3}} + 48b_{\text{b4}} \\ &+ -128c_{\text{neg}} + 4c_{\text{b0}} + 8c_{\text{b1}} + 16c_{\text{b2}} + 32c_{\text{b3}} + 64c_{\text{b4}} >= 42. \end{aligned}$$

We may use a pair of such constraints to define equality and sum constraints. If we were solving using these constraints, we would get very weak propagation, but we will explain why this does not matter in the following section.

Not equals. Not equals constraints can be expressed using two *half-reified* linear inequalities: we introduce a Boolean flag f, and define the constraints $f \rightarrow (A-B>0)$ and $\overline{f} \rightarrow (B-A>0)$. These can be expressed in pseudo-Boolean form as integer linear inequalities with the addition of a suitably large coefficient on the negation of the flag to handle the implication. A similar encoding can be used for the absolute value constraint.

All-different. All-different can be expressed by a set of at-most-one constraints, such as $a_{=2} + b_{=2} + c_{=2} \leq 1$, or by a clique of not-equals constraints. Again, solving using either kind of constraint would give weaker propagation than the usual GAC all-different constraint, but this is not a concern for proof logging.

Table constraints. Table constraints can be expressed in terms of an auxiliary variable, which selects which tuple is matched. For example, given the tuple sequence $[(1,2,3),(1,3,4),(2,2,5)]$ applied as a table constraint to the variables $[A,B,C]$, we could express this by adding an auxiliary variable $T \in \{0 \dots 2\}$ (called the *tuple selector variable*), and using implication constraints $t_{=0} \rightarrow (a_{=1} \wedge b_{=2} \wedge c_{=3})$ etc.

Element constraints. Element constraints come in a variety of forms. For example, consider a 2D element from constants constraint, which says that a variable V takes the $[I,J]$th entry in a two dimensional $m \times n$ array A of constants. This shows up in various places, such as the MiniCP quadratic assignment problem benchmark discussed in Section 5 (where solvers are expected to achieve generalised arc consistency on the two index variables, and bounds consistency on the assigned variable) [24]. The only constraints we will define are the unary constraints $i_{\geq 0}$, $\overline{i}_{\geq m}$, $j_{\geq 0}$ and $\overline{j}_{\geq n}$, and then $(i_{=x} \wedge j_{=y}) \rightarrow v_{=A[x,y]}$ for each array entry. Such constraints, on their own, obviously do not enforce the desired consistency levels, but they have the advantage of being simple. This technique also generalises. For example, if the array A is not constant then the implication constraints can become half-reified equalities instead – and this in turn makes it easy to define array minimum and array maximum constraints.

Other constraints. Other constraints are usually similarly easy to express. The critical point is that encodings need only be correct, not good, and so if we know how to express the constraint at all in CNF or as integer linear inequalities then that is sufficient. Similarly, if a constraint easily fits in a table, then it can be handled that way. Combined with the ability to use auxiliary variables, even constraints like "forms a connected subgraph" are manageable [13].

4 Proofs for Search and Propagation

The core of a proof for an unsatisfiable constraint satisfaction problem is a description of the solver's search tree. This is expressed as a RUP statement for every backtrack, and ends with a contradiction when we backtrack from the root node. The idea is that whenever the constraint solver backtracks, it should be "obvious" that the sequence of guesses made leads to a dead end, and is thus a RUP clause. Gocht et al. [13] provide a worked example of this process in a branch and bound setting for a clique algorithm.

In order to make this process work with global constraints, we will need to include additional proof statements to justify non-obvious propagations (in the same way that Gocht et al. had to justify the clique algorithm's bounds). The core invariant we use is that at every backtrack, any variable-value deletion that is known to the CP solver (and thus part of the decision to backtrack) must be visible to the proof verifier either through unit propagation, or through reverse unit propagation of the backtrack clause. This section elaborates on what this means for various global constraints.

4.1 RUP Justifications and Table Constraints

Achieving generalised arc consistency for table constraints involves two kinds of inference: detecting when a tuple becomes infeasible, and detecting when a variable's value is no longer supported by any feasible tuple. There are several different propagation algorithms for performing this inference [29, 22, 7, 20, 32], but from a proof logging perspective it does not matter how the inference is performed, only what is inferred.

A tuple becoming infeasible requires no justification. Recall that tuples are defined with constraints like $t_{=0} \rightarrow (a_{=1} \wedge b_{=2} \wedge c_{=3})$. If, for example, A loses the value 1 from its domain, this constraint will unit propagate, setting $t_{=0}$ to false. This also holds when assignments are guessed: by the core invariant, asserting through RUP that the guessed assignments imply false will propagate the value loss, which in turn propagates the tuple becoming infeasible.

In contrast, suppose we have only two tuples supporting $A = 1$, and these are both made infeasible by other variables, so a solver infers $A \neq 1$. Let $\mathcal{G}$ be the set of equality variables corresponding to our current set of guessed assignments (for example, $\{b_{=2}, c_{=5}\}$). Then the assignment $\overline{a}_{=1}$ does *not* follow by unit propagation in the proof under the assertion of $\wedge\mathcal{G}$, which means it must be justified in some way. Fortunately, this is very simple, and we need only give a small hint to the proof verifier: we claim that $\wedge\mathcal{G} \rightarrow \overline{a}_{=1}$ will be a RUP constraint. Indeed, its negation is $(\wedge\mathcal{G}) \wedge a_{=1}$. Now consider each tuple t supporting $a_{=1}$ in turn. There must be some constraint derived that, under $\wedge\mathcal{G}$, falsifies a different variable in this tuple, which in turn forces the tuple selector variable to not equal t. Additionally, we know that A must take at most one value, and so for each $i \neq 1$, $a_{=i}$ will propagate to false; this in turn propagates every other tuple selector variable to false. Finally, we know that the tuple selector variable must take at least one value, but we have ruled out every value it could take, giving the desired contradiction.

Putting these facts together, the only proof logging needed for a table constraint is to log one RUP step whenever a variable loses a value due to lack of support. Intuitively, infeasibility of tuples is so obvious that we need not mention it. In contrast, loss of support is not immediately obvious to detect, but if we tell the proof verifier that it has in fact occurred then it is easy to check that we are telling the truth.

4.2 Explicit Justifications and Integer Linear Inequalities

Some constraints require more work. Elffers et al. [10] have already shown how both propagation and failure detection for the all-different constraint can be justified using cutting planes proofs. Their approach works in our setting, with one caveat: they require at-least-one constraints for certain CP variables, which we do not have in our model. However, recall that these constraints can be introduced using RUP where needed.

Integer linear inequalities are a similar case. Suppose we have a constraint $2A+3B+4C \leq 42$, with all three variables non-negative. In a typical CP solver, this constraint will achieve integer bounds consistency [17, 4]. As an example, suppose we know that under some set of guessed assignments $\mathcal{G}$ that $A \geq 5$ and $C \geq 3$, then a CP solver will infer that $\wedge\mathcal{G} \rightarrow B \leq 6$. We can derive this fact in a proof as follows. By assumption, we either have or can introduce RUP constraints that $\wedge\mathcal{G} \rightarrow a_{\geq 5}$ and $\wedge\mathcal{G} \rightarrow c_{\geq 3}$. This in turn means we either have or can introduce RUP constraints for the binary representation, saying that $\wedge\mathcal{G} \rightarrow \sum_{i=0}^{h-1} 2^i a_{\mathrm{b}i} \geq 5$ and $\wedge\mathcal{G} \rightarrow \sum_{i=0}^{h'-1} 2^i c_{\mathrm{b}i} \geq 3$ for appropriate values of h and h'. Now using cutting planes steps, we can multiply the first of these by 2 and the second by 4 (their coefficients in the original linear inequality), add both to the constraint defining the linear inequality, and divide the result by the coefficient of B, 3. It can be verified that the resulting mess is now sufficient to make $\wedge\mathcal{G} \rightarrow \overline{b}_{\geq 7}$ a RUP constraint. It is also routine to prove that this example generalises to arbitrary integer linear inequalities (although negative variables and / or coefficients require several awkward case by case analyses).

4.3 Element Constraints

Recall the special case of a two-dimensional element from constants constraint, where a variable V takes the $[I, J]$th entry in a two dimensional $m \times n$ array A of constants. In the interests of having a simple pseudo-Boolean encoding, we defined this using $(i_{=x} \wedge j_{=y}) \rightarrow v_{=A[x,y]}$ constraints. However, we wish for our solver to achieve generalised arc consistency on I and J, and bounds consistency on V. One way to proof log this reasoning is as follows. As a one-time operation at the start of search, we will use extension variables to turn this into a one-dimensional element constraint. We will introduce $m \times n$ new variables s_k, each of which is true if and only if a different $i_{=x} \wedge j_{=y}$ holds. We will then build an at-least-one constraint over the s_k variables via a sequence of $O(m \times n)$ RUP steps, as follows. For each value x for the first index variable I, we are going to derive via RUP that $\sum_k s_k + \overline{i}_{=x} \geq 1$. For this to hold, we first derive via RUP that for each value y for the second index variable J, that $\sum_k s_k + \overline{i}_{=x} + \overline{j}_{=y} \geq 1$. The desired at-least-one constraint now follows via RUP; in effect, we have performed an exhaustive backtracking search over the pair of variables I and J under the assertion that the desired at-least-one constraint does not hold, and shown that no solution satisfying I and J exists. From this point forwards, we are effectively dealing with a one-dimensional element constraint.

(Of course, one could ask why we convert from two dimensions to one dimension in the proof, and not in the model when we define the element constraint. We could certainly do things this way. However, our point here is to demonstrate that *we don't have to* handle model reformulations by changing the model: instead, we can use the most straightforward low-level model imaginable, and then prove that our reformulations are valid as part of the proof. We will explore this further below.)

We can also view our new encoding as being like a table constraint, but where the tuple selector variable is channeled to the two index variables. If we wished to achieve generalised arc consistency on the assigned variable V, we would simply reuse the inference techniques discussed in Section 4.1. However, this would require introducing a pseudo-Boolean equality variable $v_{=n}$ for each value in V's domain. This is potentially not what is desired, if the range of the constants is large and V is only otherwise used in a bounds-consistent manner. Therefore, instead of justifying that V does not take each value no longer present in feasible parts of A in turn, we would like to assert a bounds change using a $v_{\geq n}$ variable. This does not immediately follow by RUP on its own, although it will if we repeat the iteration technique used in creating the index variable, but iterating only over feasible array entries.

A downside to this approach is that it produces a proof containing $O(m \times n)$ steps to justify each bounds propagation. As Michel et al. [24] explain, by storing the array in a sorted manner, it is possible for a propagator to avoid looking at most array entries most of the time, and so have better than $O(m \times n)$ performance in the typical case. We suspect that this algorithm can be replicated in a proof efficiently, if we are prepared to establish a set of ordering constraints at the start of the proof.

Finally, an observant reader might have noticed that deletions on the one-dimensional array index will not unit propagate backwards to I and J. In fact, these deletions are RUP.

4.4 Not Equals

At this point, it should be clear how the not equals constraint can be handled: when one variable A is instantiated to a value v, it follows using RUP that the other variable B cannot also be v since the flag f would have to be both true and false to allow $f \rightarrow (A - B > 0)$ and $\overline{f} \rightarrow (B - A > 0)$ to hold simultaneously. However, there is another alternative, which we

will see is more efficient in some scenarios. Instead of deriving under a sequence of guesses $\mathcal{G}$ that $\wedge\mathcal{G} \vee a_{=v} \rightarrow \overline{b}_{=v}$, we could simply introduce the RUP constraint $\overline{a}_{=v} + \overline{b}_{=v} \geq 1$, independently of the guesses. Propagation of the not equals constraint for v would then follow by simple unit propagation.

4.5 Autotabulation and Other Reformulations

Linear equality constraints can be defined and propagated as two linear inequalities. However, sometimes a solver may wish to achieve generalised arc consistency on a linear equality. This is NP-hard, but is still a good idea sometimes for small variables – for example, if $2X + 2Y + Z = 7$ then Z must be odd, but this will not be inferred from the inequalities. One way a solver might handle such constraints is by automatically turning the two linear inequalities into a table constraint. An implementation of this process might, of course, be buggy, and so we would like to prove that this tabulation is valid, rather than simply defining the table constraint in the pseudo-Boolean model. This is indeed possible, using a more advanced form of the kind of argument previously used to turn a 2D element constraint into a 1D element constraint.

Let us start by finding the set of solutions to the constraint. For each solution, we introduce an extension variable t_s which is true if and only if that solution is selected, in the same way as for a table constraint. We also introduce an extension variable g which is true if and only if at least one of these t_s variables is true. Next we perform and log a backtracking search to find all of the solutions to the constraint, except that we use g as an additional guessed assignment at every stage. At the end of the search, we have a proof that g must be true, which in turn gives us an at-least-one constraint over the t_s variables. We have now created all the constraints we need to define a table constraint.

We expect that similar techniques will be useful for many other kinds of reformulation as well, re-emphasising our ability to prove more than just the core solving process. One modelling technique that likely *cannot* easily be handled this way is symmetry breaking constraints. However, Bogaerts et al. [2] show that a slight extension to the VeriPB proof system would make this possible: this raises the intriguing possibility of taking a symmetry breaking lex or ordering constraint that is defined in a high level model, omitting it from the pseudo-Boolean model, and then efficiently proving that the constraint is in fact valid.

5 An Implementation

We have implemented a basic constraint programming solver which supports proof logging (see supplementary material). Our solver generally follows a conventional design, similar to MiniCP [24], although we have chosen for novelty reasons to make use of some modern C++ features like lambdas and variant types instead of a pure object-oriented design. We were not aiming for sheer speed, and so our solver does not include optimisations like multiple propagation queues, backtrackable variables, stateful or support-tracking propagators, or special handling of binary variables. The solver supports only integer variables, and implements the absolute value, all different, comparison (with half and full reification), element, linear equality and inequality, minimum and maximum, and table constraints. We include example programs implementing four of the five MiniCP benchmarks (a quadratic assignment problem, n-queens, magic series, and magic square; the TSP example is not included because

we do not yet have a circuit constraint), as well as the MiniZinc cake optimisation problem[2], the classic "send more money" and Crystal Maze problems, the world's hardest Sudoku puzzle[3], and an odd-even sum problem using an auto-tabulated GAC sum constraint.

Throughout the development process, we have *not* tried particularly hard to produce a solver which is free from bugs. Instead, our goal is to produce a solver that will not produce undetectable incorrect outputs. The rest of this section describes the key aspects of the solver design that involve proof logging, discusses what we have learned from using the VeriPB system in a constraint programming setting, and evaluates its performance.

5.1 Constraint Compilation, or Why Trust the OPB File?

To create the OPB file, we use a single pass approach, outputting definitions as soon as variables and constraints are generated. Variables are handled centrally, whilst each constraint is responsible for providing its own pseudo-Boolean encoding. OPB creation is done purely using text, and the solver stores only the model line numbers for certain constraints – it does not explicitly store any pseudo-Boolean information.

An obvious difficulty with our proposed process is that this compilation from a CP model to an OPB model is not verified. This is somewhat offset by the deliberate use of extremely simple encodings, but one must ask: "why are the authors so sure that they have designed and implemented the encoding correctly, particularly for fiddly global constraints?". The answer to this question is that we are not sure at all, and so we rely upon a special test system, as follows. For a given constraint, we generate many different possible input domains for its variables. For each set of inputs, we use a small generate-and-test program that provides the full set of solutions to the constraint, making no use of the constraint programming solver or any clever logic or programming. (This can be moderately slow, for example for the element constraint.) We then use the constraint solver to solve the problem consisting of just that constraint on those inputs, and verify that the set of solutions found this way is identical. Finally, we verify the proof produced by this solving process: because this is an enumeration problem, this verifies that the OPB file also has exactly the same set of solutions (and additionally that the propagator found them all legitimately, although this is not the main point of the test).

This process is not perfect, but it does severely reduce the scope for errors: for example, it immediately flagged a typo where a reified greater than or equal constraint had accidentally been implemented as a reified greater than constraint, and a bug when the index constraint for an element constraint contained only out-of-range values.

5.2 Producing the Proofs

Recall that to produce a proof, we need to log our backtracking search, and certain variable-value eliminations. In the design of our solver, we opted for a careful separation of the notion of a variable and its current state: the former we represent as a handle, whilst the latter is stored separately in a central location to allow for easy backtracking. It was therefore natural to force every modification to a variable's values to go through a common set of functions, and to make these functions take a mandatory argument that can be either "no justification needed", "output a RUP statement for this", or "call the following piece of code to produce an explicit set of proof steps". Making this argument mandatory forces constraint authors to think explicitly about justifications, and avoids the potential for illicit modifications to be hiding in places where they can not easily be found by inspecting a proof log.

[2] https://www.minizinc.org/doc-2.5.5/en/modelling.html#an-arithmetic-optimisation-example
[3] https://abcnews.go.com/blogs/headlines/2012/06/can-you-solve-the-hardest-ever-sudoku

For backtracking search, we treat guessing on a branching variable to be a special kind of inference. Outputting the proof log then simply consists of tracing the search as backtracks are performed. Again, at no point was it necessary to manipulate pseudo-Boolean constraints or proof steps as anything other than simple strings created using a template.

5.3 Identifying Solver Bugs

Our experience has been that once the core solver is working and producing proofs for simple problems, it is somewhat more common to have bugs in the proof-producing code for new propagators than in the propagators themselves. Usually these bugs are extremely easy to fix, because VeriPB immediately flags the faulty line of the proof, and our solver can include a comment line immediately above any proof line saying exactly where in its source code that line originated. Similarly, propagator bugs are usually obvious from proof logs. For example, when we first implemented propagation for linear inequalities, we did not yet have a full proof logging setup for variables with large domains. We therefore used a VeriPB feature which allows for unchecked assertions to be included in the proof log (subject to an angry warning being issued at the end of the verification process) so that the remainder of the proofs could be verified. However, our implementation contained a bug, because one of the authors did not realise they did not understand the rules for rounding and integer division when both a variable and its coefficient are negative. Throughout conventional testing on the remainder of the solver, we never saw a single wrong answer being produced by this bug – but as soon as proof logging was implemented, we were told the exact line of code in our solver that was incorrect, even though correct sets of solutions were still being produced. Of course, one could claim that better testing would have identified this, but this relies upon the tester having intimate knowledge of how the propagator works and remembering that integer division of negative numbers could be a source of errors.

It can sometimes be harder to understand the problem when faulty proofs arise from insufficient justifications. For example, for the absolute value constraint, one of the authors had originally lazily assumed that its propagations would follow by a single RUP step in the same way as for not equals – and indeed this is often but not quite always the case. (This experience has left us extremely envious of the skills of authors of lazy clause generation solvers, who are able to write similar propagators without the benefit of machine verification.) This can lead to a proof verification error that only occurs several propagation steps later than the actual bug: the verifier always tells us if something is wrong, but does not always make it trivial to figure out where. However, because our solver forces all propagations to go through a central function call, it is easy to change the way proof logs are written so that all propagations are checked, including those which would usually be implicit.

5.4 Performance and Overheads

Having discussed the design and implementation of proof logging, we now talk about actually using it. This section answers two questions: “does proof logging work at all?”, and “how expensive is proof logging in practice when used on large problems?”.

To answer the second question, we must first establish whether our solver is “fast enough” that its results are likely representative of what would be seen if proof logging were introduced into a mature solver. For MiniCP, Michel et al. [24] include five benchmarks that are designed to test solver speed: they specify an exact search order, and propagation strength for global constraints, so that every solver is producing the same search tree for a fair speed comparison. Their aim was not to have the best model or search for a problem, but rather to benchmark

Table 1 Experimental results from our anonymous solver on six different problem instances. The first four problems are from the MiniCP benchmark suite and have a fixed model, search order, and propagation strength, to allow for a fair comparison between solvers. The final two problems are relatively simple, but use further global constraints that are not supported in MiniCP.

QAP: a quadratic assignment optimisation problem with linear inequalities, not equal constraints, a 2D element constraint, and large variables.						
Runtimes:	MiniCP:	16.9s	OscaR:	7.1s	Choco:	11.3s
	Anon:	5.6s	logging:	149.5s	VeriPB:	232,655.1s
Statistics:	propagators:	355	recursions:	125,805	inferences:	4,521,801
	OPB size:	6.4MBytes	log size:	19GBytes		
	RUP steps:	39,170,568	RPN steps:	413,295	red steps:	101,394
Magic Series: finding the only magic series of length 300, and proving it is unique. Uses linear equality and reified equality constraints.						
Runtimes:	MiniCP:	29.6s	OscaR:	8.8s	Choco:	29.8s
	Anon:	8.2s	logging:	425.2s	VeriPB:	est. 39 days
Statistics:	propagators:	90,301	recursions:	1,193	inferences:	15,584,073
	OPB size:	108MBytes	log size:	12GBytes		
	RUP steps:	7,923,342	RPN steps:	342,401	red steps:	358,800
Magic Square: finding the first 10,000 magic squares of size 5. Uses sum, not equal, and less than constraints.						
Runtimes:	MiniCP:	61.1s	OscaR:	32.3s	Choco:	32.9s
	Anon:	31.0s	logging:	1894.1s	VeriPB:	108,772.8s
Statistics:	propagators:	315	recursions:	6,042,079	inferences:	92,891,165
	OPB size:	145KBytes	log size:	100GBytes		
	RUP steps:	141,528,806	RPN steps:	70,946,952	red steps:	2,550
Queens: finding the first solution to the 88 queens problem. Uses not equals constraints.						
Runtimes:	MiniCP:	876.2s	OscaR:	477.8s	Choco:	438.8s
	Anon:	410.0s	logging:	3450.5s	VeriPB:	60,643.7s
Statistics:	propagators:	11,484	recursions:	49,339,390	inferences:	535,852,330
	OPB size:	8.9M	log size:	104GBytes		
	RUP steps:	50,130,687	RPN steps:	0	red steps:	31,152
Crystal Maze on the usual 8-vertex graph, all solutions. Uses GAC all-different, absolute value, and sum constraints.						
Runtimes:	Anon:	0.01s	logging:	0.13s	VeriPB:	6.3s
Statistics:	propagators:	35	recursions:	259	inferences:	8,737
	OPB size:	60K	log size:	2.6MBytes		
	RUP steps:	32,903	RPN steps:	6,685	red steps:	1,496
With autotabulation and GAC propagation on the sum constraints:						
Runtimes:	Anon:	0.01s	logging:	0.06s	VeriPB:	3.9s
Statistics:	propagators:	52	recursions:	139	inferences:	2,601
	OPB size:	60K	log size:	2.0MBytes		
	RUP steps:	29,467	RPN steps:	102	red steps:	2,958
Sudoku on Arto Inkala's "world's hardest Sudoku puzzle", all solutions. Uses GAC all-different and equals constraints.						
Runtimes:	Anon:	0.03s	logging:	0.05s	VeriPB:	0.52s
Statistics:	propagators:	48	recursions:	103	inferences:	1,388
	OPB size:	320K	log size:	484KBytes		
	RUP steps:	4,561	RPN steps:	460	red steps:	0

solvers performing the same well-defined task. We support enough global constraints (linear inequalities, sum, not equals, reified equals, a special element constraint, and less than) to implement four of these five benchmarks; we do not yet support the circuit global constraint for the fifth. In the first four rows of Table 1 we present computational results from a machine with dual Intel Xeon E5-2697A v4 CPUs, 512GBytes RAM, and a pair of solid state drives in a RAID 0 configuration, running Ubuntu 20.04.3 LTS, and benchmarking against the versions of the other solvers included in the supplementary material provided by Michel et al. In each case our solver is faster than the fastest of MiniCP, OscaR, and Choco, although sometimes only by a few percent. We therefore believe that the results that follow cannot be said to be unfairly optimistic due to the use of a slow solver.

Table 1 also shows runtimes for running our solver with proof logging enabled, together with statistics showing the size of the OPB models and VeriPB proof logs produced, and the number of RUP steps, groups of cutting planes steps (VeriPB works with sequences of cutting planes steps in reverse Polish notation, RPN, rather than one step per line), and redundance-based strengthening steps (red; two such steps are used to introduce an extension variable). On the four MiniCP benchmarks we see a slowdown of between 8.4 and 61.1 from proof logging. This should not be particularly surprising: without proof logging, our solver is making between eight hundred thousand and three million successful inferences per second, and the proof logs to justify these inferences range from ten to over a hundred GBytes in size. Furthermore, our implementation of proof logging is deliberately pessimal. We make use of C++ text output streams for file writing, which are notoriously inefficient. We write comments for most proof log lines generated, we make use of expressive variable names (which require several string concatenation operations and a hash table lookup for each variable written out), and proof lines are manipulated as strings for ease of implementation; all of these decisions are extremely helpful for exploratory research, but not for performance. Finally, these MiniCP benchmarks make use of only relatively simple and extremely fast propagators, which is where proof logging is most expensive. We therefore consider these performance results to be close to a worst case scenario, and would not be surprised if the overheads could be cut by at least a factor of five for some problems if implemented in a production solver that aimed purely for performance rather than for research and teaching.

Returning to the first question, we were able to verify the entire proofs for three of the four MiniCP problems; based upon the first ten percent of the proof for the remaining magic series problem, VeriPB estimates it will take 39 days to verify. We were able to verify entire proofs for smaller instances of the magic series problem. We have also produced and verified proofs for a range of other problems that make heavier use of global constraints – we show two of these in the bottom of Table 1, and other example problems and per-constraint tests are included in our supplementary material. Considering these results, and all the bugs that have been identified during development, we are comfortable in claiming that proof logging can be effective in practice. Although it may not (yet) scale practically to some of the more challenging combinatorial benchmark instances, it is already able to handle moderately sized problems involving several different global constraints, large variables, and reformulation.

6 Conclusion and Future Work

Proof logging gives us a way to trust outputs, not solvers. Trusting solvers seem to be a long way from being a practical reality for for constraint programming: even relatively simple propagators like all different have resisted attempts at formally verified implementation [8] even without the extensive optimisations used by modern solvers [11]. In contrast, we have

shown that, with the right proof format, it is relatively easy to add proof logging to a wide variety of propagators, without requiring the proof verifier to understand anything about constraint programming – and this does not stand in the way of propagator optimisations such as greediness and incrementality.

There is still a lot of work to do before proof logging can be used by everyone all of the time. Firstly, there are many more global constraints and propagators to consider. Most of these will be straightforward, and will re-use existing strategies for proof logging in familiar ways. However, some will not be, and it is an open question as to whether cutting planes with extension variables give a sufficiently strong system to provide practical proof logging in every situation. We expect that recent work in proof logging for symmetry and dominance relations [2] might be necessary to justify certain propagators, as well as for reformulations involving symmetries, and would be interested in a deeper investigation into the relationship between constraint programming propagators and proof systems (with the caveat that "this system polynomially simulates natural deduction and so it can do everything" is not a helpful answer unless the polynomial is of very low order and with small constants).

Secondly, we must think about performance. Using formatted text output and string lookups to produce proof logs is useful for development and exploratory purposes, but for a production solver a better approach is probably needed. Verification performance is also a concern, although we have many reasons to be optimistic that this will improve. For example, very small changes to how proofs are written can give a huge improvement to verification times. We discussed two different ways of proof logging the not equals constraint, one of which involved justifying every propagation subject to the current guessed assignments, and the other which produced new clauses to assist unit propagation. On the MiniCP queens benchmark, using the former would have produced a 1.1TByte proof log that would take an estimated 138 days to verify, rather than a 100GByte proof that could be verified in under a day. If we are prepared to put slightly more cleverness into a solver, and abandon the gratuitous use of RUP steps in favour of a little more logic, we expect that proof sizes for some other constraints can be reduced by a similar factor.

An automated tool that performs proof minimisation would also be useful in this respect. Although potentially expensive to run, this would be very useful for auditability where proofs are to be stored, shared, and potentially verified more than once by different people. Such a tool could also provide annotations that would make RUP steps much quicker to verify – such an approach is already used for formally verified verifiers for DRAT, which actually verify a simplified format called LRAT [6].

On the other hand, relatively slow verification is not a fatal flaw. Proof logging is very good at catching solver bugs that will not be detected by conventional testing, even on relatively small instances. Because the same logic and code paths in a solver can be used whether or not proof logging is enabled, it is a useful feature to support even if it is not enabled all of the time. And, of course, many useful problems with serious real-world consequences derive most of their difficulty from the variety of constraints involved, rather than from being close to the limit of what we can solve computationally.

And thirdly, although we have a reasonably good solution for being confident in our translation from a constraint programming model into OPB, we have not discussed the further difficulty of verifying compilation from high level languages like Essence or MiniZinc. Perhaps it would be worth investigating techniques from formally-verified compilers to help with this translation.

References

1 Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, 2018. doi:10.1007/978-3-319-98334-9_46.

2 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022*, 2022.

3 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.

4 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.

5 William J. Cook, Collette R. Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987. doi:10.1016/0166-218X(87)90039-4.

6 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017. doi:10.1007/978-3-319-63046-5_14.

7 Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 207–223. Springer, 2016. doi:10.1007/978-3-319-44953-1_14.

8 Catherine Dubois. Formally verified constraints solvers: a guided tour. CICM. Invited talk, 2020.

9 Leon Eifler and Ambros M. Gleixner. A computational status update for exact rational mixed integer programming. In Mohit Singh and David P. Williamson, editors, *Integer Programming and Combinatorial Optimization - 22nd International Conference, IPCO 2021, Atlanta, GA, USA, May 19-21, 2021, Proceedings*, volume 12707 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2021. doi:10.1007/978-3-030-73879-2_12.

10 Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1486–1494. AAAI Press, 2020. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5507.

11 Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artif. Intell.*, 172(18):1973–2000, 2008. doi:10.1016/j.artint.2008.10.006.

12 Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, 2019. doi:10.1007/978-3-030-30048-7_33.

13 Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, 2020. doi:10.1007/978-3-030-58475-7_20.

14 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1134–1140. ijcai.org, 2020. doi:10.24963/ijcai.2020/158.

15 Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3768–3777. AAAI Press, 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16494.

16 Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe Conference (DATE)*, pages 10886–10891. IEEE Computer Society, 2003.

17 Warwick Harvey and Joachim Schimpf. Bounds consistency techniques for long linear constraints. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems*, pages 39–46, 2002.

18 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013. URL: http://ieeexplore.ieee.org/document/6679408/.

19 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2013. doi:10.1007/978-3-642-38574-2_24.

20 Linnea Ingmar and Christian Schulte. Making compact-table compact. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 210–218. Springer, 2018. doi:10.1007/978-3-319-98334-9_14.

21 Evelyn Lamb. Two-hundred-terabyte maths proof is largest ever. *Nature*, 545:17–18, 2016.

22 Christophe Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints An Int. J.*, 16(4):341–371, 2011. doi:10.1007/s10601-011-9107-6.

23 Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Comput. Sci. Rev.*, 5(2):119–161, 2011. doi:10.1016/j.cosrev.2010.09.009.

24 Laurent D. Michel, Pierre Schaus, and Pascal Van Hentenryck. MiniCP: a lightweight solver for constraint programming. *Math. Program. Comput.*, 13(1):133–184, 2021. doi:10.1007/s12532-020-00190-7.

25 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.

26 Adrian Rebola-Pardo and Luís Cruz-Filipe. Complete and efficient DRAT proof checking. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018. doi:10.23919/FMCAD.2018.8602993.

27 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.

28 Peter J. Stuckey. Certifying optimality in constraint programming, February 2019. Talk at KTH Royal Institute of Technology.

29 Julian R. Ullmann. Partition search for non-binary constraint satisfaction. *Inf. Sci.*, 177(18):3639–3678, 2007. `doi:10.1016/j.ins.2007.03.030`.

30 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *10th International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008. http://isaim2008.unl.edu/index.php?page=proceedings.

31 Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1754`.

32 Hélène Verhaeghe. *The extensional constraint*. PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2021. URL: `http://hdl.handle.net/2078.1/252859`.

33 Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. `doi:10.1007/978-3-319-09284-3_31`.