A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers

Maarten Flippo ✉ 📷
Delft University of Technology, The Netherlands

Konstantin Sidorov ✉ 📷
Delft University of Technology, The Netherlands

Imko Marijnissen ✉ 📷
Delft University of Technology, The Netherlands

Jeff Smits ✉ 📷
Delft University of Technology, The Netherlands

Emir Demirović ✉ 📷
Delft University of Technology, The Netherlands

Abstract

Proof logging is used to increase trust in the optimality and unsatisfiability claims of solvers. However, to this date, no constraint programming solver can practically produce proofs without significantly impacting performance, which hinders mainstream adoption. We address this issue by introducing a novel proof generation *framework*, together with a CP proof format and proof checker. Our approach is to divide the proof generation into three steps. At runtime, we require the CP solver to only produce a proof sketch, which we call a scaffold. After the solving is done, our proof processor trims and expands the scaffold into a full CP proof, which is subsequently verified. Our framework is agnostic to the solver and the verification approach. Through MiniZinc benchmarks, we demonstrate that with our framework, the overhead of logging during solving is often less than 10%, significantly lower than other approaches, and that our proof processing step can reduce the overall size of the proof by orders of magnitude and by extension the proof checking time. Our results demonstrate that proof logging has the potential to become an integral part of the CP community.

1 Introduction

Constraint Programming (CP) is a pivotal field recognized for its efficacy in addressing intricate combinatorial challenges across diverse sectors. Central to its importance is the *model + solve* paradigm, which allows practitioners to express problem constraints in a

declarative manner without having to reason about *how* the problem needs to be solved. By combining modeling flexibility with powerful solving techniques, CP enables intuitive problem formulation and effective optimisation strategies.

In a typical CP workflow, the solver is often treated as a black box. When a solver reports a solution, it is easy to verify whether all constraints are satisfied. However, for unsatisfiability and optimality claims, no such trivial verification can be done. How can we be sure that a model has no solutions when that is the claim of a solver?

Unfortunately, bugs are common due to the complex nature of solvers, regardless of the paradigm. For example, fuzz testing [41] revealed that virtually all solvers in the MaxSAT Evaluation 2022 have bugs. Similar issues have been reported for SAT and SMT solvers [13]. A variety of techniques have been used to discover bugs in well-maintained CP solvers [4, 29]. Long-standing MIP solvers may report incorrect results due to numerical instabilities [18]. For applications where correctness is crucial, e.g., chip design [1] and combinatorial actions [22], a bug may undermine the entire process and result in legal and financial consequences.

The general approach to verifying claims of unsatisfiability or optimality is *proof logging*. The proof can then be checked by a separate *proof checker*, possibly implemented in a formally verified toolchain. As a result, when a checker accepts a proof as correct, the claim of the solver may be trusted with high confidence.

Proof logging has proven immensely successful in the SAT community [47, 46, 19]. It has enabled the closing of several open math problems [36, 34] and has become standardised in the community, e.g., solvers in the SAT competition [6] must produce proofs in order to participate. Aside from SAT solvers, SMT solvers can also commonly produce proofs [12, 8], and in 2017 the MIP solver SCIP [2] gained the ability to produce machine-checkable proofs for a subset of techniques [16].

Unfortunately, proof logging is far from standard in CP due to several challenges. First, CP solvers consider heterogeneous constraints and reasoning capabilities, making it difficult to select a suitable proof system. Second, as with proof logging for MIP and SMT solvers, the overhead of proof logging may be significant. Third, when models are distributed through standardized file formats, the internal representation of the solver could differ from those models. As a result, none of the state-of-the-art CP solvers submitted to the MiniZinc Challenge [44] and XCSP Competition [5] in 2023 supported proof logging in any form.

The current state of affairs is that proof logging is useful for CP and shows great promise, however, its mainstream adoption is clearly out of reach for current methods. An unpublished approach [27] suggests logging every nogood and propagation step, incurring a three-fold slowdown with potentially prohibitive disk space requirements. Another recent approach proposes to trace the solver using pseudo-Boolean reasoning [31], which leads to high confidence in each individual step of the solver as well as the final claim. Unfortunately, similar downsides hold, with slowdowns of (several) orders of magnitude, and even longer proof checking. Furthermore, both approaches log with respect to the internal problem representation of the solver, rather than a standardised input format.

We address the prohibitive runtime and space requirements for CP proof logging by contributing a novel CP proof generation *framework*, together with a proof format and proof checking facilities. Our approach has three steps. In the first step, the CP solver produces a *proof scaffold* of unsatisfiability by only logging nogoods (learned clauses), which can be understood as a compressed version of the final proof. Once the solving and scaffold production is done, in the second step, our proof processor (a separate piece of software) identifies nogoods in the scaffold that can be removed without invalidating the proof, and

expands the trimmed scaffold into a full CP proof by adding only the necessary inference steps based on the remaining nogoods. In the third step, the proof is verified using one of the available proof-checking approaches. Our framework has several major advantages:

- From the solver (developer) side, we only require generating a proof scaffold, which introduces a minor overhead (5-10% runtime), requires little memory (scaffolds are at most hundreds of megabytes, with most expanded proofs well below one GB – compared to tens or hundreds of GBs of current approaches), and places virtually no burden on the solver developer. All of these points are in stark contrast with previous approaches.
- Our approach reduces the proof size by removing redundant nogoods *before* expanding the proof, which in our experiments removes 20-90% of the nogoods. This is significant given that verification runtime is a bottleneck, which is correlated with proof size.
- The resulting proof is in our *CP proof format*, which, in the spirit of CP, embraces the diversity of constraints and records the proof steps in CP terminology, rather than encoding them into a different paradigm. This allows our approach to be agnostic to the verification technology. In our implementation, we verify the proof using VeriPB [30].
- The proof verification is done with respect to the *input file*, rather than the internal solver representation, decoupling proof generation from verification. In our implementation, we use FlatZinc as the input file, although XCSP and other formats may be used similarly.

We demonstrate the effectiveness of our framework by experimentally evaluating against over 4000 instances from the MiniZinc benchmark repository. These are realistic problems and many have been used in previous competitions. Furthermore, we discuss proof logging FlatZinc components with pseudo-Boolean reasoning, and introduce a novel pseudo-Boolean justification technique for the cumulative global constraint.

To summarise, we present a framework for certifying the output of CP solvers that is 1) efficient enough to be used on realistic problems with little overhead, 2) agnostic of the solver, as the proof steps are given with respect to the input file rather than the internal solver representation, and 3) generic in the sense of our CP proof format which enables using any existing verification approach and reduces the verification time by proving shorter proofs. We believe our results demonstrate that proof logging has the potential to become an integral part of the CP community with a clear value in increasing the trustworthiness of our solvers.

The paper is organized as follows: Section 2 covers some preliminaries and Section 3 covers related work. In Section 4 we present our proof logging approach, and Section 5 presents an empirical evaluation of the approach. Finally, Section 6 gives our conclusion and presents directions for future work.

2   Preliminaries

2.1   Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) $\mathcal{P} = (\mathcal{X}, \mathcal{C}, \mathcal{D})$ is a triple where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of decision variables, $\mathcal{D}$ is the domain, which, for each variable $x_i \in \mathcal{X}$, denotes the set of possible values for x_i as $\mathcal{D}(x_i)$. We consider all variables to be integer variables. Booleans are modelled as integers with domain $\{0, 1\}$, where 1 is the true value, and 0 is the false value. A constraint $C(X) \in \mathcal{C}$ is an n-ary relation between the variables $X \subseteq \mathcal{X}$. *Reified* constraints are of the form $r \leftrightarrow C(X)$, where $r \in \mathcal{X}$ is a Boolean variable which is used to conditionally enforce the constraint $C(X)$. A specific constraint that is important to this work is the *clause*, which is a disjunction of Boolean variables. An *assignment* A is a mapping from each variable x_i to a single value $v_i \in \mathcal{D}(x_i)$. If A satisfies all the constraints, then A is called

a *solution*. If there are no solutions to a CSP $\mathcal{P}$, then $\mathcal{P}$ is called *unsatisfiable*. Otherwise, if $\mathcal{P}$ has at least one solution, the problem is called *satisfiable*. A CSP can be extended to a *constraint optimisation problem* (COP) by adding an objective function $f : \mathbb{Z}^n \mapsto \mathbb{Z}$ which maps any solution A to a cost $f(A)$ such that a solution A_1 is better than solution A_2 iff $f(A_1) < f(A_2)$. A solution A is *optimal* if there does not exist a solution A' with a lower cost.

2.2 Constraint Programming

Constraint Programming (CP) is a paradigm used to solve CSPs and COPs. In a CP solver, constraints are represented through propagators. A *propagator* is a function $p : \mathcal{D} \mapsto \mathcal{D}'$ that removes values from the domain $\mathcal{D}$ which cannot be present in any solution. *Fixpoint propagation* (FP) is the process of applying all propagators until fixpoint, i.e. until no more values can be removed. Solvers recursively partition the domain in at least two subproblems over which it calls fixpoint propagation. This process continues until a subproblem is found where all variables have singleton domains, or the solver concludes no solution exists.

An *atomic constraint* is a predicate $\langle x \diamond v \rangle$, where x is an integer variable, $\diamond \in \{\leq, \geq, =, \neq\}$, and v is a constant. During the search, CP solvers may learn *nogoods*, i.e., a conjunction of atomic constraints that cannot be part of a feasible solution. In this work, we will use the term *learned clause* to mean the negation of a nogood, following conventions from lazy clause generation CP [25, 40] and CDCL-SAT solving [38]. The set of all clauses in the solver is called the *clause database*. The propagation of nogoods/clauses to fixpoint is referred to as *Boolean constraint propagation* (BCP). The notation $\mathcal{C} \vDash_{BCP} l$ means that the atomic constraint l is entailed by the set of clauses $\mathcal{C}$ and this implication is identified through BCP.

2.3 Proof Logging for Black-Box Solvers

In this work, we are primarily concerned with unsatisfiability claims, since a proof of optimality can be understood as an unsatisfiability proof claiming that there is no solution with a lower objective value than the optimal value. This applies directly to linear search (branch-and-bound) commonly used in CP solvers, and a similar approach could be used for other optimisation approaches such as core-guided search [26] and logic-based Benders decomposition [21, 37] (known as implicit hitting sets in MaxSAT [20]).

Conceptually, a proof of unsatisfiability $P = [S_1, \ldots, S_n]$ is a sequence of proof steps, such that each step $S_i \in P$ is entailed by the model and all preceding steps $S_j : 1 \leq j < i$. Every step should be checkable in polynomial time of the number of steps preceding it. The final step of the proof $S_n = \bot$ is the trivial conclusion that the model is unsatisfiable. For solvers, the proof is generally a log of the reasoning steps performed during the search. Note that if a solver finds a solution to the model, it will still have logged its proof steps.

Given a model F and a proof of unsatisfiability P, a *proof checker* certifies that every step in the log is a valid reasoning step. If all the proof steps are correct, then the proof is valid and the model is confirmed to be unsatisfiable. Note that the checker merely asserts the correctness of the proof, and does not provide any guarantees on how the proof was generated. In this sense, an incorrect solver may still produce a correct proof, and as long as the proof is correct, the checker will accept it and assert the claim.

2.4 Proof Logging for SAT

In propositional proofs of unsatisfiability produced by SAT solvers, every step in the proof is a clause which is entailed by the conjunction of the clauses in the problem and the previous clauses in the proof. The entailment has to be identified through *reverse unit propagation*

(RUP) [33, 28], i.e. $\neg c \wedge F \wedge C \vDash_{BCP} \bot$ for a proof clause c, its predecessors in the proof C and the original formula F. The last clause is the empty clause, which cannot be satisfied and is therefore equivalent to $\bot$. Proofs of this style are referred to as RUP proofs. They have the advantage that it is not necessary to indicate what premises lead to the derivation of the clause c. However, a disadvantage is that the runtime cost of checking every clause grows as the number of clauses in the clause database grows.

As the SAT solver runs, it learns many RUP clauses using the CDCL solving framework [38] and adds the clauses to its database. Occasionally, a portion of the learned clauses are deleted. This deletion is often also logged in the proof, extending the RUP format to DRUP (Deletion Reverse Unit Propagation) [35]. The checker can use the deletion information to remove the clause from consideration, to reduce its memory usage and runtime cost. Note, DRUP has been superseded by the modern DRAT [46] and LRAT [19] formats in SAT solvers.

Proof checkers for SAT proofs check the RUP property for every clause in the proof. To do so, there are two possible directions the proof can be traversed: *forward checking* or *backward checking* [33]. In forward checking, the proof is read in one clause at a time. When reading clause C_i, the RUP property is checked and if it holds, C_i is added to the clause database for the checker to use at clause C_{i+1}. Backward checking takes the opposite approach. The entire proof is read into the checker and it starts by checking RUP on the empty clause and moving toward the first clause in the proof. When checking clause C_i, it is first removed from the clause database and then the RUP property is checked. All the clauses from the proof that are used to conclude $\bot$, are marked. Then, as the checker comes across clauses that are not marked, they can be removed from the clause database *without* checking the RUP property, as that clause did not contribute to the derivation of any clause used to derive the empty clause. Essentially, a backward checking proof checker trims the proof it is checking, only verifying proof steps that are used to derive other proof steps.

3 Related Work

Proof logging has been successfully developed for SAT solvers over the last 20 years, and is now adopted by virtually all modern SAT solvers. The introduction of proof logging surfaced many bugs in solver implementations, and the SAT competition, a yearly competition between SAT solvers, requires solvers to be proof-producing. Early proofs were resolution-based proofs [47], but since then the common proof systems are so-called clausal proofs [35, 46, 19].

SMT solvers like Z3 [10] and cvc5 [7] can also produce proofs. These systems are similar to CP solvers in that they have an extremely large number of possible reasoning steps they perform. For any new theory, the reasoning must be individually justified in the proof. The proofs are prohibitively large since the solvers perform a vast amount of reasoning which turns out to be irrelevant to derive unsatisfiability.

A third solving paradigm which has seen work in proof logging is the MIP paradigm. VIPR [16] is a format for MIP proofs using cutting planes reasoning. It is implemented for the SCIP [2] solver. When logging, SCIP has to run with exact arithmetic, which impacts the performance already. On top of that, the large I/O overhead means that the impact of logging the proof limits its applicability.

The first implementation of a proof-producing CP solver was done in the PCS/HaifaCSP solver [45], which produced resolution proofs. For each propagation, the reasoning is explicitly stated in terms of the premises and the conclusion in clausal form to perform the resolution steps. It is unclear what the experimental impact is of this proof logging implementation.

Another promising approach implements proof logging into the Chuffed LCG solver [27]. The proof format can be seen as an extension of the DRUP format, with propagations introduced as temporary clauses. The work is not published, and their experimental data shows a threefold slowdown when proof logging is enabled with potentially large proof files.

A more recent approach to proof logging for CP is done in the Glasgow CP solver [31], which encodes propagations and conflicts into pseudo-Boolean form. This is an elegant approach that decouples the proof format from the capabilities of the CP solver, and new propagators can be supported by devising an encoding of the filtering algorithms into pseudo-Boolean form [24]. It does require the encoding into pseudo-Boolean form [14] to become part of the trusted base of the application, as that component is not formally checked. The key issue, however, is that it imposes a (several) orders-of-magnitude runtime overhead on the solver, as encodings are created during the solving process, and since every propagation is logged, the proof files easily exceed tens and hundreds of GB even for small problems which result in even longer verification times. Although engineering efforts may bring down the runtime, two main issues hinder scalability: 1) CP solvers spend most of their time in the propagation engine, so adding additional work during propagation is likely to negatively impact performance, and 2) only a portion of the work done by the solver is needed for the proof of unsatisfiability. Our framework addresses these issues by removing logging during propagation via scaffolds and expanding only the necessary part of the proof before checking.

Previous CP proof logging approaches show the value of proof logging CP solvers, however, all have similar drawbacks with varying degrees: considerable overhead for the solver, large file requirements to store the proofs, and the proof is done in terms of the internal solver representation rather than a standardised input format.

Finally, ensuring the correctness of solvers has been investigated by correct-by-construction methods [15] using Coq, an automated theorem prover. This removes the necessity of proof checkers, however, the solver is not competitive with implementations in imperative languages, as many of the implementation strategies are difficult to express in formalisms like Coq.

4 Our Contribution: A Multi-Stage Framework for CP Proof Logging

Our framework has three stages: proof scaffolding, proof processing, and proof verification. Before diving into the details, we first show two illustrative examples.

▶ **Example 1.** On the left, an unsatisfiable CSP is given. The table on the right shows one possible proof for why the CSP is unsatisfiable. Note that the there is another, shorter proof that could have been found.

$$x, z \in \{0, 1\}, y \in \{0, 1, 2\}$$
$$c_1 : 2x + y + 2z \geq 2$$
$$c_2 : 2x + y - 2z \geq 0$$
$$c_3 : 2x - y + 2z \geq 0$$
$$c_4 : 2x - y - 2z \geq -2$$
$$c_5 : -2x + y + 2z \geq 2$$
$$c_6 : -2x + y - 2z \geq 0$$

#	Implied by	Proof step
1	c_3	$\langle x \leq 0 \rangle \wedge \langle y \geq 2 \rangle \rightarrow \langle z \geq 1 \rangle$
2	c_4	$\langle x \leq 0 \rangle \wedge \langle y \geq 2 \rangle \rightarrow \langle z \leq 0 \rangle$
3	1, 2	$\langle x \leq 0 \rangle \wedge \langle y \geq 2 \rangle \rightarrow \bot$
4	c_1	$\langle x \leq 0 \rangle \wedge \langle y \leq 1 \rangle \rightarrow \langle z \geq 1 \rangle$
5	c_2	$\langle x \leq 0 \rangle \wedge \langle y \leq 1 \rangle \rightarrow \langle z \leq 0 \rangle$
6	3, 4, 5	$\langle x \leq 0 \rangle \rightarrow \bot$
7	c_5	$\langle x \geq 1 \rangle \rightarrow \langle z \geq 1 \rangle$
8	c_6	$\langle x \geq 1 \rangle \rightarrow \langle z \leq 0 \rangle$
9	6, 7, 8	`UNSAT`

▶ **Example 2.** Let us consider the three-queens problem. The problem states that we have to place three queens on a three-by-three chessboard, such that no two queens are in the same row, column, or on the same (anti)diagonal. On the left is a model for the three-queens problem, and on the right a proof for why it is unsatisfiable.

$$q_1, q_2, q_3 \in \{1, 2, 3\}$$
$$c_1 : \texttt{Distinct}(q_1, q_2, q_3)$$
$$c_2 : \texttt{Distinct}(q_1 + 1, q_2 + 2, q_3 + 3)$$
$$c_3 : \texttt{Distinct}(q_1 - 1, q_2 - 2, q_3 - 3)$$

#	Implied by	Proof step
1	c_1	$\langle q_1 = 1 \rangle \to \langle q_2 \neq 1 \rangle$
2	c_1	$\langle q_1 = 1 \rangle \to \langle q_3 \neq 1 \rangle$
3	c_2	$\langle q_1 = 1 \rangle \to \langle q_2 \neq 2 \rangle$
4	c_1	$\langle q_2 = 3 \rangle \to \langle q_3 \neq 3 \rangle$
5	c_3	$\langle q_2 = 3 \rangle \to \langle q_3 \neq 2 \rangle$
6	1-5	$\langle q_1 = 1 \rangle \to \bot$
7	c_1	$\langle q_1 = 2 \rangle \to \langle q_2 \neq 2 \rangle$
8	c_2	$\langle q_1 = 2 \rangle \to \langle q_2 \neq 3 \rangle$
9	c_3	$\langle q_1 = 2 \rangle \to \langle q_2 \neq 1 \rangle$
10	7-8	$\langle q_1 = 2 \rangle \to \bot$
11	c_1	$\langle q_1 = 3 \rangle \to \langle q_2 \neq 3 \rangle$
12	c_1	$\langle q_1 = 3 \rangle \to \langle q_3 \neq 3 \rangle$
13	c_2	$\langle q_1 = 3 \rangle \to \langle q_2 \neq 2 \rangle$
14	c_1	$\langle q_2 = 1 \rangle \to \langle q_3 \neq 1 \rangle$
15	c_3	$\langle q_2 = 1 \rangle \to \langle q_3 \neq 2 \rangle$
16	6, 10-15	`UNSAT`

In Example 2 the solver can derive nogoods (steps 6 and 10) based on the propagations it performs. In the example, these nogoods are all singleton, i.e. they contain one atomic constraint. However, nogoods can contain an arbitrary number of atomic constraints, as shown in Example 1.

We proceed with describing each of our stages, referring to these examples as appropriate.

4.1 Verification Flow

One of the main barriers to wider adoption of CP proof logging is the significant potential overhead on the solver in terms of runtime and memory, and possibly the effort required from the solver developer to enable proof logging. Our aim is to address both of these concerns.

Since proof production may have a significant overhead on the solving process, we aim to log as few steps as possible during the search. As a CP solver can potentially log many proof steps, especially when logging individual propagations, proofs can grow prohibitively large. It may also be that the overhead of logging turns out to be wasted, for example when a satisfaction problem turns out to be satisfiable or when the solver never concludes optimality in an optimisation problem. We therefore aim to log as few steps as possible during the solving phase to keep the runtime cost introduced by proof logging as low as possible.

Given this motivation, we split the proof production into multiple phases. Note that while each phase may introduce bugs, a sound checker will only ever accept valid proofs, and bugs due to unsound reasoning will be detected. A schematic overview of the following steps in proof production is given in Figure 1.

1. **Scaffolding** The first step is to run the solver. During the solving process, the solver produces a *proof scaffold*, which is a list of nogoods identified by the solver.

   The proof scaffold of Example 2 would consist of two single-predicate nogoods, $\langle q_1 \neq 1 \rangle$ (step 6) and $\langle q_1 \neq 2 \rangle$ (step 10), and the unsatisfiability claim (step 16). For Example 1, the proof scaffold would consist of proof steps 3, 6, and 9.

   The requirement is that the i-th proof step can be recovered by contradiction in polynomial time by the next stage considering the original problems and the first $(i-1)$ proof steps (similar to RUP), i.e., negating the nogood leads to a conflict by propagation.
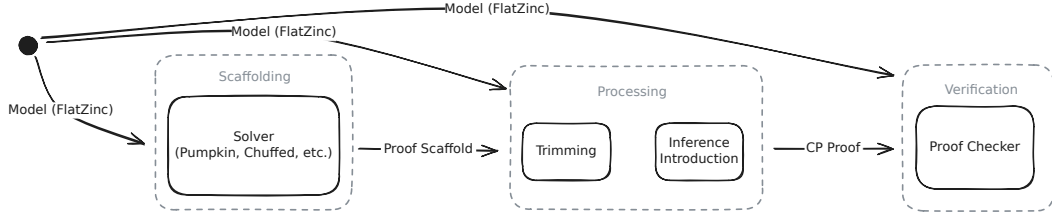
Figure 1 Schematic overview of our multi-stage framework for CP proof logging.

2. **Processing** The proof scaffold is completed into a full proof by the proof processor. Here we describe what the processing step provides to the framework, and Section 4.3 presents the algorithm to achieve it. Conceptually, there are two transformations the processor runs on the scaffold:

   Trimming A proof scaffold will likely contain many nogoods that are unnecessary to derive unsatisfiability. For SAT solvers, proof trimming is a vital part of proof checking [35], and our experiments in Section 5 show the same holds for CP proofs. Given that the next step can potentially introduce many steps to the proof, the key idea is that we trim before inference introduction.

   In the presented three-queens example, trimming would not remove any of the two nogoods in the proof. However, the solver could also have derived the nogood $\langle q_2 \neq 1 \rangle$, meaning the proof scaffold would have been $\langle q_1 \neq 1 \rangle, \langle q_2 \neq 1 \rangle, \langle q_1 \neq 2 \rangle$. Had this been the case, then trimming would have removed the middle nogood since it is redundant in proving unsatisfiability.

   Inference Introduction For every nogood C in the trimmed proof scaffold, the necessary inferences (i.e. propagations) relevant to concluding C are generated. The output of the inference introduction stage is considered to be a full CP proof. It explains, in CP terms, the reasoning steps required to prove unsatisfiability.

   Note that this implies two points. First, the processor needs to be able to reason at least as strongly as the solver to accept the proof scaffold as a valid proof. Second, the processor is free to provide reasoning that is different from the reasoning used by the solver used during runtime, as long as it reaches the same conclusion.

   In Example 2 and Example 1, the introduced inferences are the proof steps which are implied by constraints from the model, rather than any previous proof steps.

3. **Verification** After proof processing we get what we refer to as the full *CP proof*. It is ready to be checked by a proof checker. Ideally the checker is formally verified, to leave as little doubt in the conclusion as possible.

4.2   Proof Format

The proof format we propose reasons about the CP model in terms of atomic constraints and nogoods over those atomic constraints. The format is valid for both the proof scaffold and the full CP proof. A step in a valid CP proof in this format is one of the following:

Nogood ⟨nogood id⟩ ⟨clause⟩ A clausal representation of a nogood, which satisfies the RUP property. In a solver which derives nogoods using the CDCL [43] algorithm, all nogoods satisfy the RUP property. A trivial nogood which can be logged by non-learning CP solvers consists of current decision variable assignments when a conflict is detected. In Example 2, steps 3 and 6 are nogoods.

Deletion ⟨nogood id⟩ The nogood with the given id is no longer relevant for BCP, and can be removed from consideration in the checker. This step is required to reduce the memory usage and BCP runtime. In an LCG solver, as in SAT solving, it is common to occasionally delete nogoods that are deemed unworthy by the solver to save space and propagation time. Therefore, such nogoods cannot contribute to later propagations (unless they are learned again), and therefore we can help a proof checker by indicating the deletion.

Inference ⟨name⟩ ⟨premises⟩ ⟨propagated atomic constraint⟩ A lemma which is enforced by a particular propagator in the solver. It states the premise that leads to the given propagated atomic constraint. The inference may be used in checking the RUP property of a nogood. For instance, in Example 2, step 1 is an inference given by constraint c_3, and it is used when checking the nogood in step 3.

The inference is required to satisfy an extension of the RUP property, i.e., asserting the premises and the negation of the propagated atomic constraint leads to a conflict by *fixpoint propagation*. Given this requirement, the "name" is not strictly necessary – as in the scaffolding phase, this could be left freely to the processor to determine the reason. Nevertheless, the name serves as a hint to simplify the verification step and in our evaluation, the name was necessary to select the proper encoding for the inference. Typically a CP solver performs many propagations during search. Keeping all inferences in the proof in memory would slow down BCP when checking a nogood. Hence, we specify that all inferences only apply to the first nogood following it. After checking that nogood, the inferences are deleted from memory. This means the same inference may be present multiple times in the proof file, but it avoids having to delete inferences explicitly.

Conclusion ⟨conclusion⟩ The final step of the proof. The value of ⟨conclusion⟩ can be one of the following:

- `UNSAT` when the conclusion is that the problem is unsatisfiable,
- a literal corresponding to the atomic constraint ⟨*objective* ≥ *v*⟩, where v is the value of the objective variable in the optimal solution.

Conceptually, this format extends the DRUP [33] format. The nogoods in the proof are the RUP clauses which are checked identically to the RUP clauses in a DRUP SAT proof, and the inferences are temporary problem clauses which are created by the propagators in the CP solver. An interesting property of our approach is that for pure SAT problems, i.e., problems containing only clauses and Boolean variables, our approach resembles standard SAT proof logging and introduces no overhead compared to SAT proof logging, contrary to other CP proof logging approaches.

4.3 Proof Processor

Given a CP proof scaffold, the processing stage can now apply trimming and inference introduction. The proof processor uses a backward checking approach [33] to implement trimming and inference introduction in a single pass.

The pseudo-code for the proof processor is given in Algorithm 1. The proof is loaded, the empty clause is marked as used, and the output proof P' is initialized to an empty sequence. The proof processor then goes over the scaffold back to front. The last clause is popped from the input proof sequence, and if it is marked, the RUP property is checked. Unmarked clauses are skipped immediately. The PROPAGATION(φ) procedure propagates all constraints until the fixpoint. In case of no conflict, the implication graph is empty and the checker cannot complete the proof scaffold, as the producer of the scaffold performed reasoning the

■ **Algorithm 1** Proof Processing. F is a set of constraints, and P is the proof scaffold.

Require: The top-most element of P is $\emptyset$.
Ensure: P' is the full CP proof.
1: $\varphi \leftarrow F \cup P$
2: $marked \leftarrow \{peek(P)\}$　　　　　　　　$\triangleright$ *peek*: Get top-most element in P without popping
3: $P' \leftarrow \emptyset$
4: **while** $|P| > 0$ **do**
5:　　$c \leftarrow pop(P)$
6:　　$\varphi \leftarrow \varphi \setminus \{c\}$
7:　　**if** $c \in marked$ **then**
8:　　　　$P' \leftarrow push(P', c)$
9:　　　　$confl \leftarrow \text{PROPAGATION}(\varphi \wedge \bigwedge_{l \in c} \neg l)$　　　　$\triangleright$ Perform the RUP check.
10:　　　　**if** *confl* is empty **then**
11:　　　　　　**return** Fail
12:　　　　**end if**
13:　　　　$nogoods, inferences \leftarrow \text{ANALYZE}(confl)$
14:　　　　$marked \leftarrow marked \cup nogoods$
15:　　　　$P' \leftarrow P'$ **concatenated with** *inferences*
16:　　**end if**
17: **end while**
18: $P' \leftarrow \text{REVERSE}(P')$
19: **return** Success

---

processor cannot imitate. In case of a conflict, the RUP property for the nogood holds and the PROPAGATION(φ) procedure returns an implication graph describing the conflict. The ANALYZE(G) procedure walks the implication graph backwards. Through this traversal, the nogoods used to derive the conflict are marked, and the propagations performed by propagators are introduced as inferences. Finally, after processing all the nogoods in the scaffold, the output proof P' contains the full CP proof in reverse, so it is reversed.

4.4   Checking Proofs

Once we obtain the full CP proof, the final step is to check the proof with respect to the model. A possible approach is implementing a formally verified proof checker with a theorem prover. This approach was suggested in previous work [27] and would fit the framework well. Another approach, which we adopted in our evaluation, is to encode the CP proofs to a pseudo-Boolean format and check the proofs with VeriPB, following the Glasgow solver [31].

For this paper, the description of a CP model is given in the FlatZinc format. Therefore, the models need to be encoded to a pseudo-Boolean formula. Then, every proof step in the CP proof is encoded to an equivalent pseudo-Boolean justification.

4.4.1   FlatZinc Encoding

Encoding the FlatZinc model into a pseudo-Boolean formula entails encoding the variables, the constraints, and, if present, the objective variable. Following the narrative from [31], we encode the FlatZinc variables with a binary encoding. For example, if variable x has a domain $D(x) = [0..5]$, the binary encoding will have three Boolean variables $a_3 a_2 a_1$, and the binary expansion of x is $1 \times a_1 + 2 \times a_2 + 4 \times a_3$. Next, we focus on three types of constraints in this work:

Linear Inequalities A linear inequality is a constraint in the form $\sum w_i x_i \geq c$, where $w_i, c \in \mathbb{Z}$ are integer constants and $x_i \in \mathcal{X}$ are integer variables. These constraints are encoded as described in [31].

Reified Linear Inequalities A reified version of the linear inequality constraint. We rewrite constraints of the form $r \leftrightarrow \sum w_i x_i \geq c$ as a big-M constraint in the pseudo-Boolean model; more formally, this constraint is introduced as $M \neg r + \sum w_i x_i \geq c$ and $Mr - \sum w_i x_i \geq 1 - c$, where M is a suitably large integer constant. Encoding this constraint is now the same as encoding a regular linear inequality. Additionally, any justification for an inference implied by a reified linear inequality is rewritten to a linear bounds justification over the big-M constraint.

Cumulative To encode the cumulative constraint [3] we used Equations 2–3 from Bofill et al. [11].

Finally, if the FlatZinc model defines an objective variable $\mathcal{O}$, the FlatZinc objective function is given as the binary expansion of $\mathcal{O}$.

4.4.2 Atomic Constraint Encoding

The pseudo-Boolean encoding of the proof starts by introducing the atomic constraints as pseudo-Boolean variables. We achieve this by reifying the corresponding condition with redundance-based strengthening [32]. For example, an atomic constraint $\langle x \geq v \rangle$ is encoded by introducing a new pseudo-Boolean variable (which we introduce with the same notation) and deriving two big-M constraints $M \neg \langle x \geq v \rangle + x \geq v$ and $M \langle x \geq v \rangle - x \geq -v + 1$ for large enough values of M.

Given the semantics of integer variables and values, every proof *implicitly* includes the constraints enforcing the consistency of the atomic constraints, including $\forall x \in \mathcal{X}, v \in \mathbb{Z}$: $\langle x \geq v \rangle \rightarrow \langle x \geq v + 1 \rangle$ or $\forall x \in \mathcal{X}, v \in \mathbb{Z}$: $\langle x \leq v \rangle \rightarrow \langle x \leq v - 1 \rangle$. Therefore, after declaring the atomic constraints, we introduce the constraints above with VeriPB inferences. For example, a condition $\langle x \geq u \rangle \rightarrow \langle x \geq v \rangle$ for constants u, v such that $u \geq v$ is introduced by taking the half-reification definitions of form $M \neg \langle x \geq u \rangle + x \geq u$ and $M \langle x \geq v \rangle - x \geq -v + 1$, adding them and dividing with rounding up by the big-M constant. The resulting condition reads as $\neg \langle x \geq u \rangle + \langle x \geq v \rangle \geq 1$, which is precisely the desired implication.

4.4.3 Proof Encoding

After introducing the atomic literals, the encoder unpacks each CP proof step into valid VeriPB statements. The nogood and deletion steps correspond to VeriPB RUP and deletion steps respectively. Hence, we only discuss the encoding of the inference steps.

To support the constraints listed in Section 4.4.1 we have to justify two kinds of inferences: linear bound reasoning for linear inequalities, as used in Example 1, and time-table [9] reasoning for the cumulative constraint. As stated previously, reified linear inequalities are rewritten to regular inequalities and the justification for reified linear inequality inferences is regular linear bound reasoning. A pseudo-Boolean justification of inferences implied by linear inequalities is described by [31]. Hence, we only describe the time-table justification here.

Let $\langle s_i \geq v \rangle \wedge \langle s_i \leq v' \rangle \rightarrow \langle s_j \geq w \rangle$ be a time-table inference, where variables s_i and s_j model the start time of activities i and j, respectively. This inference states that the bounds on variable s_i are such that activity j cannot be scheduled before time point w, because, task i must consume a certain amount of resource in an interval ending at time point $w - 1$ and scheduling variable s_j before w would guarantee to overload the resource.

The justification is done by contradiction, i.e., we assume activity j is scheduled before time point w. From the encoding of the constraint, we have Boolean literals corresponding to $\langle s_i \leq t < s_i + p_i \rangle$ for every time point t, where $p_i \in \mathbb{Z}$ is the constant duration of activity i. These literals indicate that task i is active at time point t. The premises of the inference correspond to setting these literals to true for the time points within the bounds described by the inference. The negation of the propagated atomic constraint also leads to these literals becoming true in the time interval ending at time point w. Pseudo-Boolean propagation then identifies the conflict in resource usage, so the inference is valid.

Finally, when the proof conclusion is the optimal objective value, the encoder will first identify the lower bound $\langle x \geq v \rangle$ on the objective variable x from the conclusion of the proof. The constraint $\neg \langle x \geq v \rangle$ is then appended to the encoding of the FlatZinc model. The rest of the proof is treated as proof of unsatisfiability for the amended model.

5   Evaluation

To show our method works on a standardised input format, we use MiniZinc [39] models. We designed experiments to evaluate our framework in three aspects:

- The overhead caused by logging the scaffold in terms of runtime. We aim to keep this low, to increase the practical use of the framework. We show that logging takes most of the time less than 10%, which is orders of magnitude lower compared to existing approaches.
- The value of scaffolding, trimming, and expansion, compared to logging all proofs steps as in other approaches. Processing the proof should not be prohibitively expensive in runtime and disk space. We demonstrate that our approach is indeed computationally inexpensive with notable reductions in proof size.
- The time spent to verify the proofs. For proof logging to be adopted, verifying the proofs should not take an unreasonable amount of time. Our current runtimes are promising, in particular for optimisation problems.

We implemented our approach in our LCG CP solver Pumpkin[1]. We support most FlatZinc primitives and the cumulative global constraint. The solver is competitive to established solvers Gecode [42] and Chuffed [17] (see Figure 2) when using the globals constraints that we support and decomposing others. This gives credibility to our results.

We use our solver within the proof processor. Note that while proofs may be potentially produced with incorrect solvers, the final proof is verified with an external tool which gives us high confidence in the results, e.g., the solver is not checking itself.

Our main focus is on proof production, nevertheless we also implemented our own checker as a proof-of-concept. The checker is based on VeriPB, and certifies that the proof is correct with respect to the FlatZinc file. Our checker implements a subset of FlatZinc primitives and the cumulative with time-table filtering, as described in Section 4.4.

We considered all MiniZinc benchmarks comprising over 4000 instances, however depending on the experiment, we used a different subset (around 3000 instances). These are detailed in the appropriate sections.

All the experiments were run on the DelftBlue [23] supercomputer. Each benchmark used a single core of an Intel Xeon E5-6248R 24C 3.0GHz processor and 12GB of RAM. The solver was given 10 minutes, and subsequent steps in the framework were only performed on instances that did not time out. Any time measurements given are of CPU time.

---

[1] Available at `https://github.com/consol-lab/pumpkin`. The version used for experiments is stored on Zenodo (see title page).
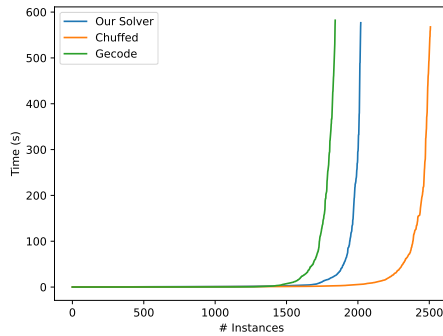
Figure 2 A cactus plot of Gecode, Chuffed and our solver on instances of the MiniZinc benchmarks. Only completely solved instances are shown.
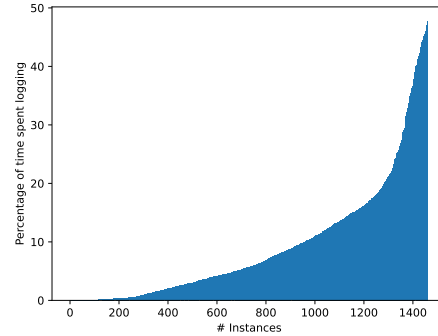
Figure 3 Distribution of logging as a percentage of the solving time for all the instances taking longer than 10 seconds.

5.1 Proof Logging Overhead

The solver was instrumented to measure the CPU time spent within the proof logging. Given the capabilities of our solver, we considered 3923 MiniZinc benchmarks. Note that we reduced the number of instances for the `mrcpsp` and `rcpsp` families by 80% by removing instances randomly, since these instances had an overwhelming presence compared to other benchmark families. The comparison with other solvers was done on this set of benchmarks as well.

Table 1 The time spent logging as a percentage of the total runtime. After discarding instances solved under 10 seconds, 1462 instances remain.

Minimum	Median	Geometric Mean	Mean	Maximum
0.01%	5.70%	3.29%	9.33%	47.78%

We discarded easy problems from the evaluation which were solved within ten seconds, leaving us with 1462 instances. Figure 3 shows the distribution of the overhead, and this information is aggregated in Table 1. Logging takes, on average, less than 10% of the runtime. This is a significant improvement compared to the orders of magnitude slowdown reported by previous methods. Even the worst case only presents a 47.78% overhead, but that is on an instance which finishes so quickly the I/O is the main bottleneck. We note that the overhead can be notably reduced by using a better implementation when writing to the file, e.g., we produce plain text files, whereas outputting in binary form would be more efficient. Nevertheless the logging time even in the current form remains low.

Our experiments highlight the advantage of this approach, and the reason for logging only the derived nogoods becomes clear. Every derived nogood typically requires many propagations, and a slowdown by a multiple of what is presented here is inevitable when the propagations are logged as well during solving.

5.2 Proof Processing

Aside from proof production, important metrics in this evaluation are the cost of proof processing and the benefit in terms of proof size reduction. We focus on instances that we support with our checker that we could completely solve with our solver, i.e., where the

solver concluded unsatisfiability or optimality. In total, we considered 3382 benchmarks and generated a total of 1825 proofs from three models from the MiniZinc benchmark repository[2]:

RCPSP: an optimisation model using the cumulative global constraint and linear inequalities. This model has 1784 instances which are solvable to optimality within 10 minutes.

2DPacking: an optimisation model with linear (in)equality constraints, with 10 instances solvable to optimality within 10 minutes.

Market Split: a satisfaction model with only linear equality constraints, and has 20 instances that are solved within 10 minutes and are unsatisfiable.

Figure 4a shows how many nogoods remain in the proof after trimming. The plot shows only 574 proof scaffolds since the remaining 1251 instances are solved with an empty scaffold, i.e., instances were determined unsatisfiable at the root level. It is notable that for 373 instances, from both RCPSP and 2DPacking, not a single nogood remains after trimming. The final proof only consists of inferences, and the trimmer used the bound on the objective to identify the required inferences and discarded all nogoods. This means that these instances with an infeasible lower bound on the objective variable can be identified through propagation alone, without any search. We note that these 373 instances had scaffolds ranging from a single nogood to 987,272 nogoods.

On the market split instances, we see different behaviour. For these instances, trimming did almost nothing. There is one instance for which 1% of the nogoods were removed, but on all the other instances not a single nogood could be trimmed away. Based on the impact of trimming on DRUP proofs for SAT solvers [35], we expected to see at least some impact. We hypothesise this is not the case because the instances are relatively small, all solved within 3 seconds. It is therefore easy to find the unsatisfiability, and the solver never ends up exploring redundant parts of the search space.

Given the benefits of the explicit processing step in terms of proof size, we now discuss the cost in terms of extra runtime. Figure 4b presents the time taken to process every proof compared to the time it took to solve that instance. From the figure we can see that processing time is between an order of magnitude quicker and an order of magnitude slower than solving the model, however, most of these orders of magnitude differences are due to very small runtimes. Out of the instances which take the longest to solve, all take less than 100% of the solving time to be processed. Since we are dealing with many instances which are solved quickly, and given that noise in the timing data is more pronounced on shorter time spans, we conclude that instances with long solving times give a good upper bound to the processing time. This aligns with our expectations: we expect processing time to take less time than solving for reasonably sized problems.

5.3   Proof Checking

As described in Section 4.4, we implemented an encoder for the proof so they can be checked by VeriPB. This means we spend some time encoding the proofs. Depending on their size, this encoding time is not negligible.

Encoding a proof is linear in the number of proof steps. Every step in the CP proof can be encoded independently from the others, and encoding a single step is cheap. On instances with only a few 100 propagations and few nogoods, the I/O required in the encoder dominates the time. However, on instances which take longer than 2 seconds to solve, the median encoding time is 77% of the solving time.

---

[2] Available at `https://github.com/MiniZinc/minizinc-benchmarks/tree/26bcd0a`
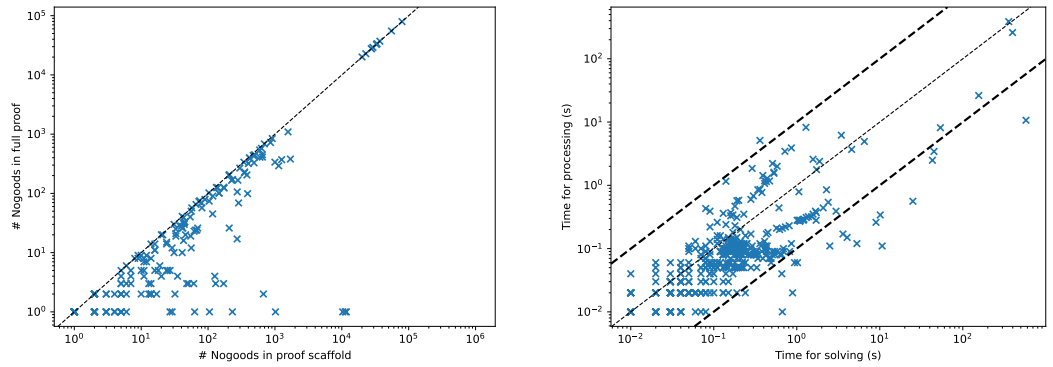
(a) Effect of trimming on the number of nogoods. This does not include the 373 instances which had no remaining nogoods after trimming.

(b) The time taken to process the proof compared to the time it took to produce the scaffold.

■ **Figure 4** Experimental results on the processing of proofs.

The final step to fully certifying the solutions is to look at what VeriPB does with the encoded proofs. Figure 5a plots, for every proof, how long it took to verify relative to the time it took to produce the scaffold. For some instances, the checking time is orders of magnitude slower than producing the proof. In particular, this is the case for the market split instances. This is likely because for optimisation problems the solver performs upper-bounded linear search, which means it starts out deriving nogoods as it finds improving solutions. Only after the last solution does the solver start to conclude optimality, which means many of the first nogoods can be removed. For satisfaction problems, this does not happen, and therefore more nogoods from the scaffold are relevant, which means checking the proof takes longer relative to the solving time compared to optimisation instances.
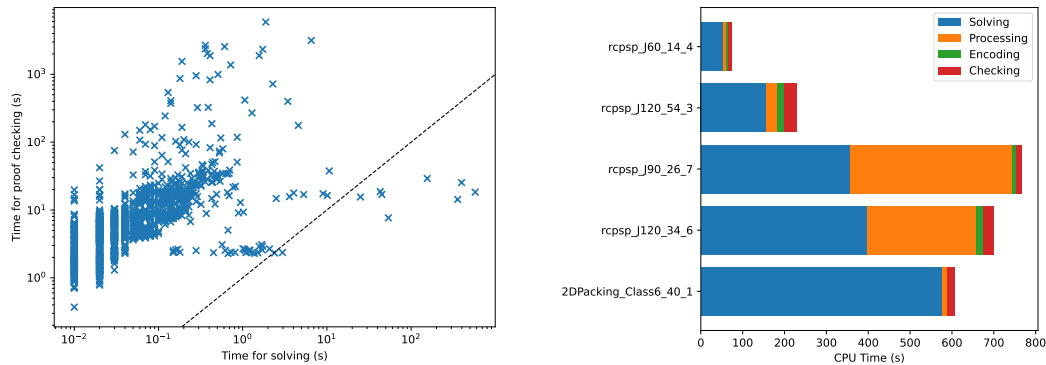
(a) Time taken to check an instance compared to solving the instance with proof generation.

(b) Solving to checking for the five instances that took the longest to solve. This excludes market split instances because they are solved too quickly.

■ **Figure 5** Experimental results on the checking of proofs.

To give an idea of the breakdown of the total time it takes for the proof to be accepted, starting with the solving time, Figure 5b shows the time taken at each stage for the five instances that took the longest to solve. We focus on the long instance since these instances are likely to run into scalability issues with the other methods. We note that this excludes any market split instances because they are all solved in less than 3 seconds, and at that

scale, the timing data is extremely noisy. However, precisely these instances highlight why our method is practically usable. Four out of the five instances are solved in more than 150 seconds, and the `rcpsp_J60_14_4` instance is solved in approximately 53 seconds. All of the instances are optimisation problems, and again we can see that checking whether a claim is really true *can* be extremely cheap compared to solving the problem.

6 Conclusion and Future Work

We have presented a framework to support proof logging in CP solvers, given a FlatZinc model as an input. It addresses problems with large proof sizes and significant runtime overhead by delaying as much of the proof construction for as long as possible: only until the necessity of the proof becomes clear does one have to pay the price to obtain the proof. The proof format itself is designed to be almost arbitrarily extensible, in acknowledgement of the versatility of CP solvers and their propagation algorithms.

Through our framework, we can feasibly certify unsatisfiability and optimality claims in practical settings, with minimal overhead compared to when the solver is not logging a proof. This minimal overhead is not trivial to achieve. On optimisation models, solvers may derive many facts which do not directly relate to proving optimality, which adversely affects proof checking times. By removing the reasoning steps which are redundant for the optimality claim, at least RCPSP and 2DPacking optimality proofs, can be checked quickly.

Future work includes expanding the number of supported constraints in the MiniZinc library. These include robust implementations of the rest of the FlatZinc builtins and would be followed by implementing MiniZinc globals other than cumulative. Supporting stronger reasoning would likely drastically shorten the proofs which would benefit not just solving and processing times, but also checking times. Furthermore, we would like to expand support to other formalisms such as XCSP, engage with solver developers to further reduce practical barriers, and include other techniques such as preprocessing in the proof format.

Another avenue to pursue is whether the design and implementation of a dedicated, formally verified checker for the CP proof format can reduce the end-to-end overhead even more, which was shown to be promising [27]. If the proof no longer needs to be encoded, and the checker can implement CP-specific reasoning natively, we postulate the checking will become cheaper and more accessible.

References

1   Tobias Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, October 2007.

2   Tobias Achterberg. SCIP: solving constraint integer programs. *Math. Program. Comput.*, 1(1):1–41, July 2009. `doi:10.1007/s12532-008-0001-1`.

3   Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. In Jean-Paul Delahaye, Philippe Devienne, Philippe Mathieu, and Pascal Yim, editors, *JFPL'92, 1ères Journées Francophones de Programmation Logique, 25-27 Mai 1992, Lille, France*, volume 17, page 51, 1992. `doi:10.1016/0895-7177(93)90068-A`.

4   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, Cham, 2018. Springer. `doi:10.1007/978-3-319-98334-9_46`.

5   Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2023 XCSP3 competition. *CoRR*, abs/2312.05877, 2023. `doi:10.48550/arXiv.2312.05877`.

6   Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions.* Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2023.

7   Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442, Cham, 2022. Springer. `doi:10.1007/978-3-030-99524-9_24`.

8   Clark Barrett, Leonardo De Moura, and Pascal Fontaine. Proofs in satisfiability modulo theories. *All about proofs, Proofs for all*, 55(1):23–44, 2015.

9   Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, pages 63–79, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. `doi:10.1007/3-540-46135-3_5`.

10   Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. Satisfiability modulo custom theories in Z3. In Cezara Dragoi, Michael Emmi, and Jingbo Wang, editors, *Verification, Model Checking, and Abstract Interpretation - 24th International Conference, VMCAI 2023, Boston, MA, USA, January 16-17, 2023, Proceedings*, volume 13881 of *Lecture Notes in Computer Science*, pages 91–105, Cham, 2023. Springer. `doi:10.1007/978-3-031-24950-1_5`.

11   Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. Compact mdds for pseudo-boolean constraints with at-most-one relations in resource-constrained scheduling problems. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 555–562, 2017. `doi:10.24963/ijcai.2017/78`.

12   Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156, Berlin, Heidelberg, 2009. Springer. `doi:10.1007/978-3-642-02959-2_12`.

13   Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57, Berlin, Heidelberg, 2010. Springer. `doi:10.1007/978-3-642-14186-7_6`.

14   Sam Buss and Jakob Nordström. Proof complexity and sat solving. *Handbook of Satisfiability*, 336:233–350, 2021. `doi:10.3233/FAIA200990`.

15   Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. A certified constraint solver over finite domains. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 116–131, Berlin, Heidelberg, 2012. Springer. `doi:10.1007/978-3-642-32759-9_12`.

16   Kevin K. H. Cheung, Ambros M. Gleixner, and Daniel E. Steffy. Verifying integer programming results. In Friedrich Eisenbrand and Jochen Könemann, editors, *Integer Programming and Combinatorial Optimization - 19th International Conference, IPCO 2017, Waterloo, ON, Canada, June 26-28, 2017, Proceedings*, volume 10328 of *Lecture Notes in Computer Science*, pages 148–160, Cham, 2017. Springer. `doi:10.1007/978-3-319-59250-3_13`.

17   Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, a lazy clause generation solver. URL: `https://github.com/chuffed/chuffed/`.

18   William Cook, Thorsten Koch, Daniel E Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, 2013. `doi:10.1007/s12532-013-0055-6`.

19   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, Cham, 2017. Springer. `doi:10.1007/978-3-319-63046-5_14`.

20   Jessica Davies and Fahiem Bacchus. Exploiting the power of mip solvers in maxsat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 166–181. Springer, 2013. `doi:10.1007/978-3-642-39071-5_13`.

21   Toby O. Davies, Graeme Gange, and Peter J. Stuckey. Automatic logic-based benders decomposition with minizinc. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 787–793, 2017. `doi:10.1609/aaai.v31i1.10654`.

22   Sven De Vries and Rakesh V Vohra. Combinatorial auctions: A survey. *INFORMS Journal on computing*, 15(3):284–309, 2003. `doi:10.1287/ijoc.15.3.284.16077`.

23   Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 1). `https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1`, 2022.

24   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*, volume 34, pages 1486–1494. AAAI Press, April 2020. `doi:10.1609/aaai.v34i02.5507`.

25   Thibaut Feydy and Peter J Stuckey. Lazy clause generation reengineered. In *International Conference on Principles and Practice of Constraint Programming*, pages 352–366. Springer, 2009. `doi:10.1007/978-3-642-04244-7_29`.

26   Graeme Gange, Jeremias Berg, Emir Demirović, and Peter J Stuckey. Core-guided and core-boosted search for cp. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21–24, 2020, Proceedings 17*, pages 205–221. Springer, 2020. `doi:10.1007/978-3-030-58942-4_14`.

27   Graeme Gange, Geoffrey Chu, and Peter Stuckey. Certifying optimality in constraint programming. unpublished, 2017. URL: `https://github.com/gkgange/cert-cp`.

28   Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008. URL: `http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf`.

29   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, Cham, 2019. Springer. `doi:10.1007/978-3-030-30048-7_33`.

30   Stephan Gocht, Ciaran McCreesh, and Jakob Nordstrom. Veripb: The easy way to make your combinatorial search algorithm trustworthy. *From Constraint Programming to Trustworthy AI*, 2020.

31   Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *DROPS-IDN/v2/document/10.4230/LIPIcs.CP.2022.25*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.CP.2022.25`.

32   Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-boolean proofs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(5):3768–3777, May 2021. `doi:10.1609/aaai.v35i5.16494`.

33   Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10886–10891. IEEE Computer Society, March 2003. `doi:10.1109/DATE.2003.10008`.

34   Marijn Heule. Schur number five. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(11), April 2018. `doi:10.1609/aaai.v32i1.12209`.

35   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, October 2013. `doi:10.1109/FMCAD.2013.6679408`.

36   Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245, Cham, 2016. Springer. `doi:10.1007/978-3-319-40970-2_15`.

37   John N Hooker and Greger Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96(1):33–60, 2003. `doi:10.1007/s10107-003-0375-9`.

38   Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 133–182. ios Press, 2021. `doi:10.3233/FAIA200987`.

39   Nicholas Nethercote, Ralph Stuckey, Peter J. Becket, Sebastian Brand, Gregory J. Duck, and Guido Tac. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543. Springer, 2007. `doi:10.1007/978-3-540-74970-7_38`.

40   Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14:357–391, 2009. `doi:10.1007/s10601-008-9064-x`.

41   Tobias Paxian and Armin Biere. Uncovering and classifying bugs in maxsat solvers through fuzzing and delta debugging. In Matti Järvisalo and Daniel Le Berre, editors, *Proceedings of the 14th International Workshop on Pragmatics of SAT co-located with the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Alghero, Italy, July 4, 2023*, volume 3545 of *CEUR Workshop Proceedings*, pages 59–71. CEUR-WS.org, 2023. URL: `https://ceur-ws.org/Vol-3545/paper5.pdf`.

42   Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode - generic constraint development environment. URL: `https://www.gecode.org/`.

43   João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pages 220–227. IEEE Computer Society / ACM, November 1996. `doi:10.1109/ICCAD.1996.569607`.

44   Guido Tack and Peter J. Stuckey. Minizinc challenge 2023. URL: `https://www.minizinc.org/challenge2023/results2023.html`.

45   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*, volume 24, pages 204–209. AAAI Press, July 2010. `doi:10.1609/aaai.v24i1.7543`.

46   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, Cham, 2014. Springer. `doi:10.1007/978-3-319-09284-3_31`.

47   Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10880–10885. IEEE Computer Society, March 2003. `doi:10.1109/DATE.2003.10014`.